Langage Assemblage et Jeu d'Instructions

Cours (9x2h00) :

- Frédéric Mallet - fmallet@unice.fr

TP (9x2h00 - 1 groupe) :

- Frédéric Mallet - fmallet@unice.fr

http://deptinfo.unice.fr/~fmallet/laji/

Structure du cours

- Performances : évolution et comparaison
- Codage de l ’information
- Fonctions logiques et éléments mémoires
- Systèmes à microprocesseurs
- La famille Intel - 80x86
- La famille IBM - UltraSPARC
- Éléments avancés (parallélisme, mémoire, ...)
- Conception d'architectures numériques - VHDL

Historique

- La structure actuelle des processeurs est le résultat de la convergence de plusieurs domaines :
  - Machines à calculer (machine de Pascal)
  - Technologie : roue, interrupteurs mécaniques puis électroniques (tube à vide, transistor)
  - Logique : Boole, Shannon, Turing
  - Commande automatique : Jacquard, Hollerith

Ordinateur

- Définition
  - Machine “universelle” capable d'exécuter un algorithme quelconque (accepte des entrées et calcule des sorties)
- Instructions machines
  - L'algorithme est décomposé en instructions élémentaires
    - Calculs arithmétiques et logiques, accès aux données, branchement
- Jeu d'instructions
  - Ensemble des instructions que peut exécuter un processeur
- Programme
  - Une séquence d'instructions forme un programme

Architecture de l'ordinateur

- C'est l'étude de son organisation et de sa conception, elle a deux composantes
- Architecture du jeu d'instructions (ISA)
  - Interface entre le logiciel et le matériel
    - Ensemble des instructions
    - Type des données
    - Organisation de la mémoire et entrée/sorties
- Architecture de l'organisation matérielle (HSA)
  - Le processeur, flots de données, structure de bus

HSA : Modèle de Von Neumann

- L'ordinateur est composé :
  - Unité de calcul : UAL
  - Mémoire :
    - contient le programme (pas câblé = Von Neumann)
    - contient les données
  - Unité de contrôle :
    - Chef d'orchestre (E.g. Multiplieur)
    - Contient en particulier une unité d'adressage pour sélectionner la prochaine instruction à exécuter

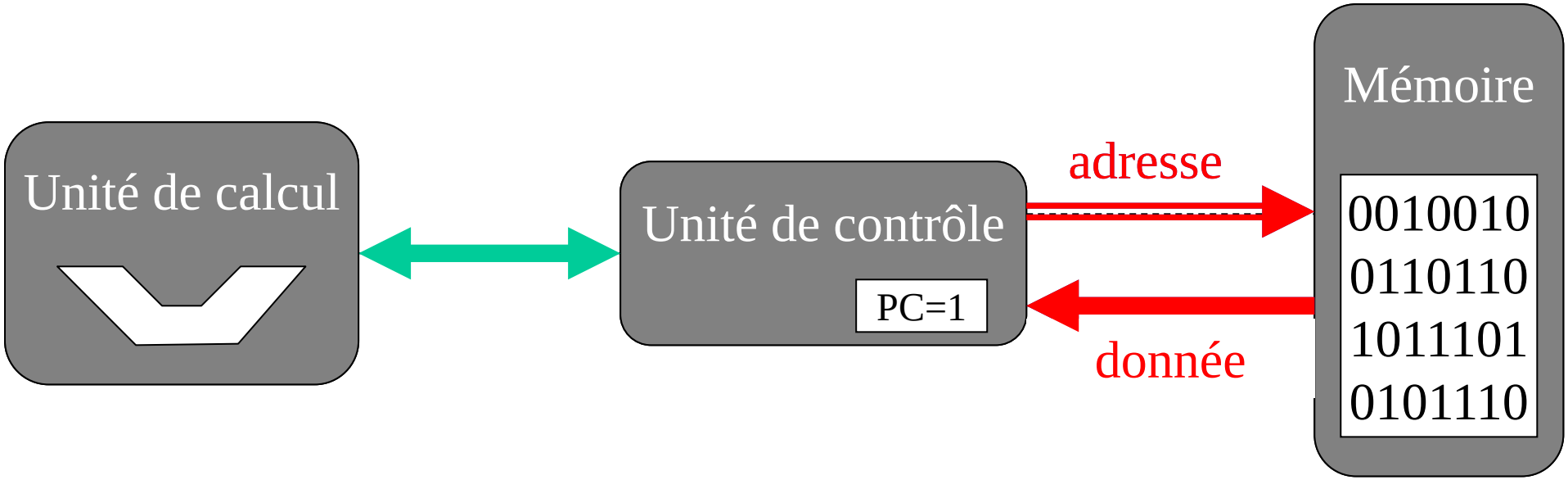
ISA : Assembleur

- Les instructions sont stockées en mémoire sous forme de 0 et de 1 : langage machine
- Les langages de plus haut niveau facilitent la programmation
  - Des compilateurs permettent la traduction d'un niveau vers un niveau inférieur
  - Langage assembleur
    - Association 1-1 entre les instructions machines et les instructions assembleurs (e.g. add r1,r2,r3 ≡ 0x0C010203)
  - Langages de haut niveau plus proches de l'anglais (C)

Exécution d'une instruction

- L'exécution d'une instruction se décompose en plusieurs phases cadencées par l'horloge :
  - L'instruction suivante est lue en mémoire : Fetch
    - L'unité de contrôle contient un registre spécial PC (Program Counter) = @ prochaine instruction
  - La mémoire envoie l'instruction à l'unité de contrôle qui l'analyse : Decode
    - Nature de l'instructions (calcul, branchement) et récupère les opérandes éventuels
  - Le processeur effectue l'opération : Execute
  - Le résultat est mémorisé avant de passer à l'instruction suivante : Write Back

Le modèle de Von Neumann



Fetch **Decode** **Execute**

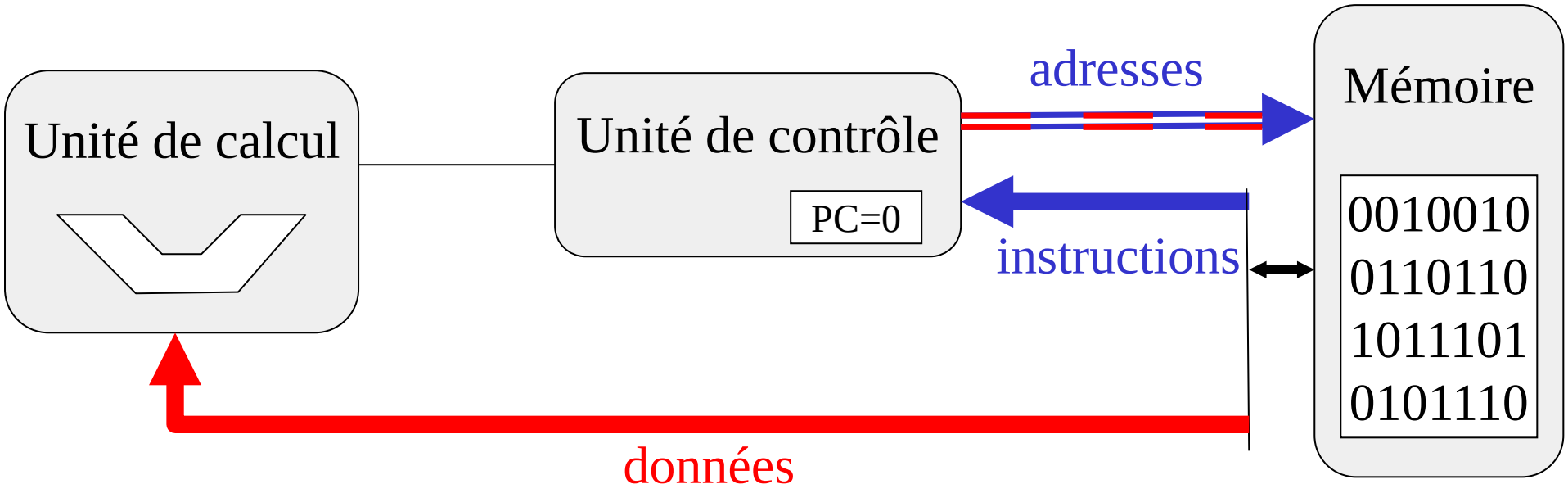
Exécution d'une instruction (suite)

- La phase de décodage peut durer plusieurs cycles
  - Si les chemins de données ne sont pas assez larges
    - `add r1,r2,r3` : nécessite de lire 2 données et d'en écrire 1
- La phase d'exécution peut
  - Durer plusieurs cycles : calculs complexes (multiplication)
  - Durer 0 cycle : branchement
- La phase de write back peut être vide
  - Si aucun résultat ne doit être écrit

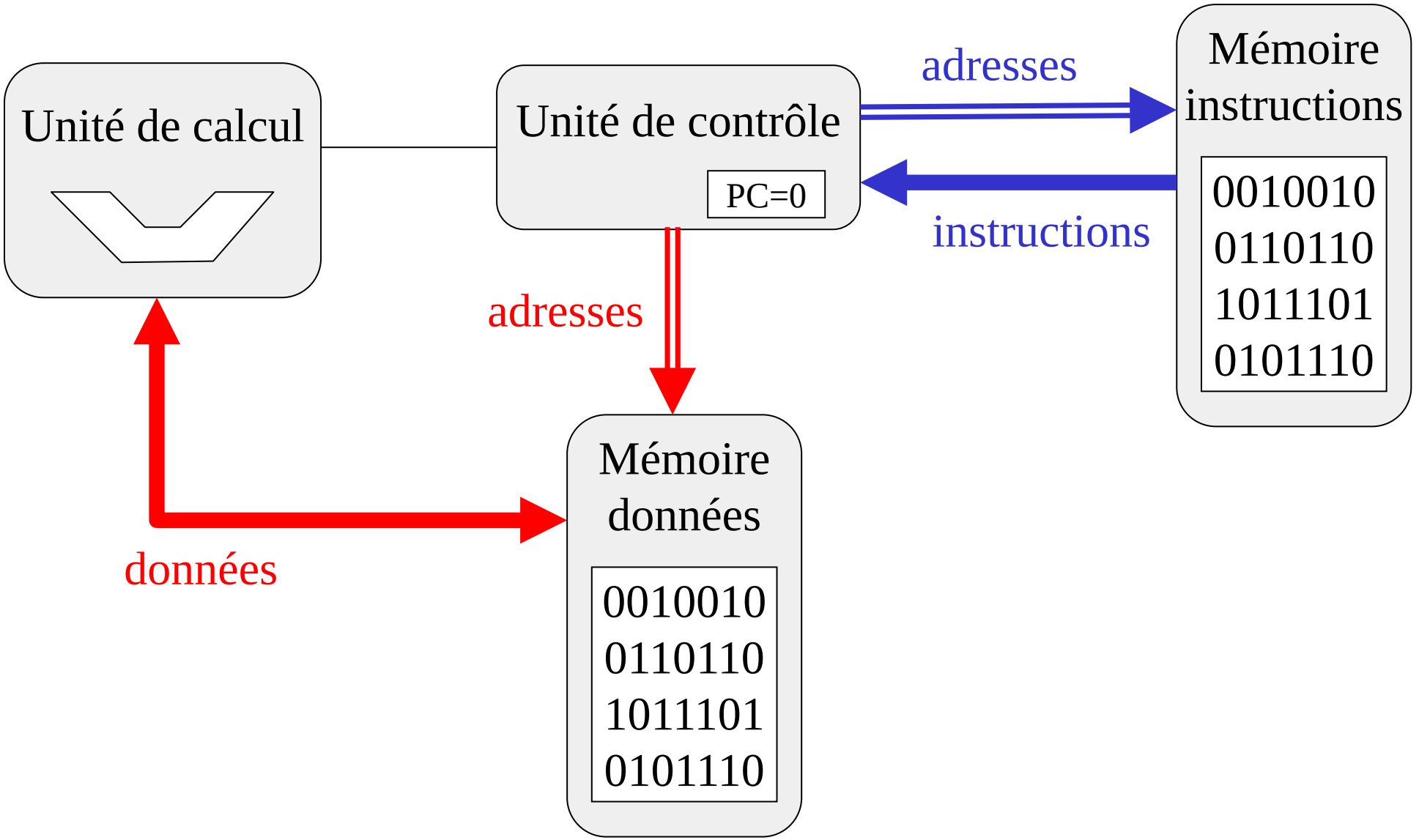
Architectures à mémoire de données

- Modèle de Von Neumann
  - 1 seule mémoire qui contient les instructions et les données
  - 1 bus d'adresse, 1 bus (bidirectionnel) de donnée
  - Logique complexe pour lire/écrire tour à tour les données et les instructions
- Modèle Harvard
  - 2 mémoires : 1 pour les instructions, 1 pour les données
  - 2 bus d'adresse, 2 bus (bidirectionnels) de donnée

Le modèle de Von Neumann



Le modèle de Harvard

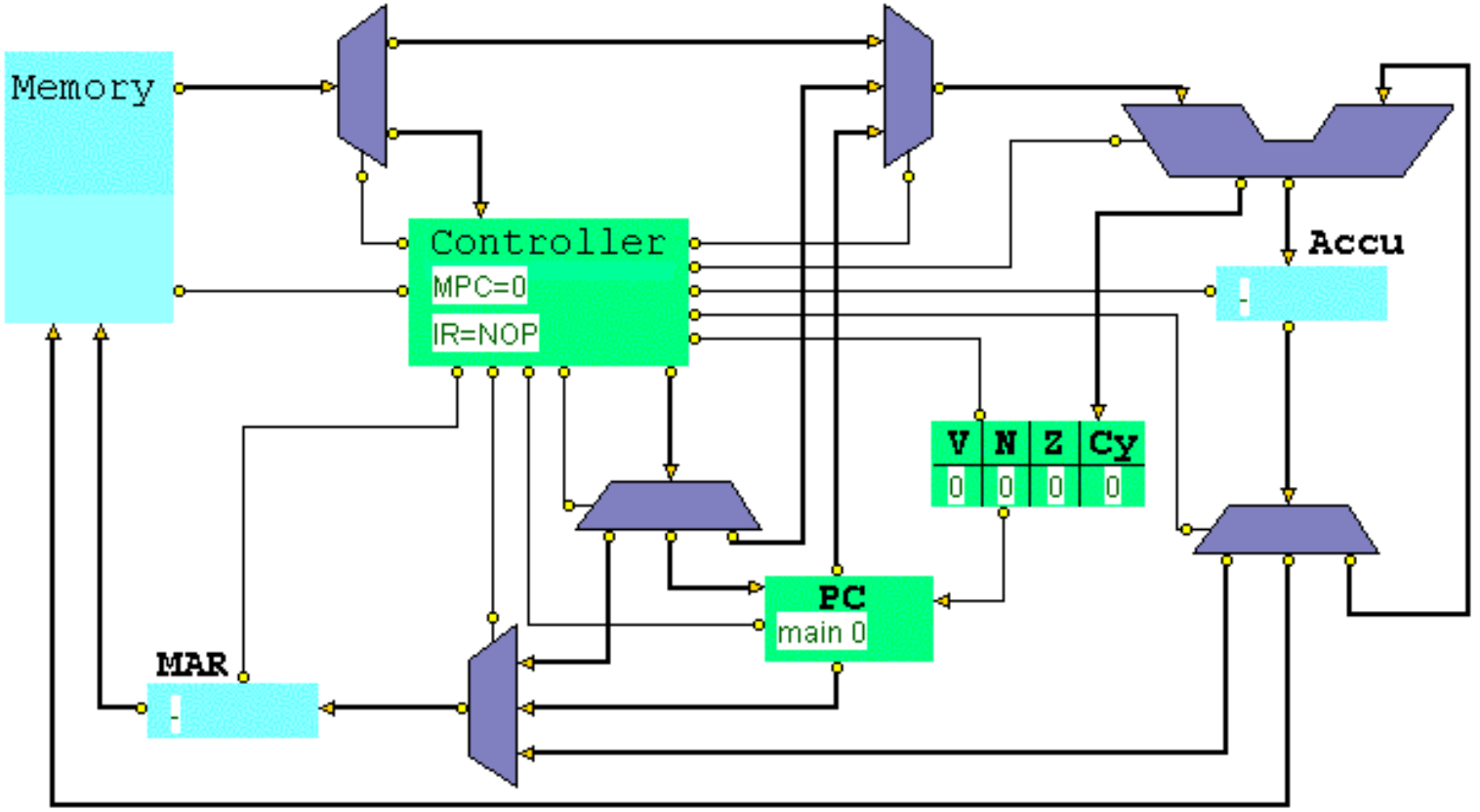


Étude de cas

Architecture à accumulateur MicroAccu

Modèle Von Neumann

MicroAccu – chemins de données



- Les calculs sont accumulés dans l'accumulateur (registre)
- Les données proviennent de la mémoire

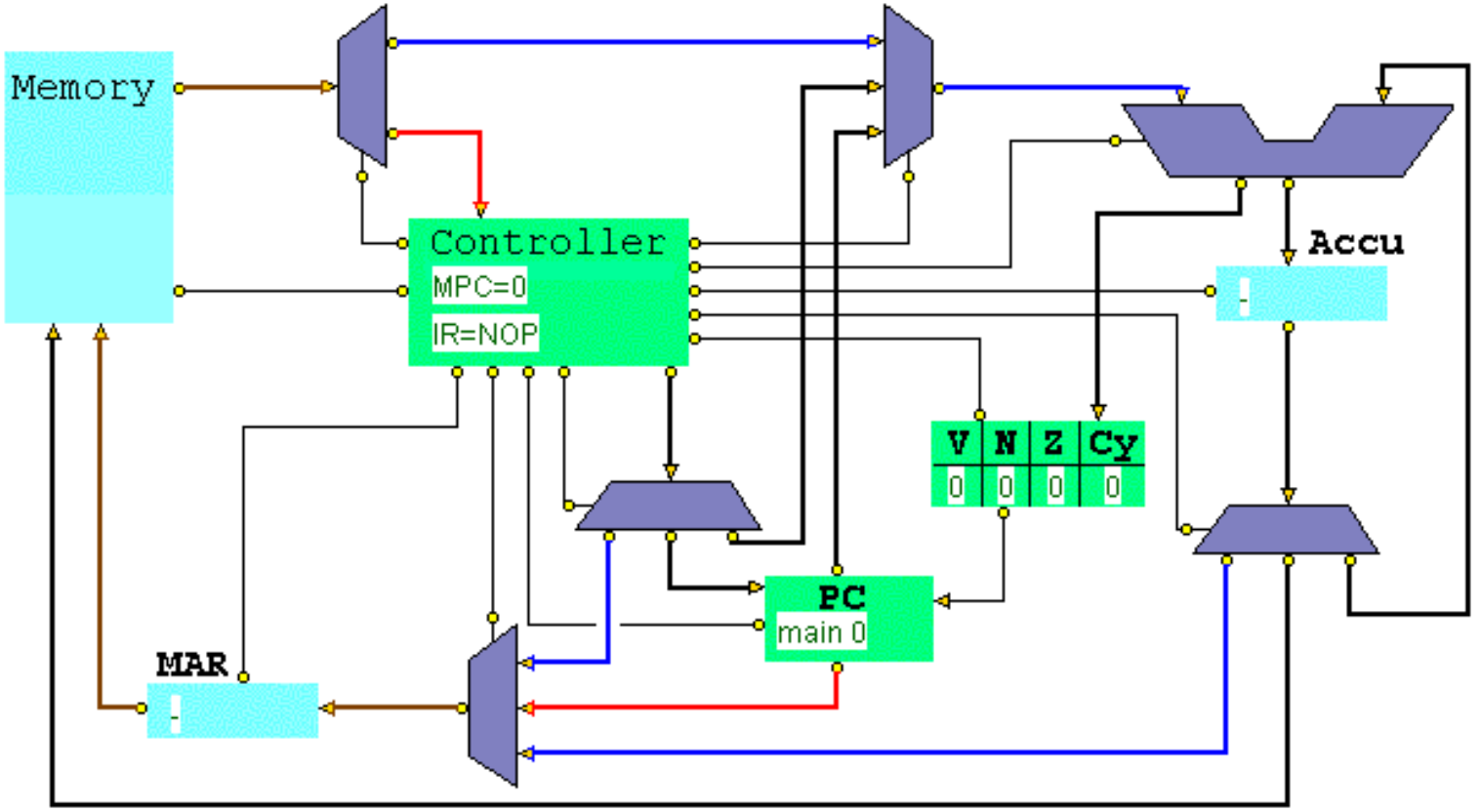
Mémoire d'instructions/données

- Les données et les instructions sont dans la mémoire
- 3 commandes :
  - READ (adresse) *produit une donnée en sortie*
  - WRITE (adresse) *écrit la donnée d'entrée*
  - FETCH (adresse) *produit une instruction en sortie*
- En pratique READ = FETCH :
  - zone de donnée (.DATA), zone d'instructions (.CODE)

Registre d'adresse : MAR

- Le registre PC (**P**rogram **C**ounter) contient l'adresse de la prochaine instruction à exécuter
  - Certaines architectures l'appel IP (**I**nstruction **P**ointer)
- Le registre MAR contient l'adresse de la prochaine donnée/instruction à lire en mémoire:
  - PC (pour le FETCH)
  - Donnée immédiate (Branchement absolu : BR 2)
  - Donnée calculée (Branchement relatif : BR PC+5)
  - Ce registre n'est pas accessible directement par l'utilisateur

Lecture donnée/instruction



- Générer la bonne adresse : charger le registre MAR
- L'instruction vers le décodeur, la donnée vers l'UAL

Jeu d'instructions : LOAD/STORE

- Toutes les opérations se font avec l'accumulateur
- Toutes les instructions ont un opérande

- LOAD adr : *accu = MEM[adr]* adressage direct
  - Charge la donnée à l'adresse *adr* vers l'accumulateur
- STORE adr : *MEM[adr] = accu*
  - Mémorise l'accumulateur à l'adresse *adr*

- LOADI valeur : *accu = valeur* adressage immédiat
  - Charge la donnée *valeur* vers l'accumulateur

Modes d'adressage

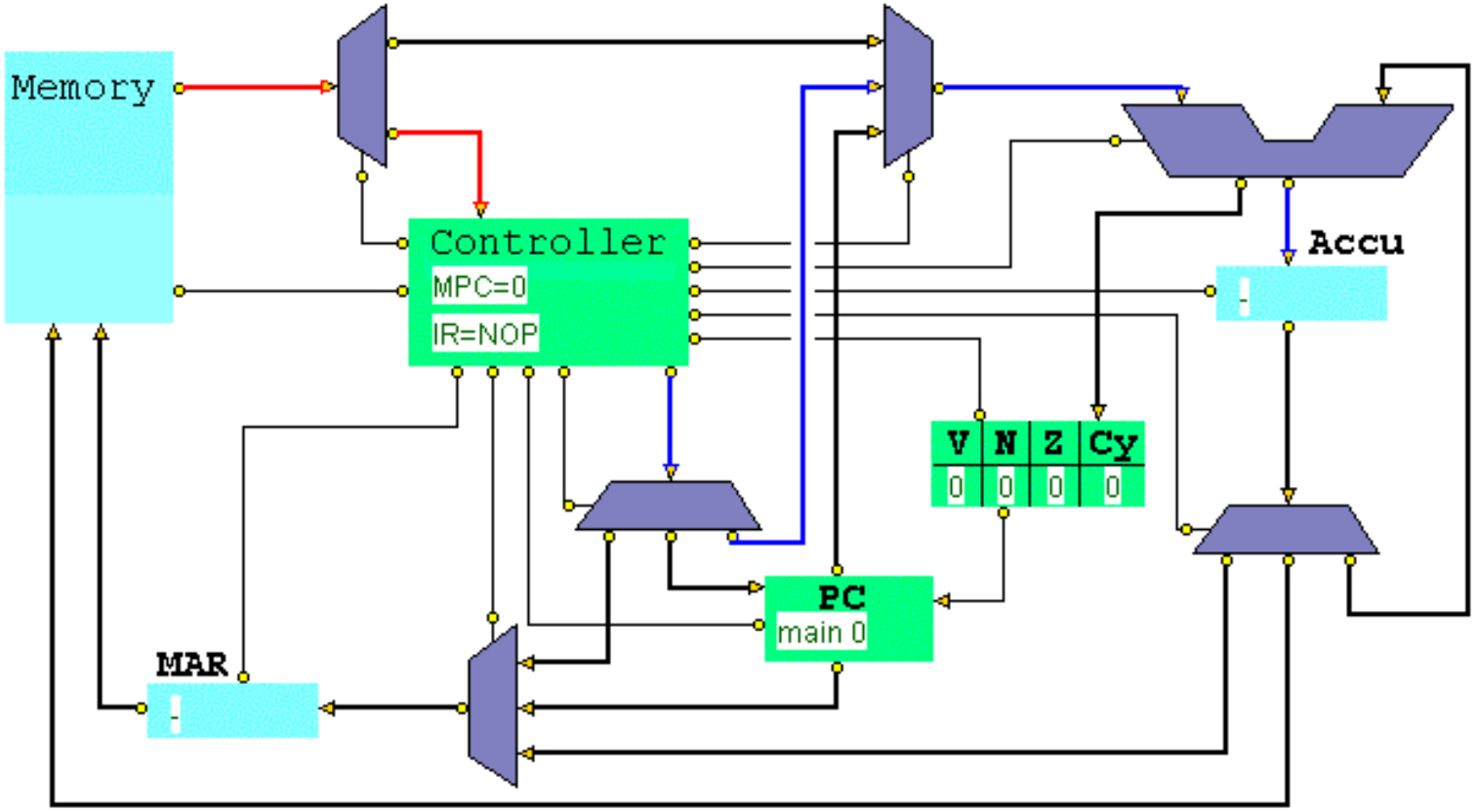
- Adressage immédiat
  - On donne la donnée à utiliser
    - LOADI 2 : charge 2 dans l'accumulateur

      accu = 2
- Adressage direct
  - On donne l'adresse de la donnée à utiliser
    - LOAD 2 : charge la donnée à l'adresse 2 dans l'accumulateur

      accu = MEM[2]
    - Il faut lire la donnée en mémoire, puis charger l'accumulateur

Adressage immédiat

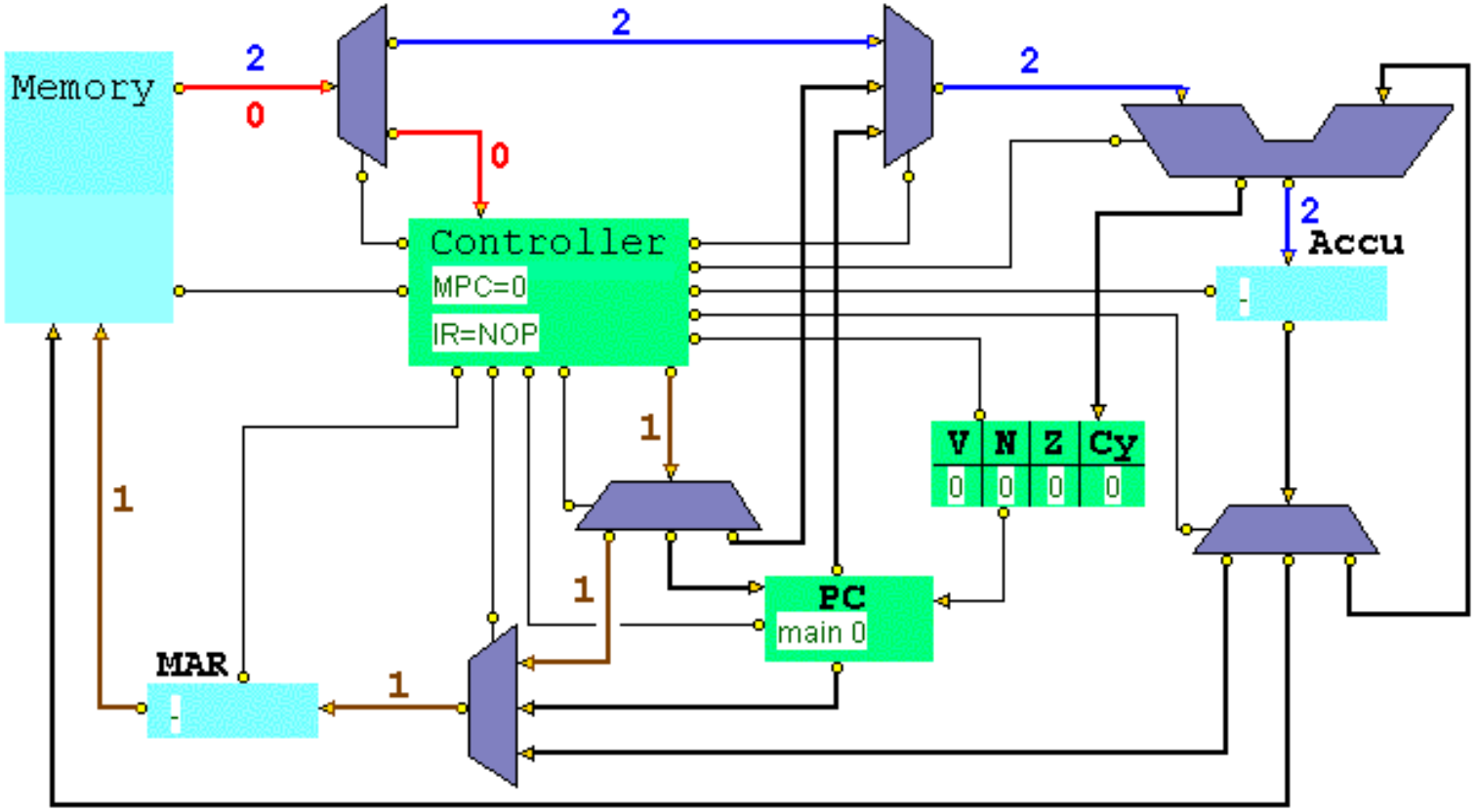
LOADI 2



- La donnée inclue dans l'instruction est dirigée vers l'accumulateur

Adressage direct

LOAD 2



- L'adresse inclue dans l'instruction est dirigée vers MAR
- Puis la donnée de la mémoire vers l'accumulateur

Opérations arithmétiques

- Toutes les opérations utilisent l'accumulateur
- 2 étapes
  - Accumuler l'opérande 1 dans l'accumulateur
    - ALU a une opération LD
  - Fournir l'opérande 2 pour faire l'opération arithmétique souhaitée (ADD, MUL, DIV, ...)

Jeu d'instruction

- **Adressage immédiat ou direct**
- Opérations arithmétiques

ADDI a :	Accu = Accu + a	ADD a :	Accu = Accu+MEM[a]
SUBI a :	Accu = Accu – a	SUB a :	Accu = Accu - MEM[a]
MULI a :	Accu = Accu * a	MUL a :	Accu = Accu * MEM [a]
DIVI a :	Accu = Accu / a	DIV a :	Accu = Accu / MEM[a]

- Dans tous les cas :
  - N est positionné ssi le résultat est négatif
  - Z est positionné ssi le résultat est nul
  - PC := PC+1

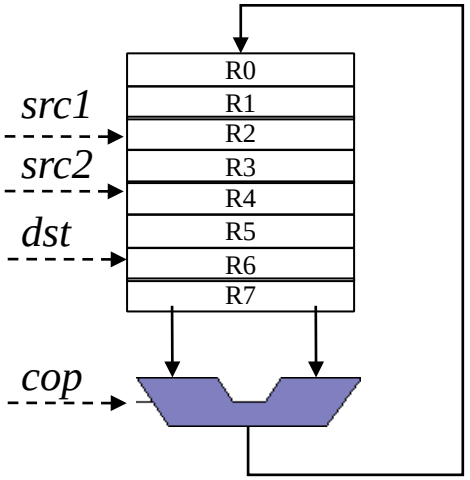
Étude de cas

Architecture à banc de registres MicroReg

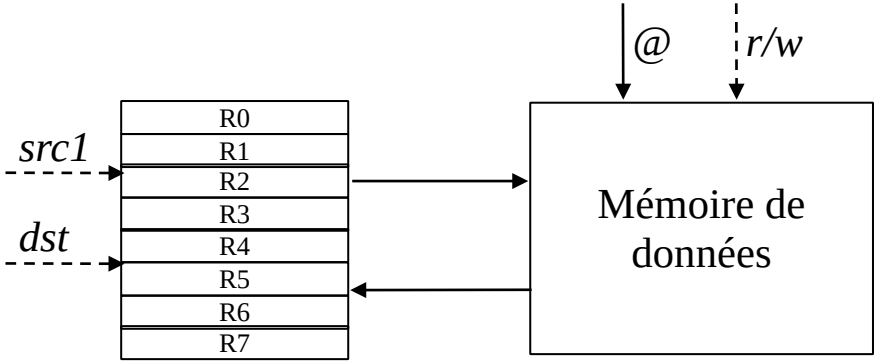
Modèle Harvard

Architecture à registres généraux

- L'accumulateur est remplacé par un banc de registres (e.g. 2 lectures, 1 écriture par cycle)
  - Opérations arithmétiques sur les registres
  - Opérations LOAD/STORE avec la mémoire

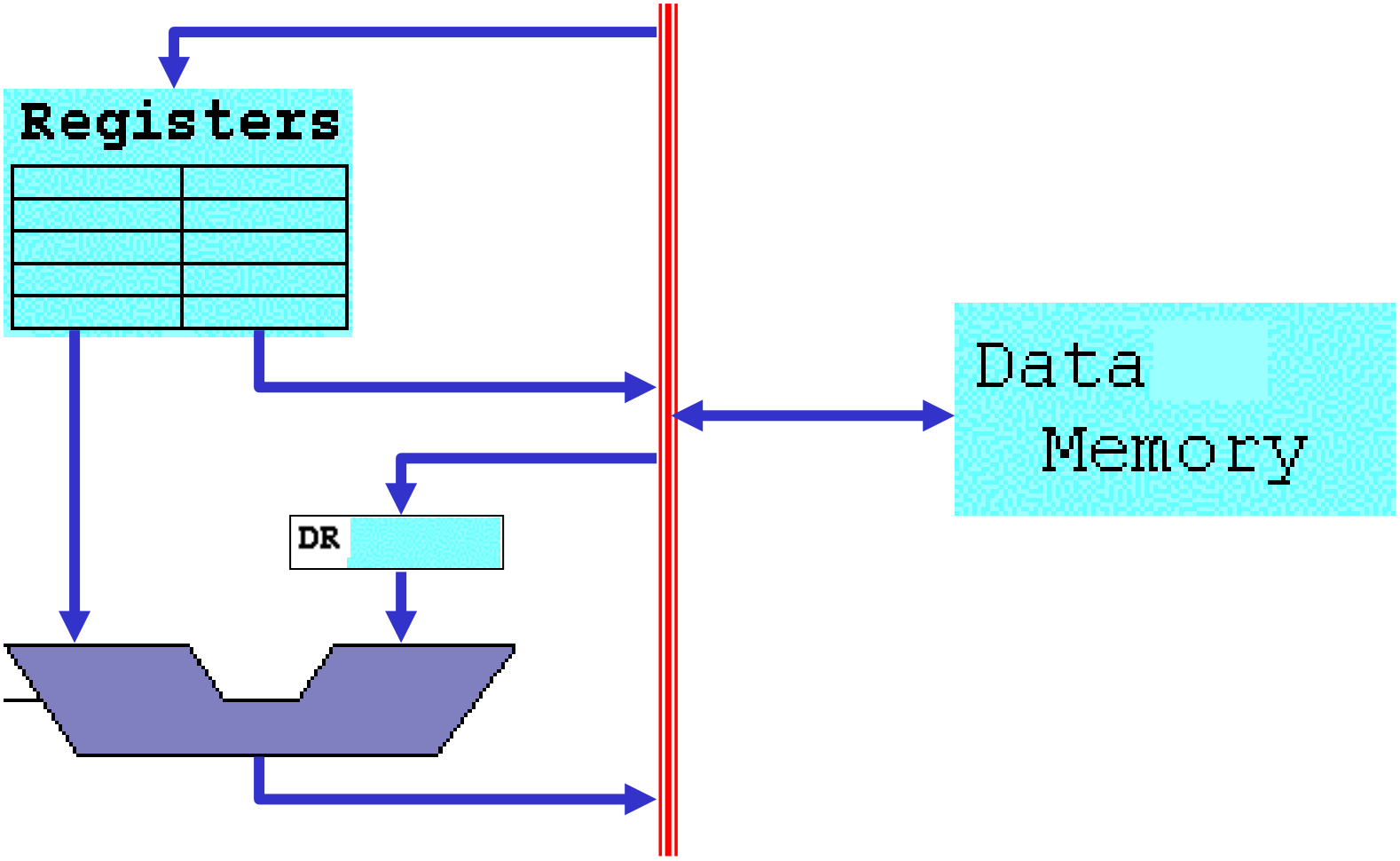


Opérations arithmétiques



Opérations LOAD/STORE

Le bus de données

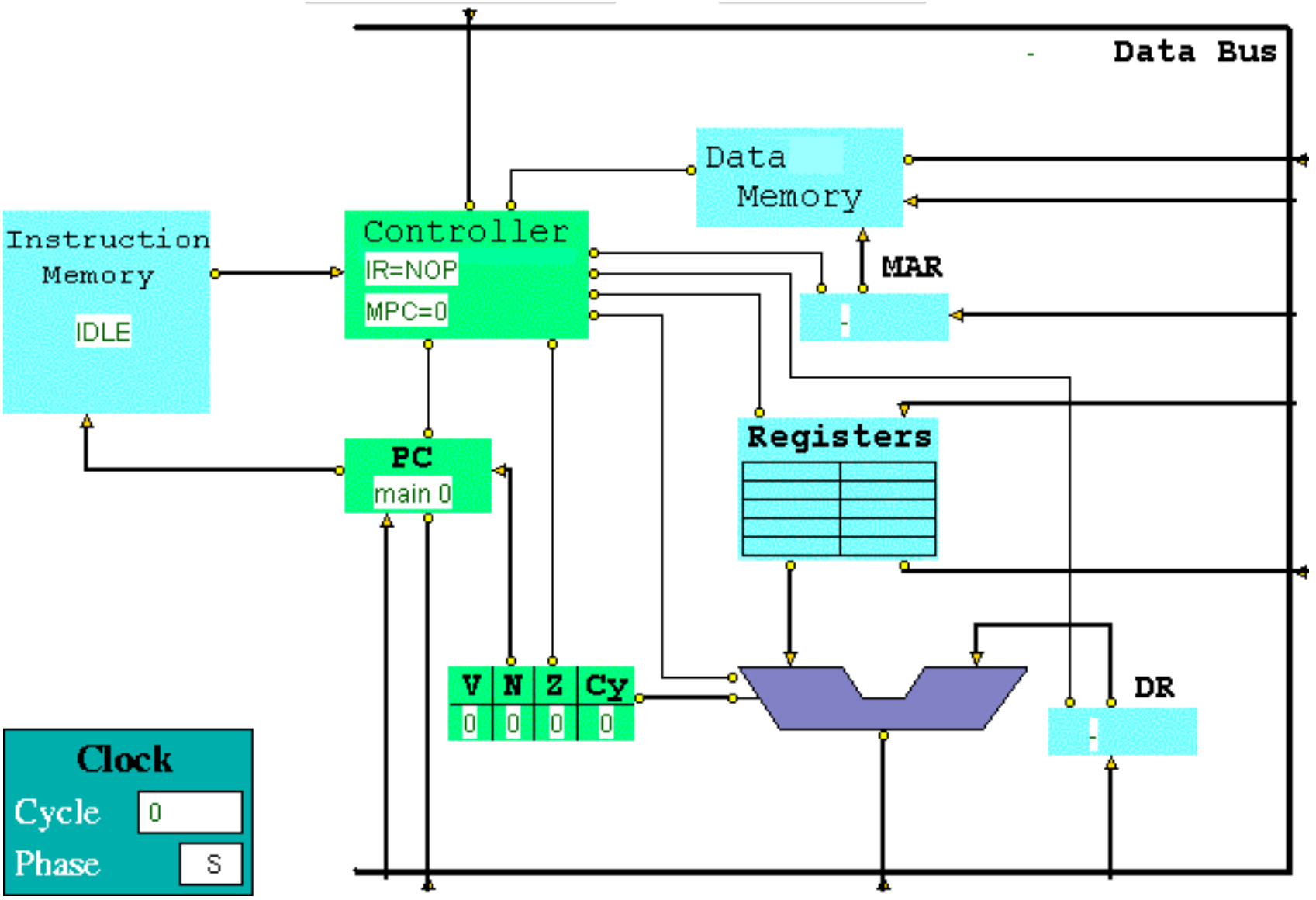


- Un bus de données permet les diffusions vers tous les acteurs

Registres généraux - Avantages

- En 1 cycle, 2 lectures et 1 écriture de données
  - Les bancs de registres sont plus rapides que les mémoires
  - ADDI R1,R2,5 remplace LOADI 5; ADD R2; STORE R1
- Les registres sont interchangeables
  - Facile à compiler, les variables sont associées à des registres
- Contraintes
  - Les compilateurs réduisent le nombre d'accès à la mémoire
  - Il faut une instruction assez grande pour coder tous les opérandes

MicroReg


Data Bus
Data Memory
Instruction Memory
IDLE
Controller
IR=NOP
MPC=0
MAR
Registers
PC
main 0
V N Z Cy
0 0 0 0
DR
Clock
Cycle 0
Phase S

Adressage par registre et indexé

src,dst,src1,src2 sont des registres

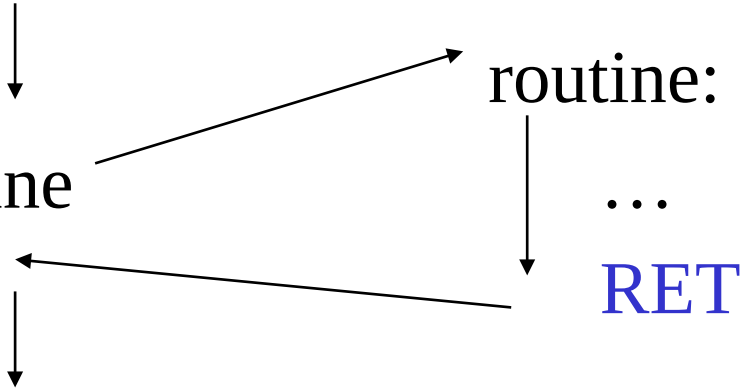
- Adressage par registre
  - **ADD** dst, src1, src2 — *dst = src1 + src2*
  - **MOV** dst, src — *dst = src*
- Adressage immédiat
  - **ADDI** dst, src, valeur — *dst = src + valeur*
  - **MOVI** dst, valeur — *dst = valeur*
- Adressage direct
  - **ADDD** dst, src, adresse — *dst = src + MEM[adresse]*
- Adressage indexé
  - **LOADRI** dst, src1, offset — *dst = MEM[src1+ offset]*

Jeu d'instruction

- Le branchement
  - JMP adr : PC := adr
- Les branchements conditionnels
  - JZ adr : PC := adr ssi Z, PC := PC+1 sinon
  - JNZ adr : PC := PC+1 ssi Z, PC := adr sinon
  - JN adr : PC:= adr ssi N, PC := PC+1 sinon
  - JP adr : PC := adr ssi P (ni Z, ni N), PC := PC+1 sinon
- Z, N et P sont les indicateurs positionnés par les instructions arithmétiques

Appel de sous-routines

- CALL adr
  - Appel la routine située à l'adresse *adr*
- RET
  - Fin de la routine, retour après l'appel
- Exemple
  - …
  - CALL routine
  - …



- Comment sait-on où il faut RETourner ?

Appel de sous-routine

- Le PC est empilé avant le CALL
  - CALL adresse
    - PUSH PC
    - JMP adresse
  - RET
    - POPR PC
- Et les paramètres ?
  - Les arguments sont empilés avant le CALL
  - La routine se trouve dans la pile
    - PC, par1, par2

La pile ?

- Une pile est un ensemble de mots mémoires accédés dans un ordre LIFO (Last In First Out)
- La plupart des processeurs proposent un mécanisme de pile
  - La pile est alors réalisée dans la mémoire de données
  - Un registre sert de pointeur de pile (sommet) : R15
  - R15 contient l'adresse du sommet de pile (SP)
- Les instructions PUSH et POP permettent la manipulation

PUSH, POPR et PUSHR

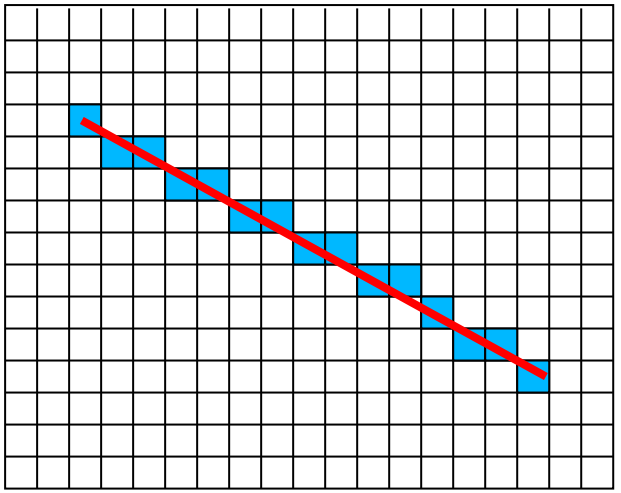
- PUSH valeur *// adressage immédiat*
  - R15 = R15 – 1 *// déplace le sommet de pile*
  - MEM[R15] = valeur *// empile la valeur*
- PUSHR registre *// adressage par registre*
  - R15 = R15 – 1 *// déplace le sommet de pile*
  - MEM[R15] = registre *// empile la valeur du registre*
- POPR registre *// adressage par registre*
  - registre = MEM[R15] *// dépile vers le registre*
  - R15 = R15 + 1 *// déplace le sommet de pile*
- R15 = adresse du sommet de pile

Périphériques (souris, écran, clavier)

- Le **bus de donnée principal** est connecté à des **bus secondaires** dédiés
  - PCI, SCSI, AGP, ISA : plus ou moins rapides
- Certains processeurs utilisent des instructions spéciales pour sortir du bus principal
  - in, out
- D'autres réservent des plages d'adresses à chaque périphériques
  - Exemple: adresse 0xA000000 pour l'écran graphique de Windows, 0xB800000 pour l'écran texte
  - Il suffit alors d'écrire dans cette plage d'adresse pour afficher un point à l'écran

Algorithme de Bresenham

- **Tracer un segment** entre deux points (x1,y1) et (x2,y2)
  - Tracer un segment depuis (x1,y1) de largeur dx=x2-x1 et de hauteur dy=y2-y1
- On ne peut afficher que des points de coordonnées entières
  - L'écran est une matrice de pixels
- L'algorithme de Bresenham affiche seulement les points les plus proches de la droite.

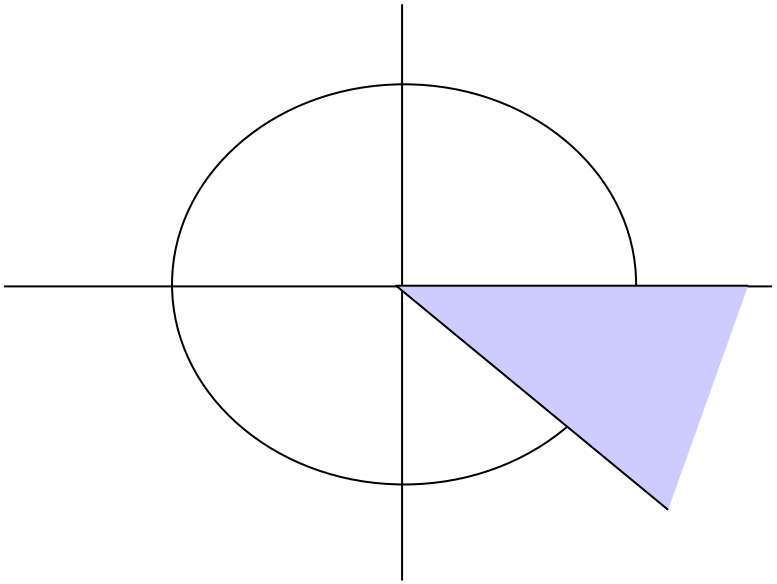


Contraintes de l'algorithmes

- On veut aller vite
  - Il ne faut pas utiliser de nombres approchés
  - Il ne faut pas utiliser de multiplications ni de divisions
    - Le calcul d'une pente = division et nombre approchés
- L'algorithme de Bresenham
  - Utilise seulement des additions, soustractions, décalage de bits (multiplication par 2)
  - Travaille avec des nombres entiers
  - Calcule le retard ou l'avance par rapport à la droite réelle à tracer et maintient le retard ou l'avance inférieure à ½ point

Idée de l'algorithme

- On ne traite que les segments dont la pente est inférieure à 45° $0 \leq \frac{dy}{dx} < 1$
- Dans les autres cas, on fait une symétrie



- Seulement 2 cas possibles:
  - Déplacement horizontal : pente < ½

  - Déplacement diagonal : pente ≥ ½

Idée de l'algorithme

- Déplacement horizontal (x'=x+1, y'=y)
  - La pente est inférieure à 1/2 : `dy/dx < ½`
  - `2*dy-dx < 0` (multiplication par 2 = décalage de bits)
- Déplacement diagonal (x'=x+1, y'=y+1)
  - La pente est supérieure à 1/2 : `2*dy - dx ≥ 0`
- `Var = 2*dy-dx`
  - Var caractérise le retard par rapport à la droite cherchée
  - Calcul du nouveau retard une fois le déplacement effectué
    - Algorithme incrémental

La pente était inférieure à 1/2

- Déplacement horizontal : `2*dy-dx < 0`
  - Quel serait le retard si on continuait horizontalement ?
  - `dy/dx + dy/dx = 2*dy/dx`
- Ce retard serait-il inférieur ou supérieur à ½ ?
  - `2*dy/dx < ½` équivaut à `4*dy-dx < 0`
  - Sinon, on fait un déplacement en diagonale
- Le calcul de var est incrémental
  - `var = var + 2*dy`

La pente était supérieure à 1/2

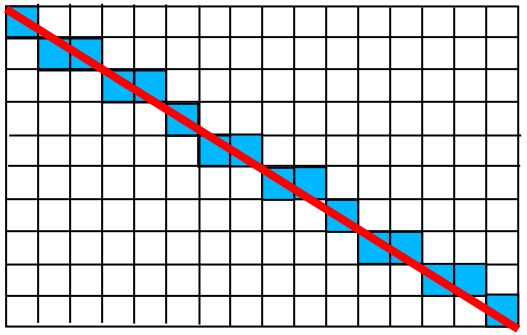
- Déplacement en diagonal : `2*dy-dx < 0`
  - Quel serait le retard si on continuait horizontalement ?
  - `(dy/dx-1) + dy/dx`
- Ce retard serait-il inférieur ou supérieur à ½ ?
  - `2*dy/dx – 1 < ½` équivaut à `4*dy-3*dx < 0`
  - Sinon, on fait un déplacement horizontal
- Le calcul de var est incrémental
  - `var = var + 2*dy – 2*dx`

Formulation

```
x = x1, y = y1
var = 2*dy – dx  ; dy = y2-y1 et dx = x2-x1
Répète
   Dessine(x, y)
   x = x + 1
   Si var < 0 alors
      var = var + 2*dy      ; parcours horizontal
   Sinon
      y = y + 1             ; parcours diagonal
      var = var + 2*dy – 2*dx
Tant que x!=x2
```

Mise en œuvre : dx = 15, dy = 9

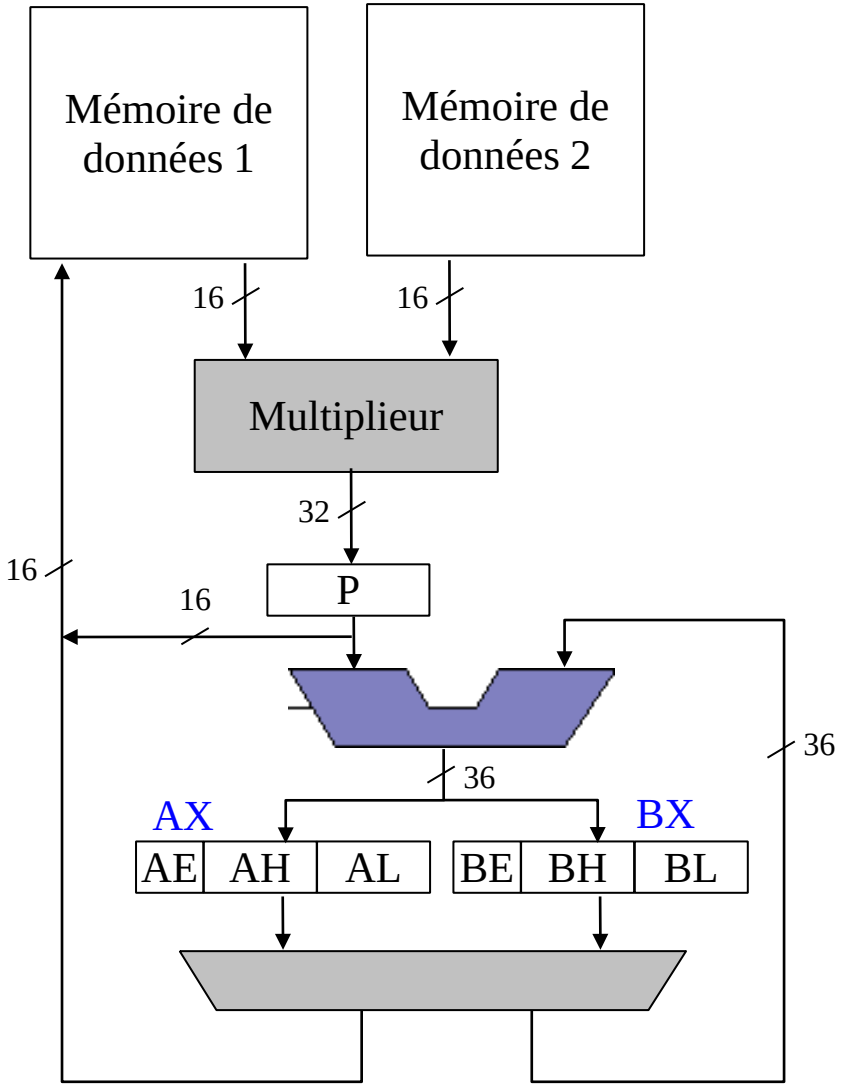
- var = 2*9 – 15 = 3 (retard>1/2) → diagonale
- var = 3 + 18 – 30 = -9 (retard<1/2) → horizontale
- var = -9 + 18 = 9 → diagonale
- var = 9 + 18 – 30 = -3 → horizontale
- var = -3 + 18 = 15 → diagonale
- var = 15 + 18 – 30 = 3 → diagonale
- Idem précédemment : -9, 9, -3, 15



Architecture de type
Digital Signal Processor

- Les chemins de données deviennent complexes pour améliorer les performances
  - Architecture mémoire de type Harvard modifiée
    - 2 mémoires de données et 1 mémoire d'instructions
  - Des registres spécialisés dispersés dans l'architecture
- Les chemins de données se calquent sur les problèmes de traitement du signal visés
  - Multiplication/Accumulation (MAC)

DSP : exemple



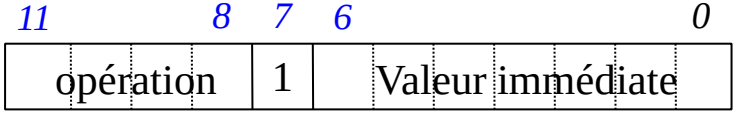
- 1 registre P(roduit) : 32 bits
  - 2 sous registres PH, PL
- 2 accumulateurs 36 bits
  - AX,BX,AH,AL,AE,BH,BL,BE
- Boucle matérielle : registre LP
- Très efficace pour les produits scalaires : $\vec{u} \cdot \vec{v} - \sum u_i \times v_i$
- Harvard modifiée
  - u_i : mémoire 1, b_i : mémoire 2
  - Résultat : accumulateur

DSP : limitations

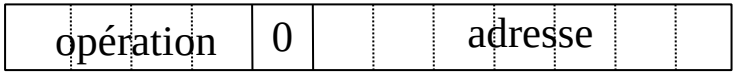
- Très inefficaces pour les applications non visées
  - Le DSP présenté ne sait faire que des produits scalaires
- Très difficile d'écrire un compilateur
  - Les variables doivent résider en mémoire (dans la bonne mémoire de données)
  - Les boucles matérielles ne savent faire que des itérations (difficiles à détecter dans du C mal écrit)
- La durée de vie d'un DSP est inférieure à 6 mois
  - Environnement de programmation difficile à rentabiliser
- => La structure des DSP se simplifie
  - Le silicium ne coûte plus aussi cher

Codage des instructions

- La complexité du chemin de données se répercute sur le codage (et décodage) des instructions
- Architectures à accumulateur
  - toutes les instructions ont un opérande : n bits
  - 2 modes d'adressage (immédiat / direct) : 1 bit de plus
  - e.g. Mémoire d'instructions : 12 bits
    - OpI a



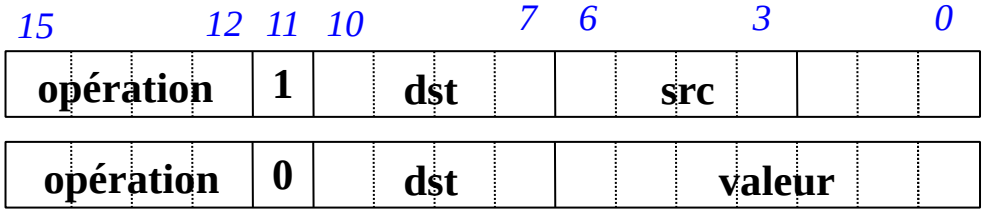
    - Op a



    - 16 instructions possibles, bit 7 : adressage
    - On ne peut manipuler que des données immédiates de 7 bits
    - On ne peut adresser que 2^7 octets de la mémoire (128 valeurs)
    - Sinon, il faut plusieurs mots pour coder les instructions longues

Architectures à registres généraux

- La première source = destination : 2 opérandes
  - Add dst, src (*dst = dst+src*)
  - AddI dst, val (*dst = dst+val*)



  - Le codage d'un registre est plus compact que celui d'une valeur imm.
- 2 sources et 1 destination distincte : 3 opérandes
  - Il faut limiter les combinaisons (modes d'adressage) possibles
    - RISC : Reduced Instruction Set Computer (e.g. ARM, PowerPC)
  - OU Élargir la taille des instructions
    - VLIW : Very Long Instruction Word (e.g. TI C6x : 128 bits)
  - OU Avoir des instructions de taille variable
    - CISC : Complex Instruction Set Computer (e.g. Intel 80X86, Motorola 68x00)

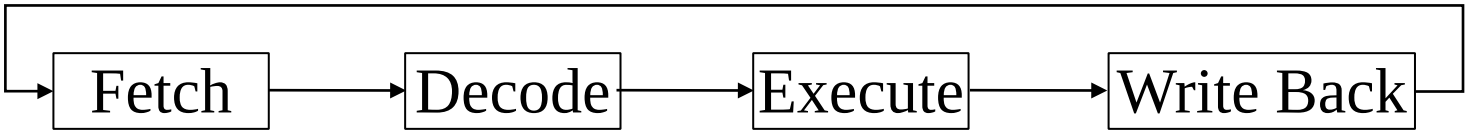
Modes d'adressage plus complexes

- Adressage indirect
  - Add dst, (adresse) *dst = dst + Mem[Mem[adresse]]*
- Adressage indirect par registre
  - Add dst, (src) *dst = dst + Mem[src]*
- Auto-incrémentation, Auto-décrémentation
  - Add dst, (src)+ *dst = dst + Mem[src]*
    src = src + 1
- Auto-incrémentation modulo
  - Add dst,(src)+S *dst = dst + Mem[src]*
    src = (src + 1) modulo RS

Familles d'architectures (1/3)

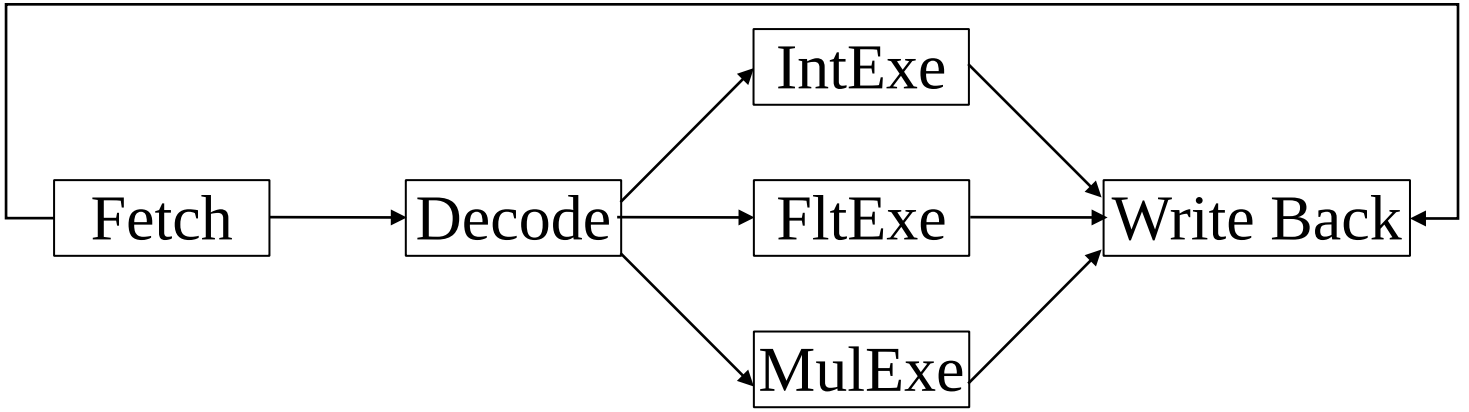
- RISC – Reduced Instruction Set Computer
  - Toutes les instructions ont la même taille
  - Que des modes d'adressage simples (immédiat, registre)
- CISC – Complex Instruction Set Architecture
  - Instructions de taille et de temps d'exécution variable
  - Modes d'adressage complexes
  - Il faut plusieurs instructions RISC pour 1 instruction CISC
- DSP – Digital Signal Processor
  - Mixte CISC/RISC : optimisés pour une application donnée

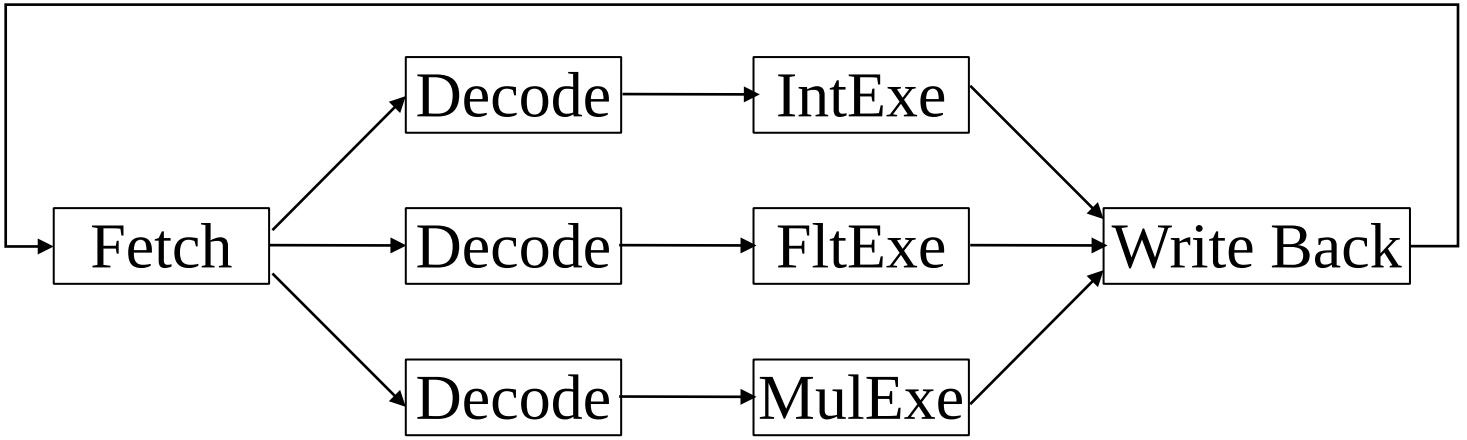
CISC/RISC



Super scalaires

- Plusieurs unités fonctionnelles (2 UAL, 2 multiplieurs, etc.)
- 1 seule instruction est décodée à la fois





- VLIW – Very Long Instruction Word
  - Plusieurs unités fonctionnelles
  - 1 instruction est composée de plusieurs mini-instructions RISC indépendantes qui sont toutes exécutées en même temps
- EPIC – Explicit Parallel Instruction Computing
  - VLIW + mais il peut y avoir des dépendances entre les mini-instructions