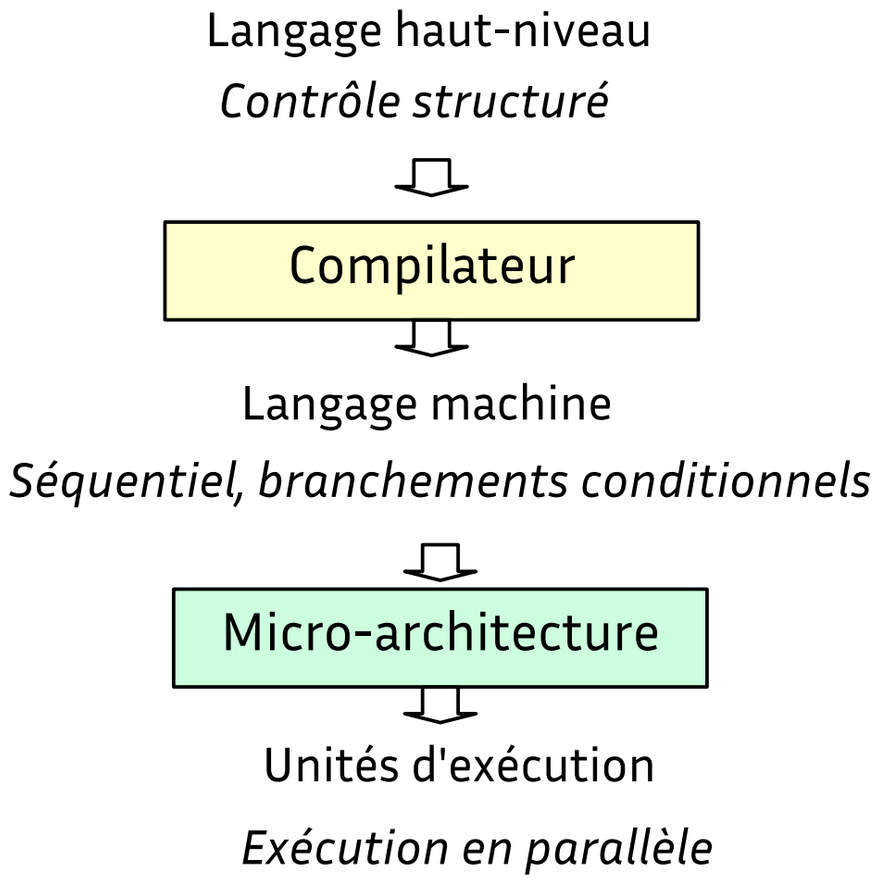
Comment concilier contrôle et parallélisme ? Approches des architectures de processeurs généralistes et graphiques

8 février 2024
Collège de France

Caroline Collange
caroline.collange@inria.fr

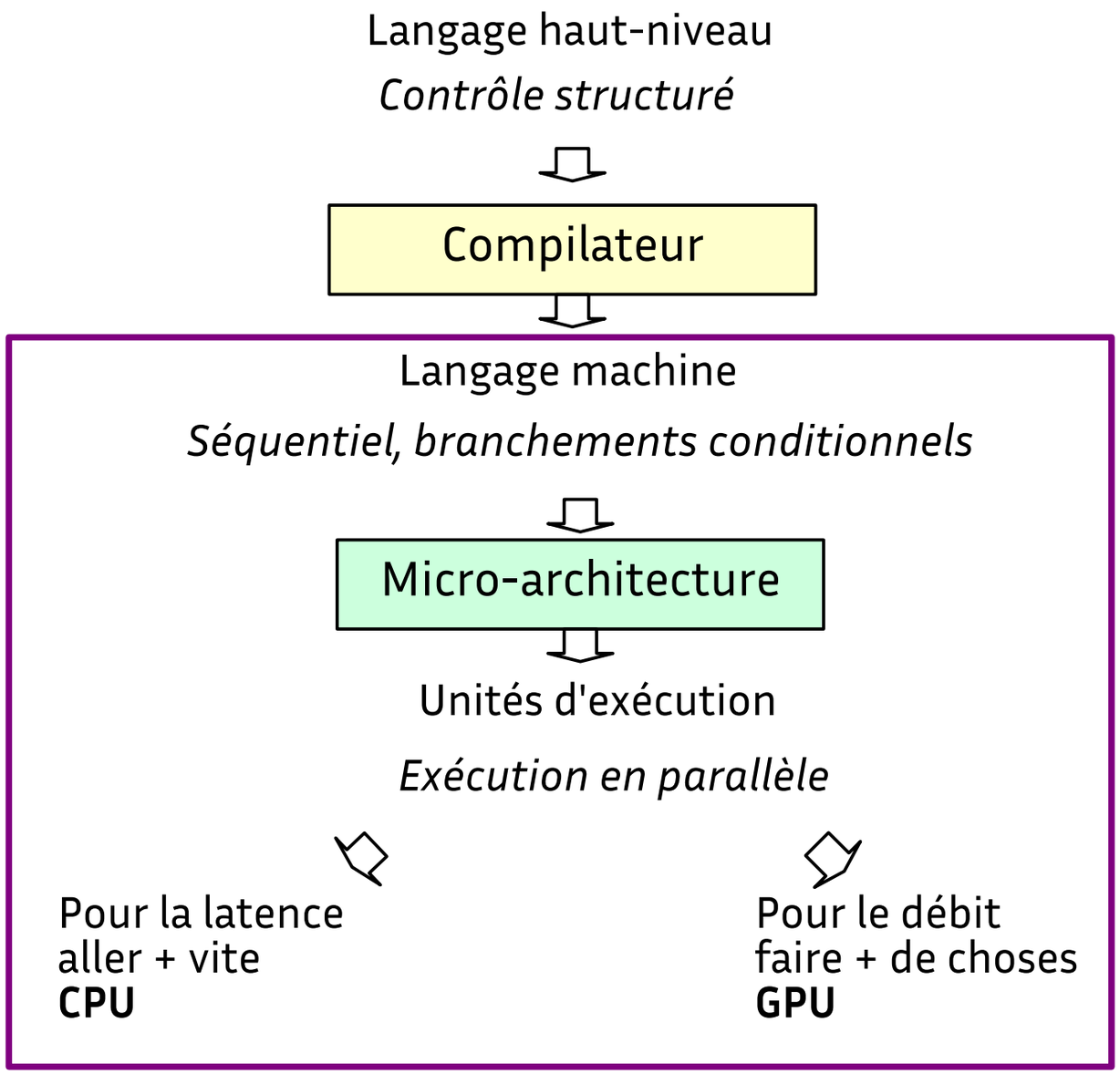
Gérer le flot de contrôle en matériel

- Cours : le contrôle dans les langages de programmation
- Ce séminaire : le contrôle dans les architectures de processeurs
  - Comment mettre en œuvre efficacement les structures de contrôle en matériel ?



Gérer le flot de contrôle en matériel

- Cours : le contrôle dans les langages de programmation
- Ce séminaire : le contrôle dans les architectures de processeurs
  - Comment mettre en œuvre efficacement les structures de contrôle en matériel ?



Plan

- Point de départ : la Machine Analytique de Babbage et Lovelace
- Processeurs généralistes : aller plus vite
  - Prédiction de branchements
  - Renommage de registres
- Processeurs graphiques : faire plus de choses
  - Retour au 19^{e} siècle
  - Exécution vectorielle masquée
- Recherche actuelle : peut-on concilier les deux ?
  - Points bloquants : exécution masquée et renommage
  - PIRAT à la rescousse

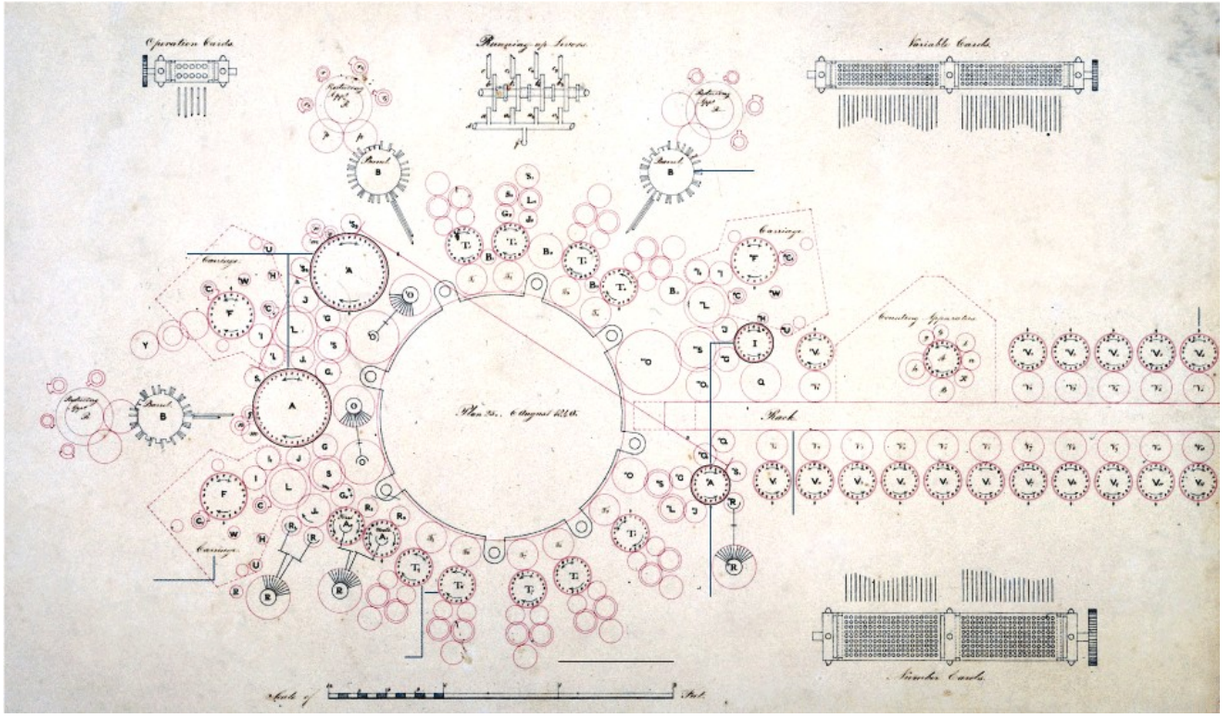
Point de départ : la Machine Analytique

19e siècle

- Charles Babbage conçoit un **ordinateur mécanique universel** : la Machine Analytique
- Ada Lovelace écrit **des programmes** pour la machine analytique

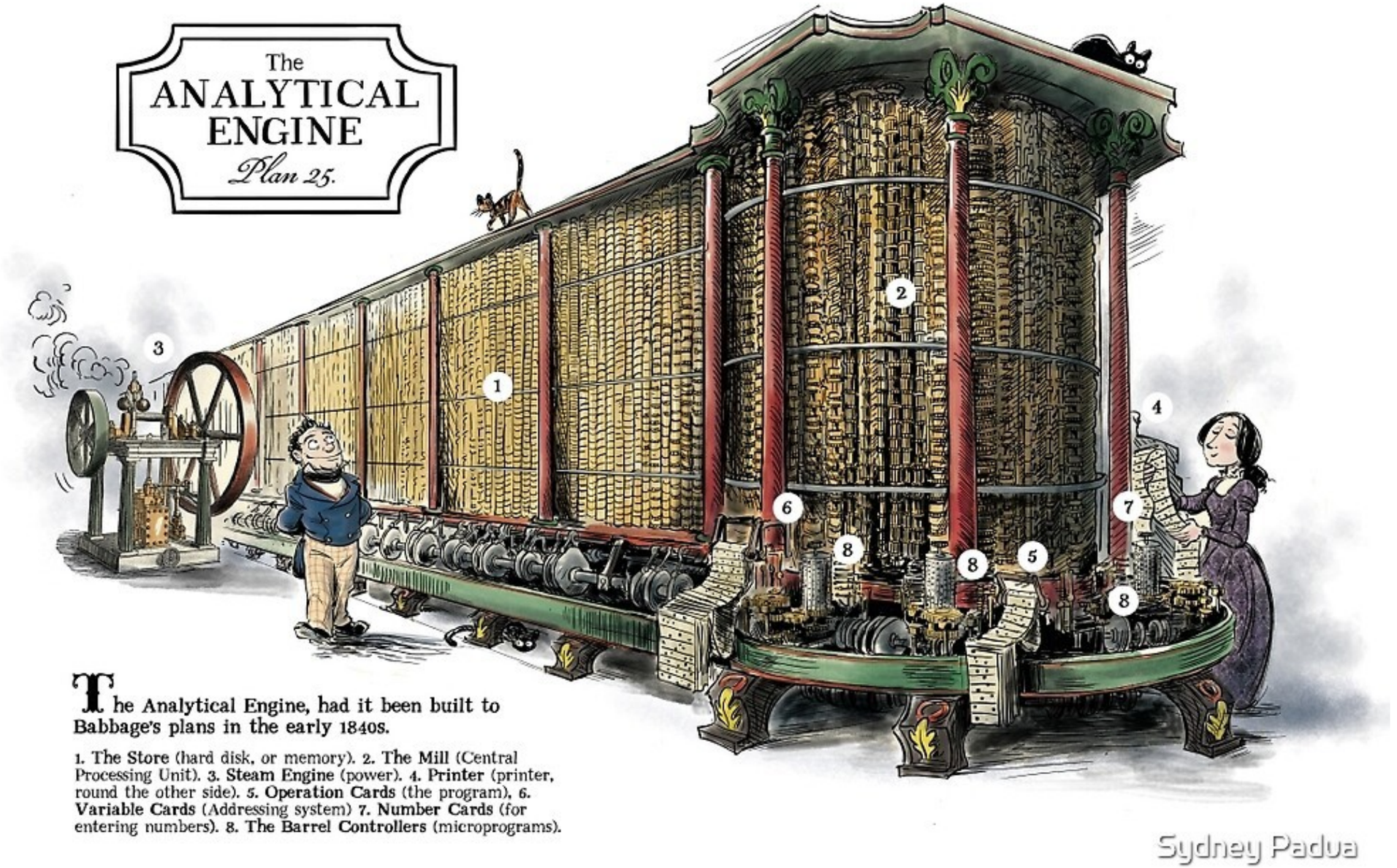
Aujourd'hui

- Principes toujours enseignés en cours d'architecture des ordinateurs
- La Machine Analytique attend toujours d'être construite



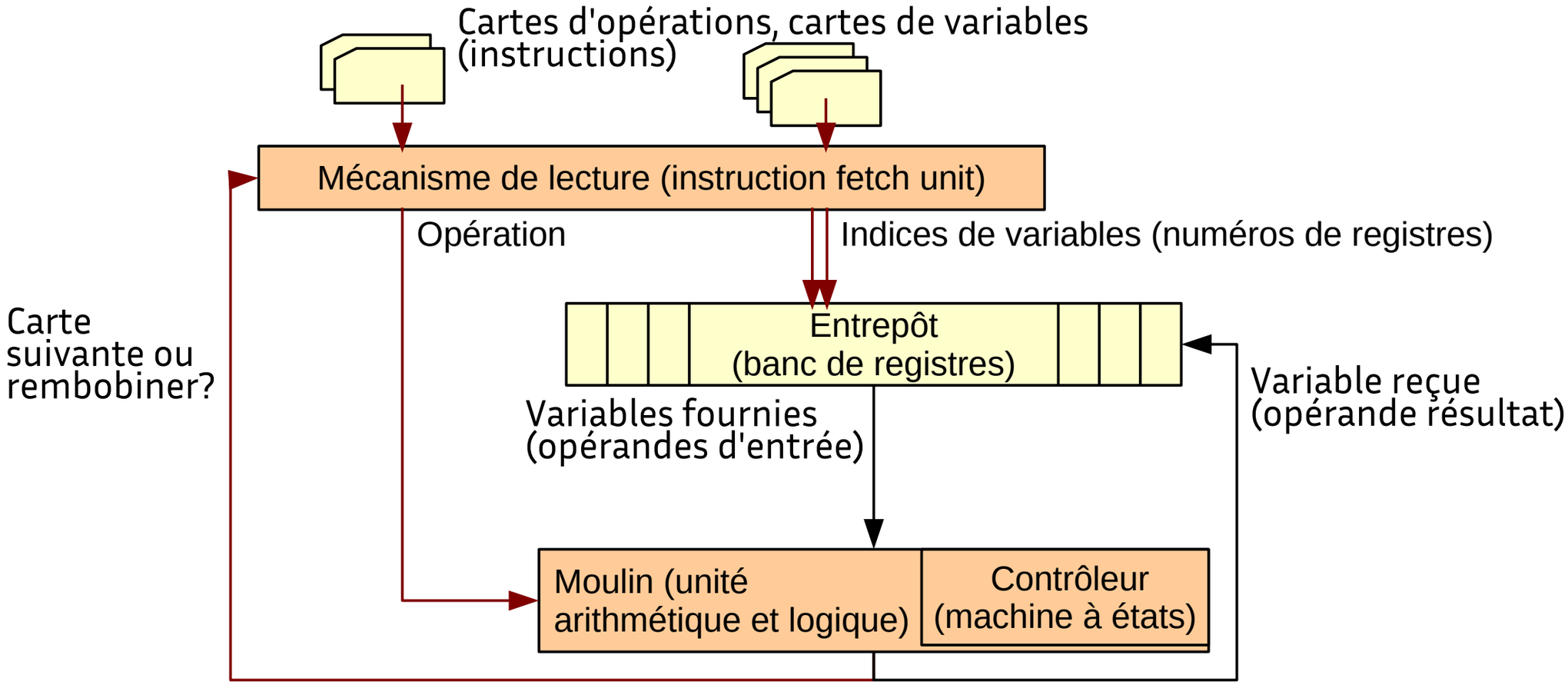
Charles Babbage, *plan de la machine analytique*, 1838

Analytical Engine internals



De: Sydney Padua. *The Thrilling Adventures of Lovelace and Babbage*. http://sydneypadua.com/2dgoggles/

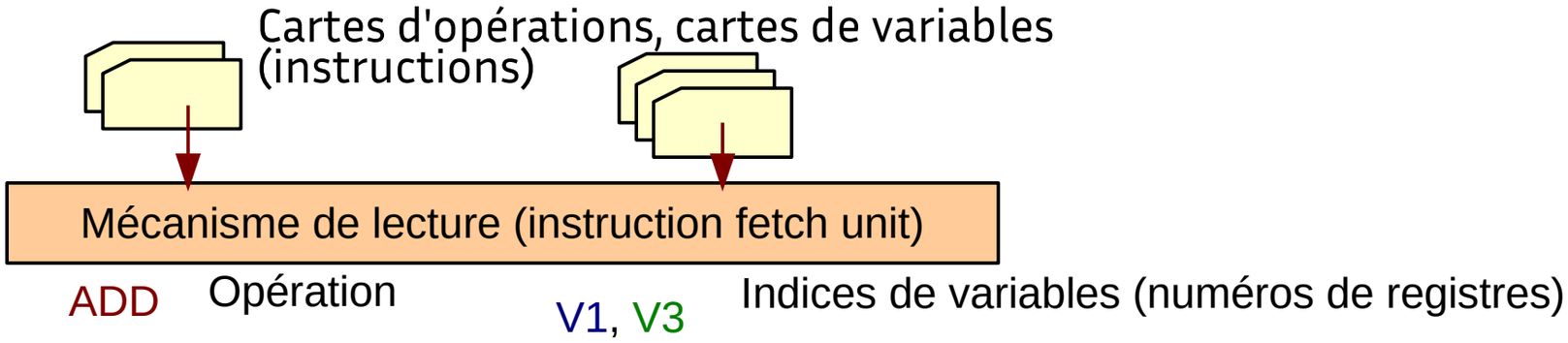
Fonctionnement de la Machine Analytique



- Déroulons l'exécution d'une instruction

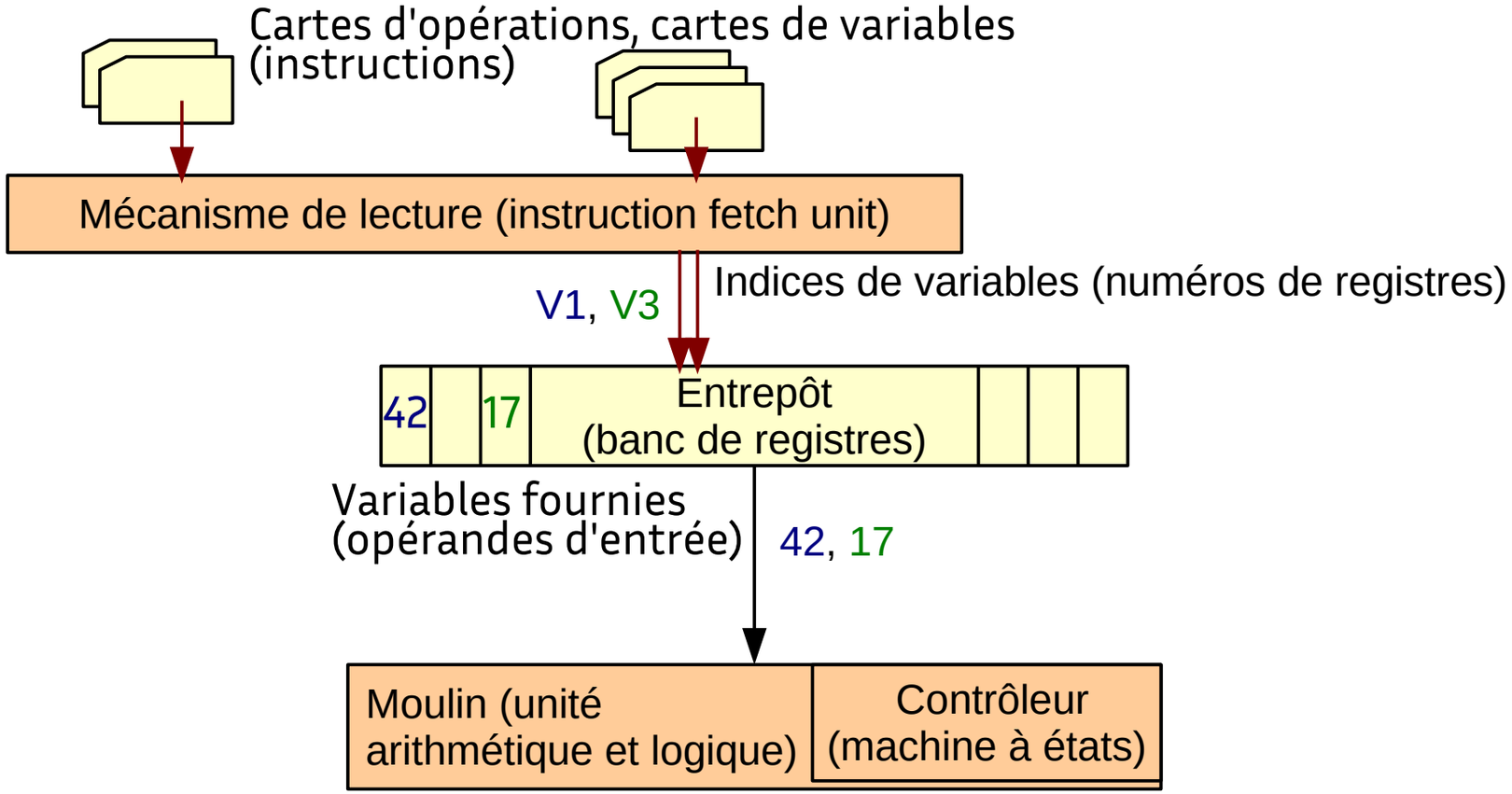
Pas à pas: Acquisition

- Le mécanisme de lecture récupère les cartes d'opération et de variables, lit leur contenu



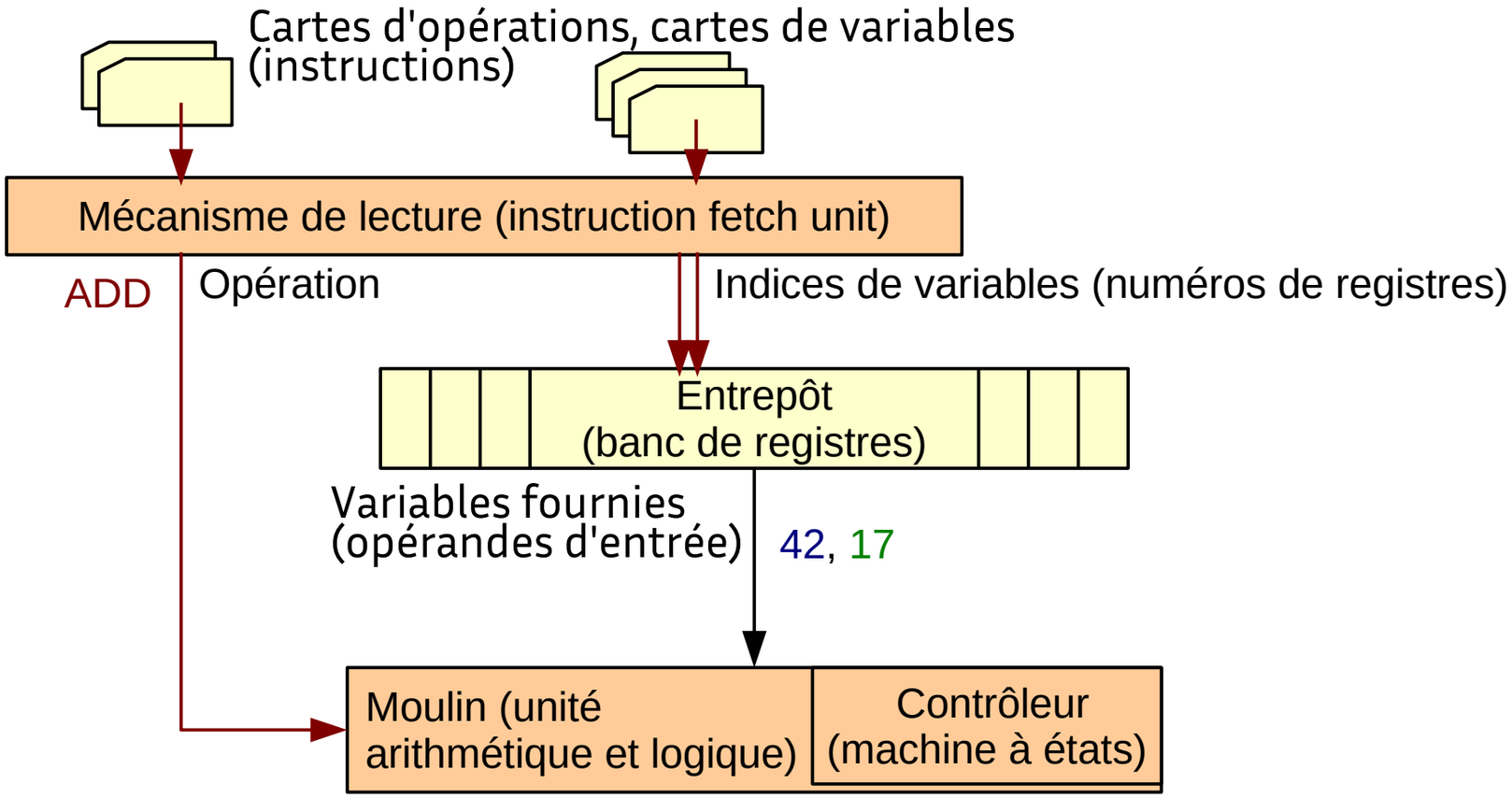
Lecture des opérandes

- L'Entrepôt lit les variables spécifiées par les cartes de variables, et les envoie au Moulin



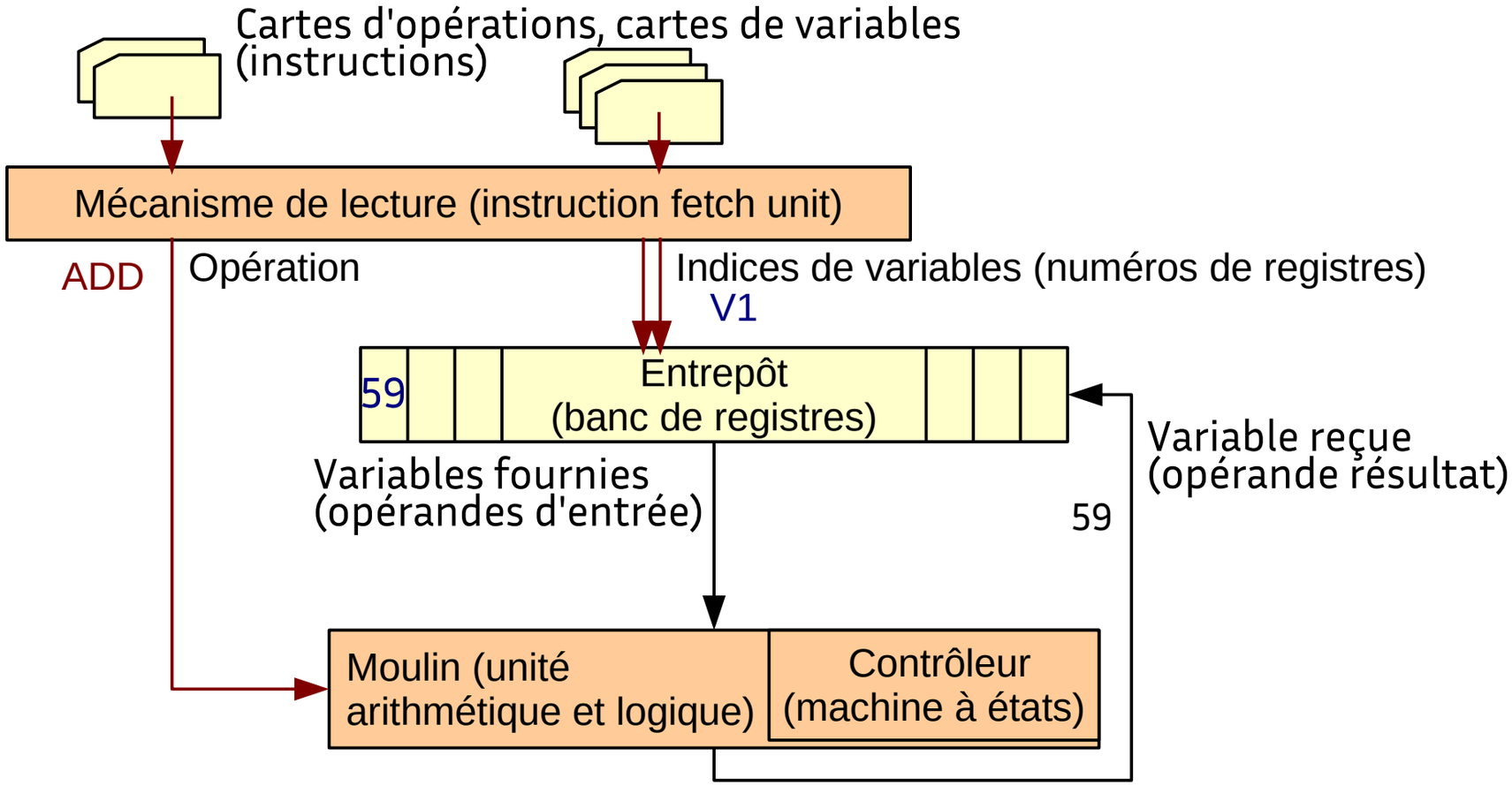
Exécution

- Le Moulin calcule l'opération spécifiée par la carte opération



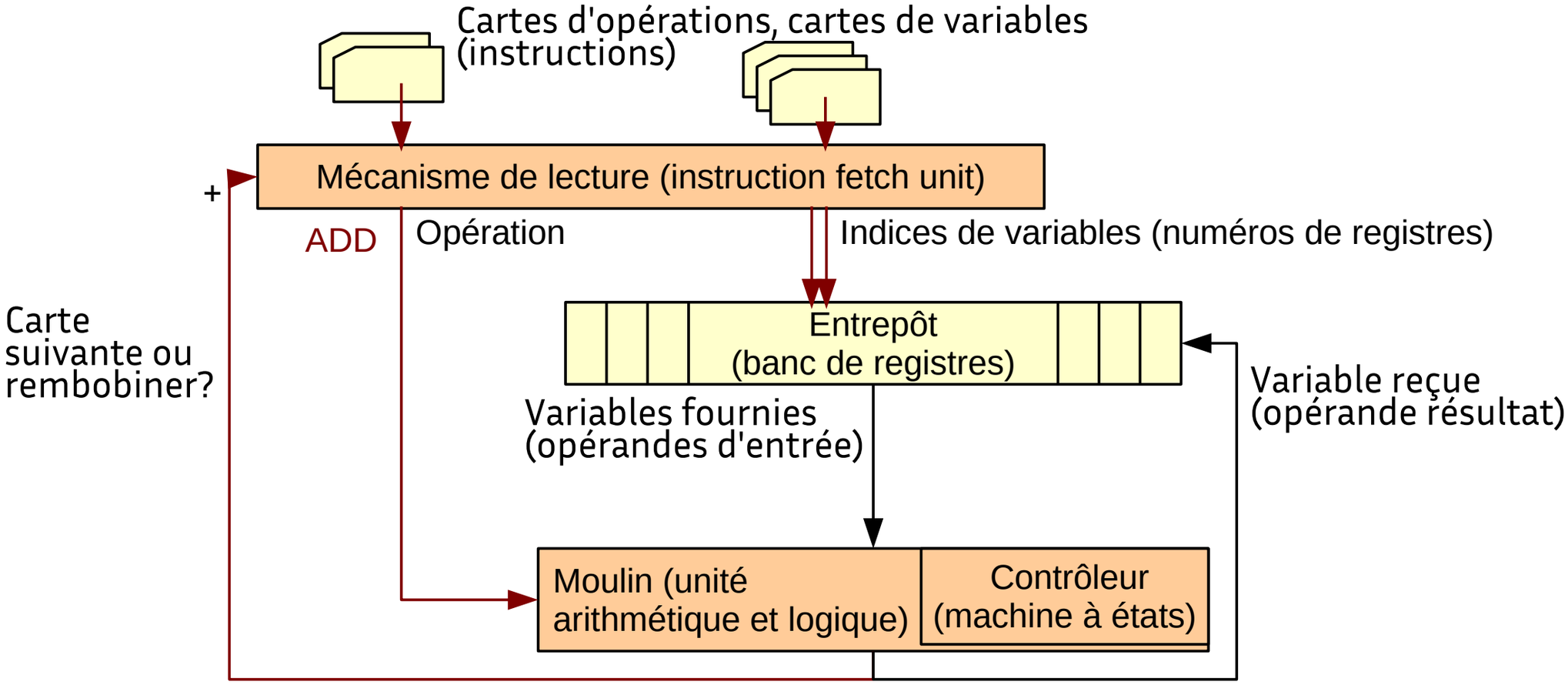
Écriture du résultat

- Le Moulin envoie son résultat au Moulin
- Le Moulin écrit la valeur dans la variable spécifiée par la carte de variable résultat



Mise à jour du compteur de programme

- Le Moulin envoie le signe du résultat au Mécanisme de lecture
- Le Mécanisme de lecture sélectionne la prochaine carte à lire suivant l'opération courante et le signe du résultat



Plan

- Point de départ : la Machine Analytique de Babbage et Lovelace
- Processeurs généralistes : aller plus vite
  - Prédiction de branchements
  - Renommage de registres
- Processeurs graphiques : faire plus de choses
  - Retour au 19e siècle
  - Exécution vectorielle masquée
- Recherche actuelle : peut-on concilier les deux ?
  - Points bloquants : exécution masquée et renommage
  - PIRAT à la rescousse

Aller plus vite : plusieurs instructions à la fois

- Exemple d'Ada Lovelace : calculer $\frac{a\,x^n}{bpy}$

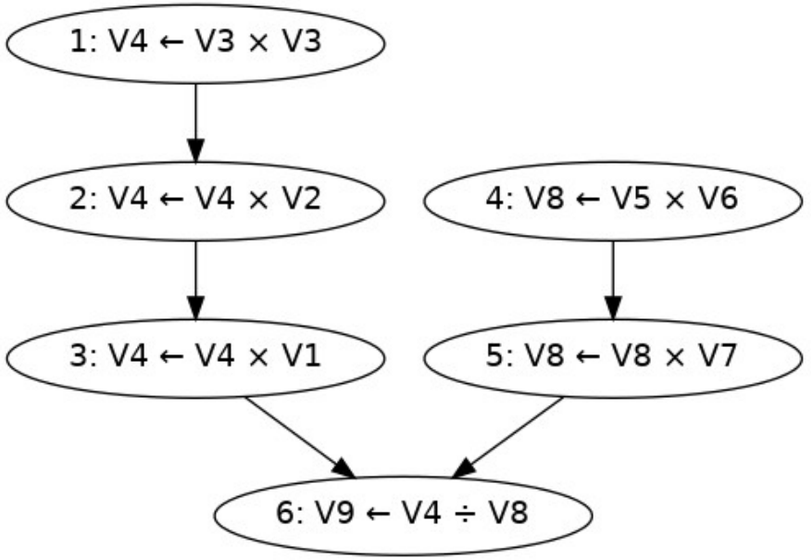
Variables :

V1	V2	V3	V4	V5	V6	V7	V8	V9
a	n	x	ax^n	b	p	y	bpy	$\frac{ax^n}{bpy}$

Programme pour n=3 :

1: V4 ← V3 × V3
2: V4 ← V4 × V3
3: V4 ← V4 × V1
4: V8 ← V5 × V6
5: V8 ← V8 × V7
6: V9 ← V4 ÷ V8

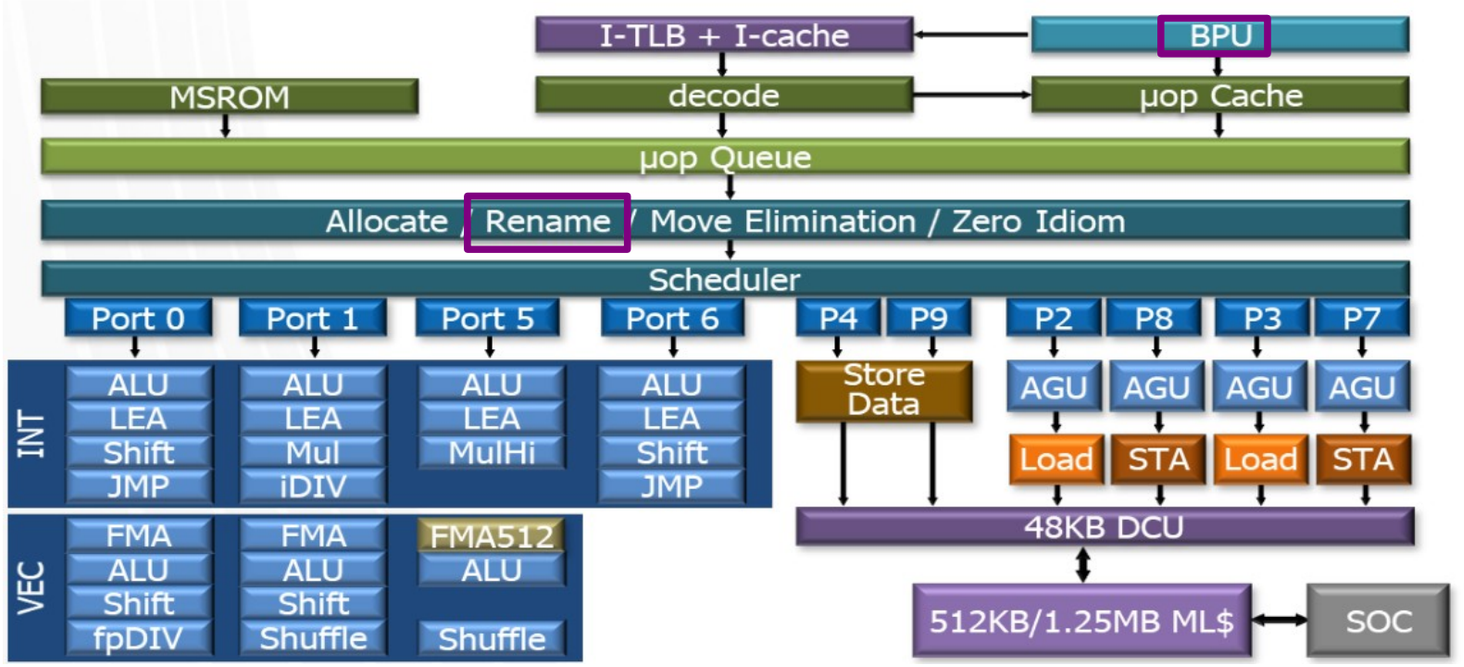
Graphe de flux de données :



- Avec 2 multiplieurs : faisable en 4 cycles d'exécution
- Nécessite de **s'affranchir de l'ordre** du programme
  - ex : exécuter instruction 4 avant 3
- Maintenir une **fenêtre d'instructions** exécutables dans le désordre

La Machine Analytique du 21e siècle

- Example: cœur de processeur Intel i7 Gen 10



	Ice Lake (per core)
Out-of-order Window	352
In-flight Loads + Stores	128 + 72
Scheduler Entries	160
Register Files – Integer + FP	280 +224
Allocation Queue	70/thread; 140/1 thread
L1D Cache (KB)	48
L1D BW (B/Cyc) – Load + Store	128 + 64
L2 Unified TLB	2K
Mid-level Cache (MB)	1.25

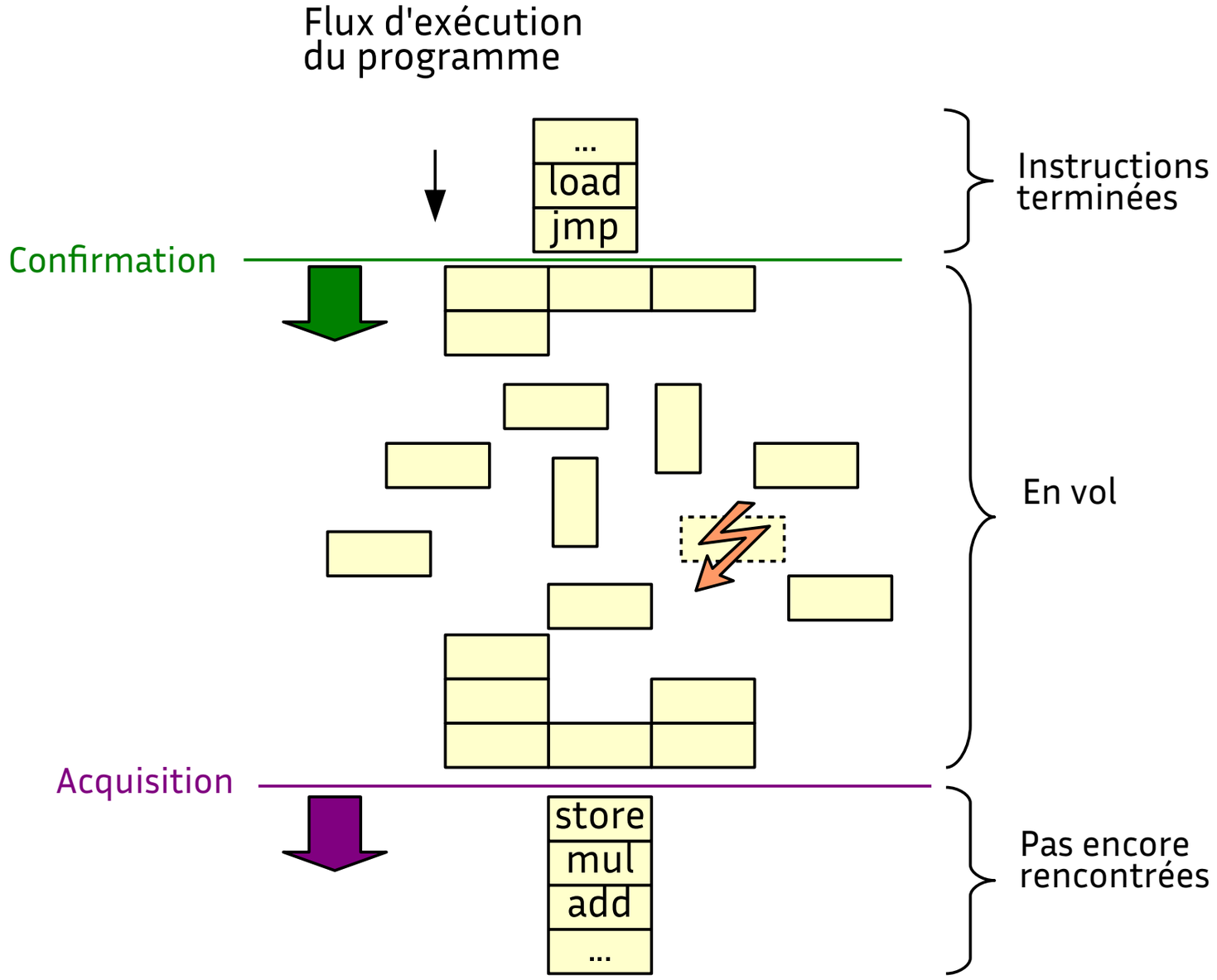
From: Irma Esmer Papazian, *New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP)*, Hot Chips 2020

- Même modèle de programmation : séquence d'instructions, dont saut conditionnel
- Autrefois : 1 carte à la fois → Maintenant : centaines d'instructions en vol

Exécution spéculative dans le désordre

Les instructions sont

- acquises depuis la mémoire (*fetch*) dans l'ordre
- renommées dans l'ordre
- exécutées dans le désordre suivant les flux de données
- certaines sont annulées (mauvaise prédiction)
- les instructions correctes sont remises dans l'ordre
- confirmation (*commit*) : rend visible les effets de bord, libère les ressources



Obstacles et contournements

- Dépendances de contrôle
  - ~1 instruction sur 5 est un branchement conditionnel
    → ~70 branchements en vol
  - La direction de chaque branchement dépend de valeurs pas encore calculées
  - Prédire les directions et cibles des branchements

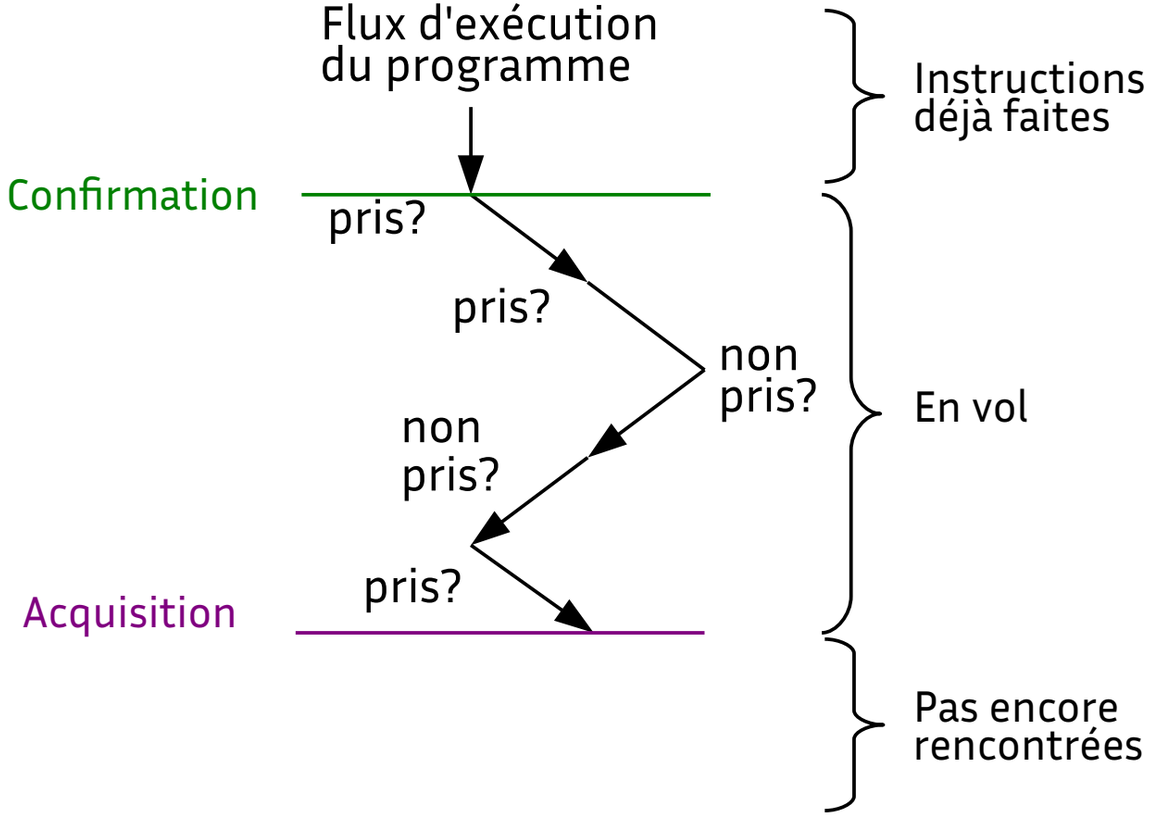
- Dépendances de données
  - Dépendance écriture-après-lecture :
    Avant d'écrire dans un registre, besoin de s'assurer que les instructions précédentes aient lu l'ancienne valeur

```
add r3 ← r1+r2
mul r1 ← r2×r2
```

  - Renommer les registres

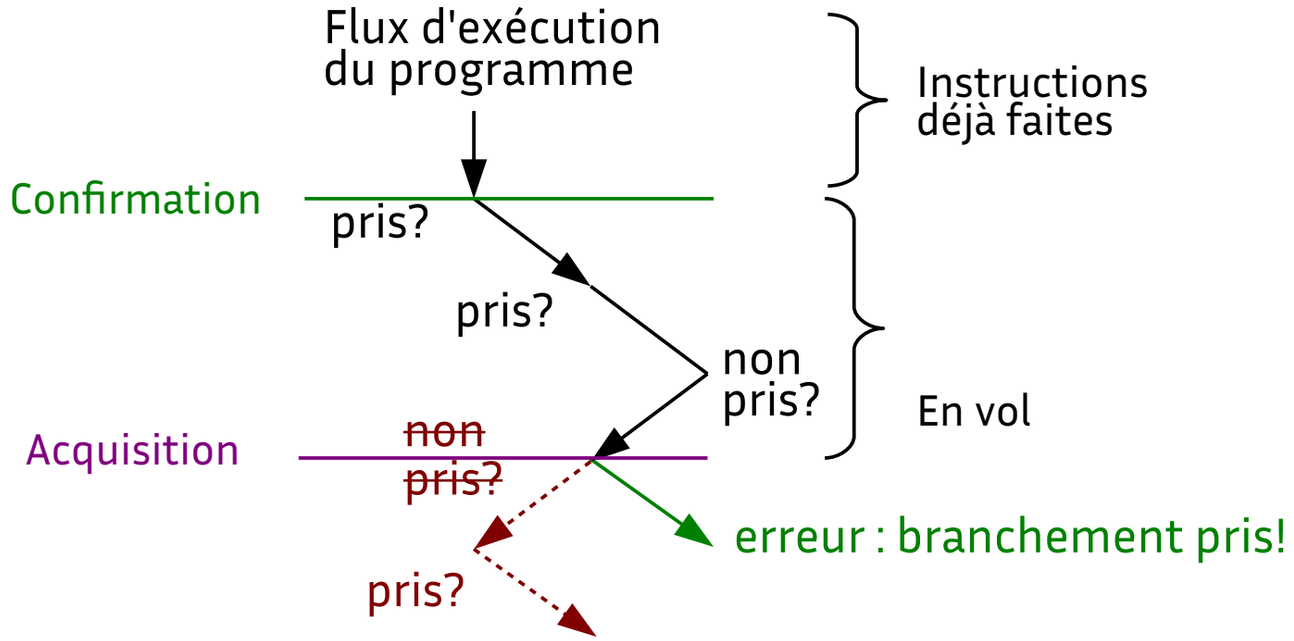
Prédiction de branchements : exemple

- Pour chaque branchement, prédire la direction
  - ex. pris ou non-pris
  - Le prédicteur apprend à la volée le comportement statistique et les corrélations entre branchements : la plupart des branchements suivent des motifs réguliers, peu sont aléatoires
  - Les prédicteurs de l'état de l'art sont très précis : ~1 à 2 mauvaises prédictions / 1000 instructions



Mauvaise prédiction

- Lorsqu'une prédiction s'avère fausse
  - Jeter les instructions du mauvais chemin
  - Reprendre l'acquisition des instruction depuis l'autre chemin



Renommage de registres

- Assigner un nouveau nom à chaque variable écrite
  - Notation d'Ada Lovelace: $^{2}\mathbf{V}_1$ est la 2e assignation de la variable 1
  - Permet d'exprimer un programme sous forme d'équations

$$\begin{aligned} V_3 &\leftarrow \mathbf{V_1} + V_2 \\ \mathbf{V_1} &\leftarrow V_2 \times V_2 \end{aligned} \quad \Rightarrow \quad \begin{aligned} {}^{3}V_3 &= {}^{1}\mathbf{V_1} + {}^{2}V_2 \\ {}^{2}\mathbf{V_1} &= {}^{2}V_2 \times {}^{2}V_2 \end{aligned}$$

impératif : assignations — fonctionnel : équations

  - Élimine les dépendances écriture-après-lecture
    → davantage d'instructions indépendantes

- Renommage de registres
  - Renomme les registres du programme en registres physiques à la volée, en matériel
  - Recycle les registres physiques quand ils ne sont plus nécessaires

L. F. Menabrea and A. A. Lovelace. *Sketch of the Analytical Engine invented by Charles Babbage*. 1842

Plan

- Point de départ : la Machine Analytique de Babbage et Lovelace
- Processeurs généralistes : aller plus vite
  - Prédiction de branchements
  - Renommage de registres
- Processeurs graphiques : faire plus de choses
  - Retour au 19^{e} siècle
  - Exécution vectorielle masquée
- Recherche actuelle : peut-on concilier les deux ?
  - Points bloquants : exécution masquée et renommage
  - PIRAT à la rescousse

Le métier Jacquart, un GPU avant l'heure ?

Programme
sur cartes perforées

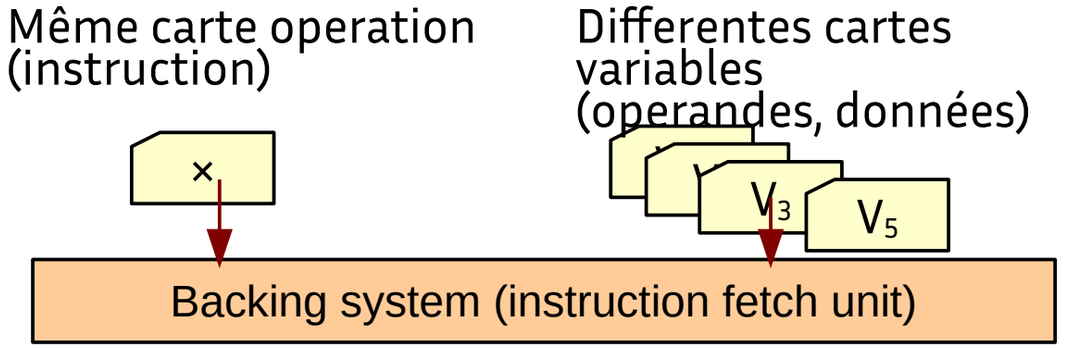
Bistanclaque

Image sur tissu

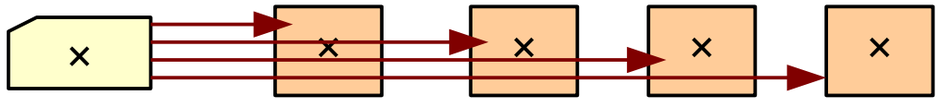
- Selection conditionnelle pour chaque fil
  - Trous sur cartes perforées contrôlent quels fils sont visibles sur chaque ligne : masque
  - Agit **en parallèle** sur 600 à 800 fils de chaîne (*warp*)

Single Instruction, Multiple Data

- Retour à la Machine Analytique
  Une instruction (ex. multiplication)
  peut être appliquée à plusieurs données
  - Ada Lovelace:
    “The ***same operation*** would be performed
    on different ***subjects of operation***.”



- Sans dépendances entre opérations : opération vectorielle
  - Calculs possibles en parallèle
  - (les vecteurs n'avaient pas encore été inventés du temps de Lovelace)



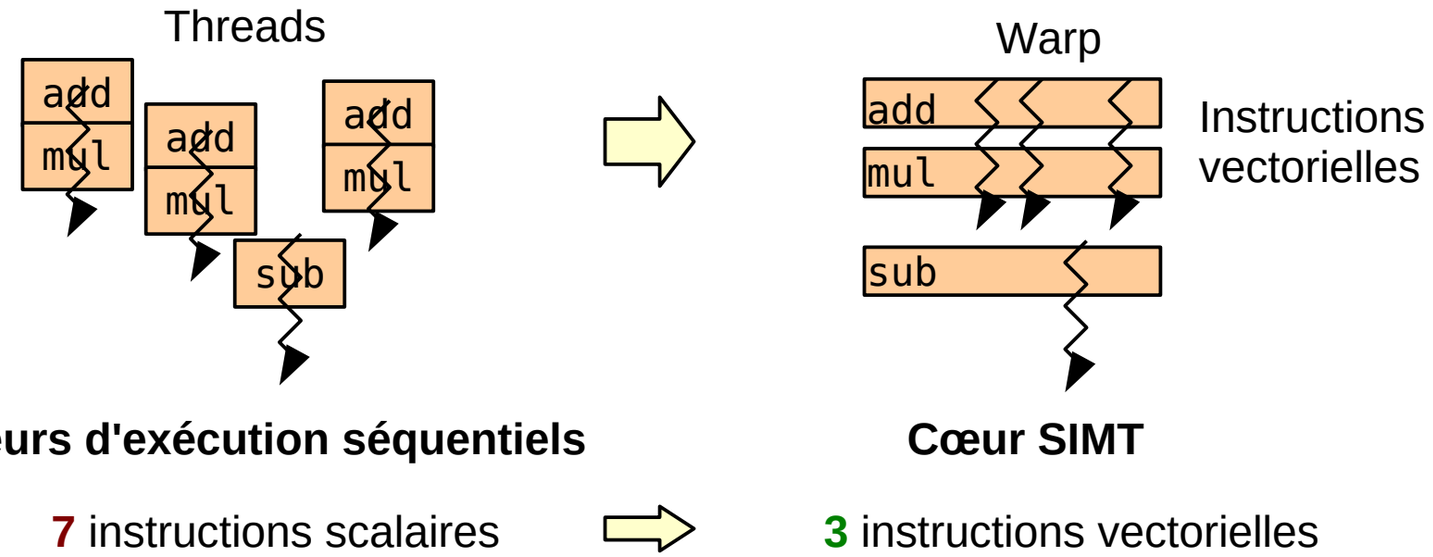
Single Instruction, Multiple Threads (SIMT)

Idée clé des architectures GPU

- Un grand nombre de *threads* exécutent le **même programme** (ex. pixels, sommets…)
- Synchroniser les threads pour qu'ils executent la **même instruction** au **même moment**
- Exécuter les instructions communes sur des **unités vectorielles efficaces**

Même programme

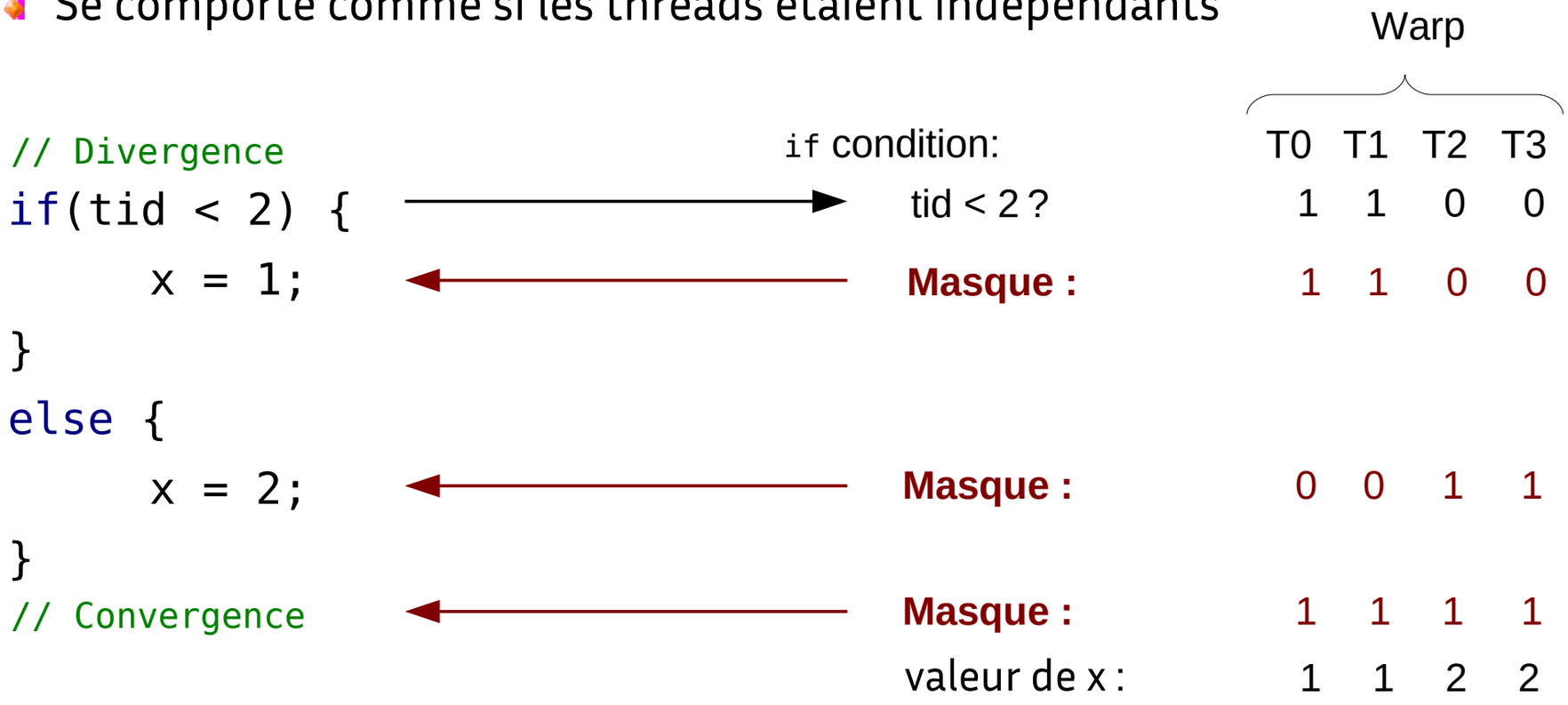
```
add r1, r3
mul r2, r1
...
sub r3, r0
```



Contrôle différencié par masquage

Différents threads peuvent prendre différents chemins dans le programme

- Prédiquer chaque instruction vectorielle par un masque de bits
  - 1 bit par thread
    1 → exécuter l'instruction
    0 → ne rien faire
  - Le GPU visite chaque chemin suivi par au moins un thread, et calcule le masque d'exécution
  - Se comporte comme si les threads étaient indépendants

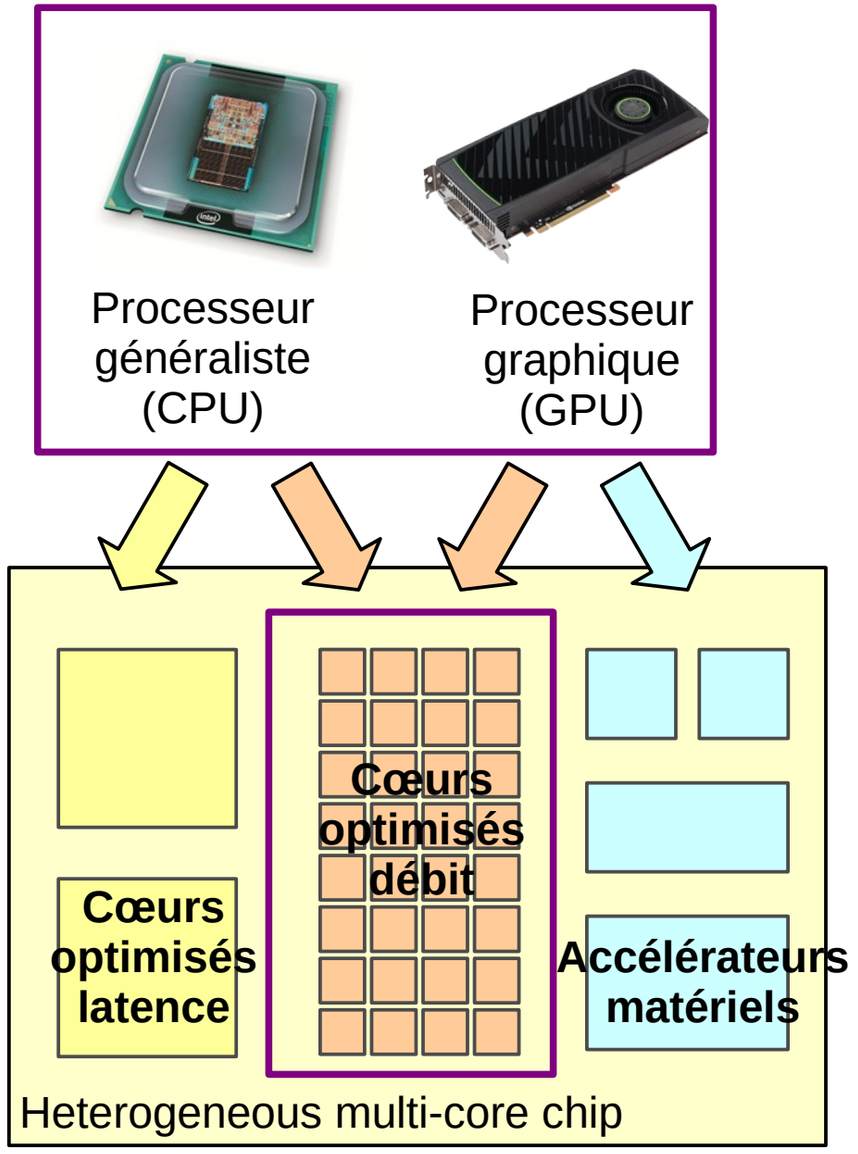


Plan

- Point de départ : la Machine Analytique de Babbage et Lovelace
- Processeurs généralistes : aller plus vite
  - Prédiction de branchements
  - Renommage de registres
- Processeurs graphiques : faire plus de choses
  - Retour au 19e siècle
  - Exécution vectorielle masquée
- Recherche actuelle : peut-on concilier les deux ?
  - Points bloquants : exécution masquée et renommage
  - PIRAT à la rescousse

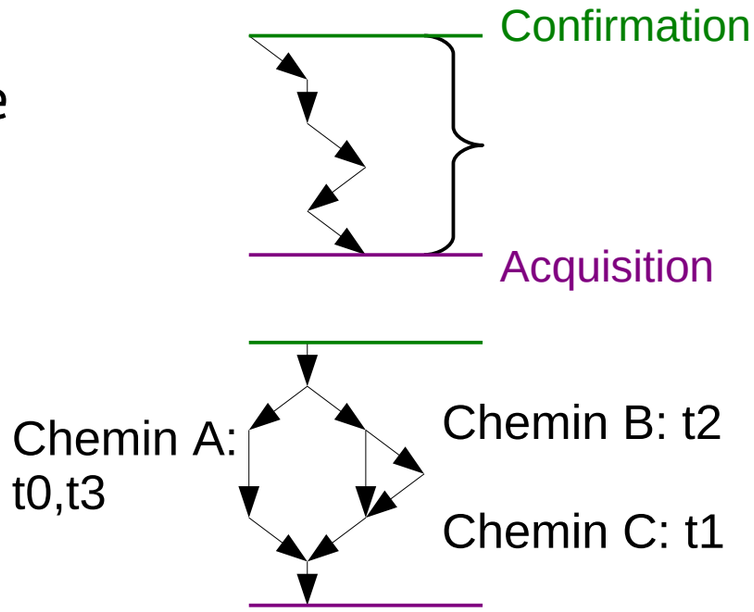
Des CPU et GPU aux futurs multi-cœurs hétérogènes

- Hier (2000-2010)
  - CPU et GPU sont des composants distincts
- Aujourd'hui (2011-...)
  - CPU et GPU sont physiquement sur la même puce
  - Mais toujours logiquement distincts
- Demain
  Multi-cœurs hétérogènes
  - Peut-on combiner les architectures CPU et GPU ?



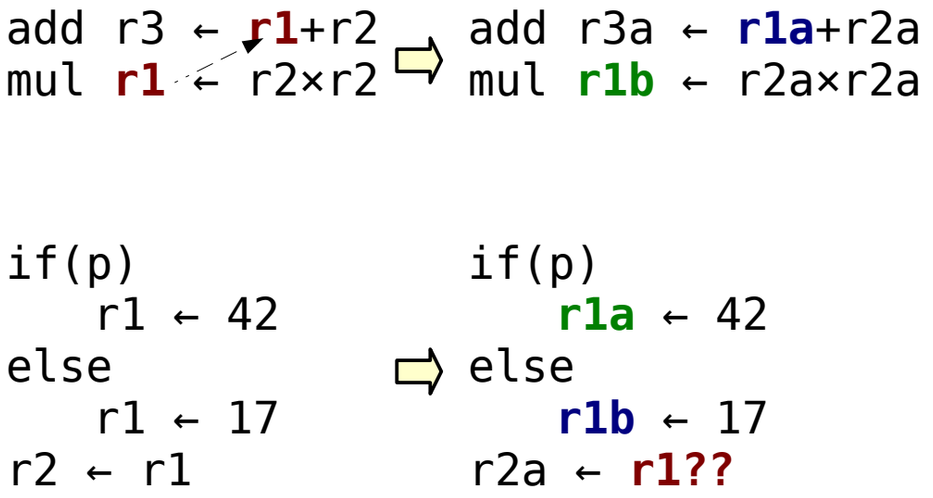
SIMT vs prédiction de branchements

- Prédiction de branchements : prédit un seul chemin
- SIMT: le flux d'exécution est un **graphe**, pas une séquence
  - Les groupes de threads peuvent converger, diverger
  - **1 chemin** à travers le graphe : 1 ensemble de threads
- Problème : sans ordre total, comment revenir en arrière ?

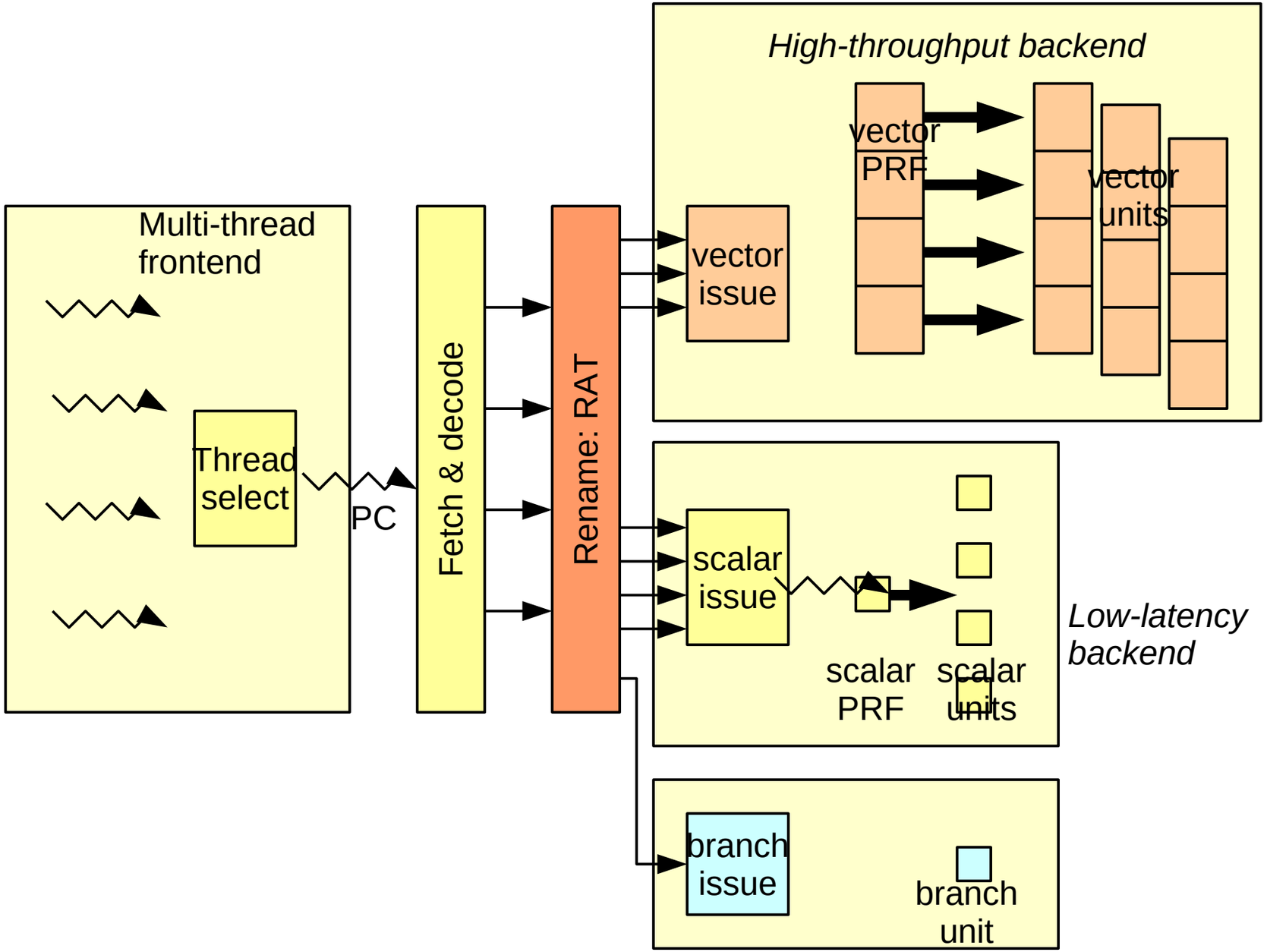


SIMT vs renommage

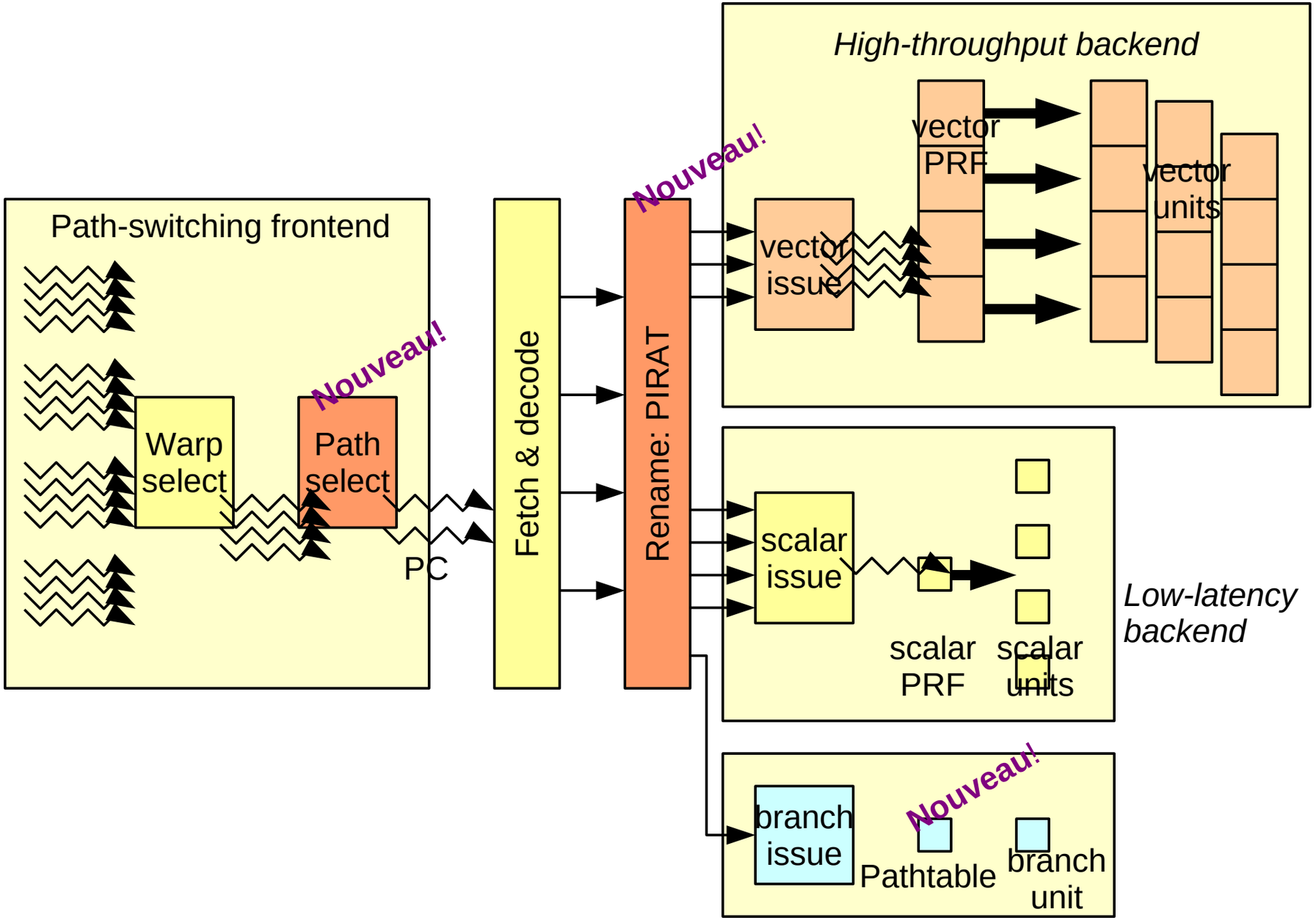
- Renommage de registres pour éliminer les dépendances écriture-après-lecture
- Cas SIMT: dépendances partielles
  - Instructions implicitement masquées
  - Écriture partielle de registres vectoriels
  - Un registre peut avoir **plusieurs producteurs**
- Problème : différents threads accèdent à différents registres : pas de nom unique



Architecture CPU de base

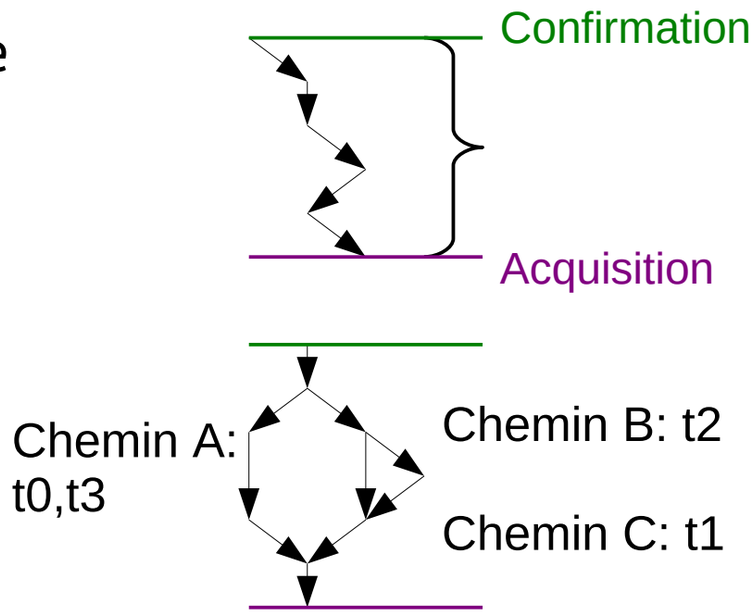


Architecture CPU SIMT proposée



SIMT vs prédiction de branchements : solution

- Prédiction de branchements : prédit un seul chemin
- SIMT: le flux d'exécution est un **graphe**, pas une séquence
  - Les groupes de fils d'exec peuvent converger, diverger
  - **1 chemin** à travers le graphe : 1 ensemble de threads
- Problème : sans ordre total, comment revenir en arrière ?

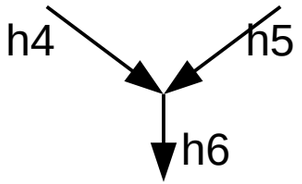


- Solution
  - Assigner un ordre total à l'acquisition,
    restaurer le **même ordre** à la confirmation
  - Convergence à l'acquisition, divergence à l'exécution
    → chaque thread suit **1 chemin spéculatif à la fois**
  - Suivi des masque dans une *Table de chemins*
  - Confirmation masquée
    → operations avec un masque entièrement nul sont ignorées, ne produisent pas d'effet

Anita Tino, Caroline Collange, André Seznec. *SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores* . TACO 2020

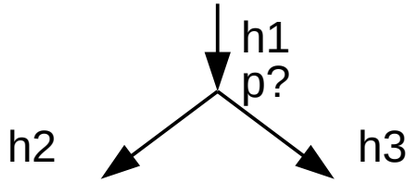
Suivi des chemins

- Chaque chemin a un masque de bits
  - Représente l'ensemble des threads qui suivent ce chemin
  - Initialement spéculatif, par sur-approximation
  - Connu entièrement à la confirmation des instructions du chemin
- Des **registres de chemins** maintiennent les masques des chemins en cours
- Registres de chemin lus et écris par des micro-instructions de convergence et divergence
  - Transforme le flux de contrôle en **flux de données**



h6 = h4 | h5

Convergence



Clear bits p in all paths from h2;
h3 = h1 & p

Divergence

Gestion des chemins : exemple

- Chemin initial emprunté par tous les fils d'exécution
  - h1 = 1111

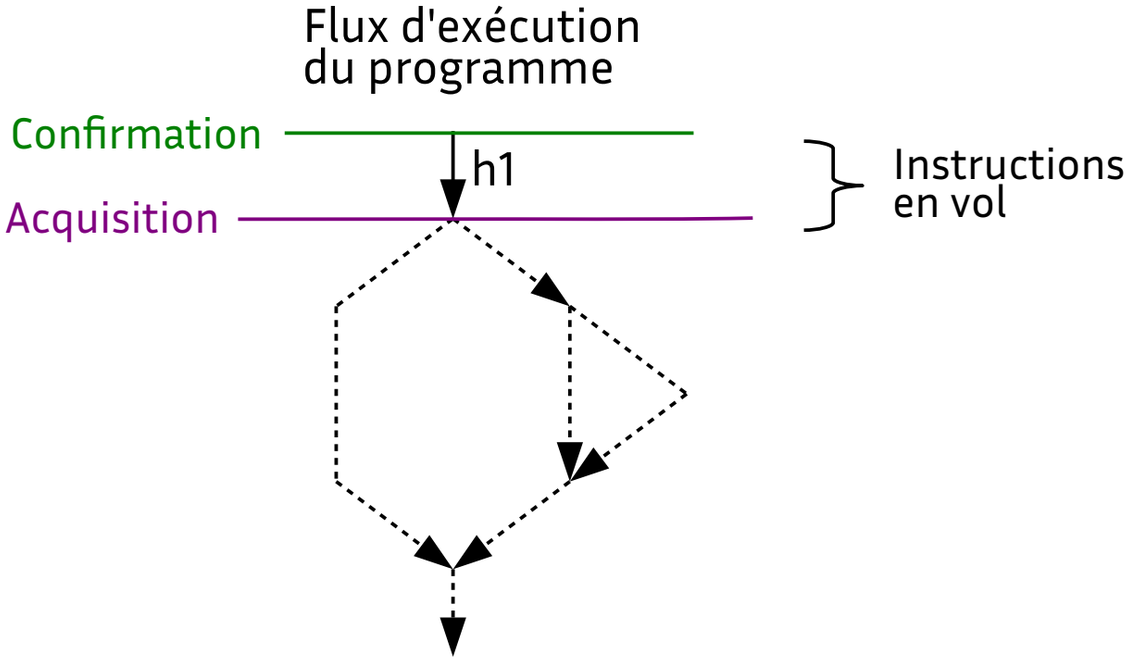


Table de chemins

Chemin	Masque
h1	1111

Gestion des chemins : exemple

- Acquisition du branchement
  - Démarrage d'un nouveau chemin h2 = h1 pour préparer potentielle divergence

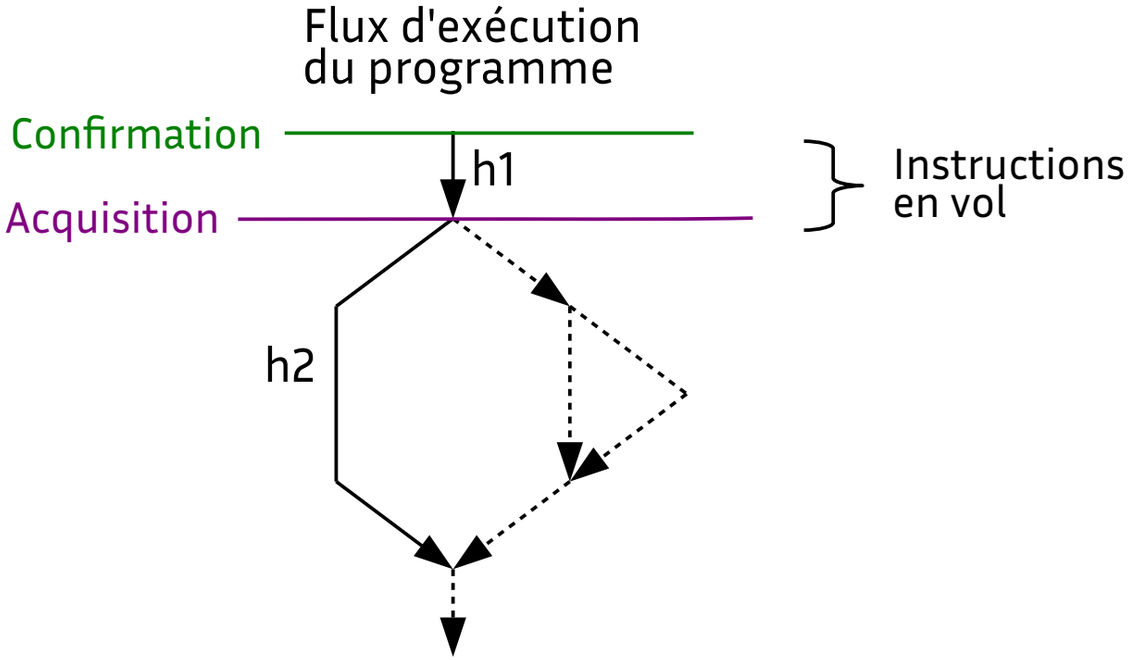


Table de chemins

Chemin	Masque
h1	1111
h2	1111

Gestion des chemins : exemple

- Résolution du branchement, divergence des threads 1-2
  - Annulation des threads 1-2 dans tous les chemins après le branchement : h2 &= ~0110
  - Démarrage d'un nouveau chemin h3 = h1 & 0110

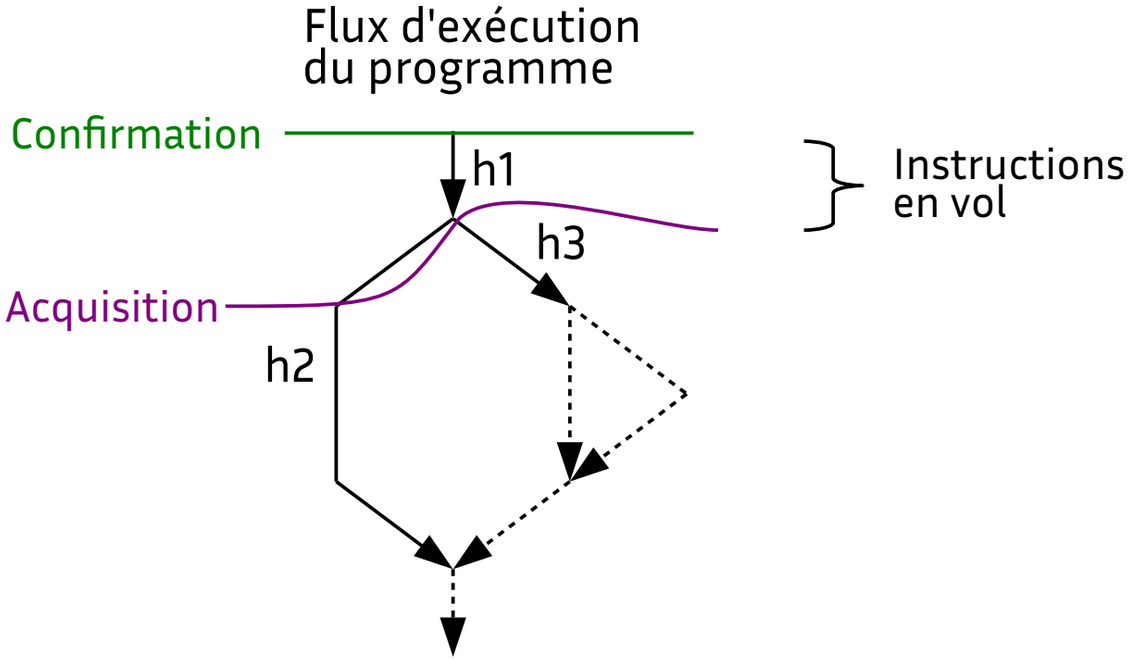


Table de chemins

Chemin	Masque
h1	1111
h2	1**00**1
h3	0**11**0

Gestion des chemins : exemple

- Prédiction de divergence probable
  - h4 = h3

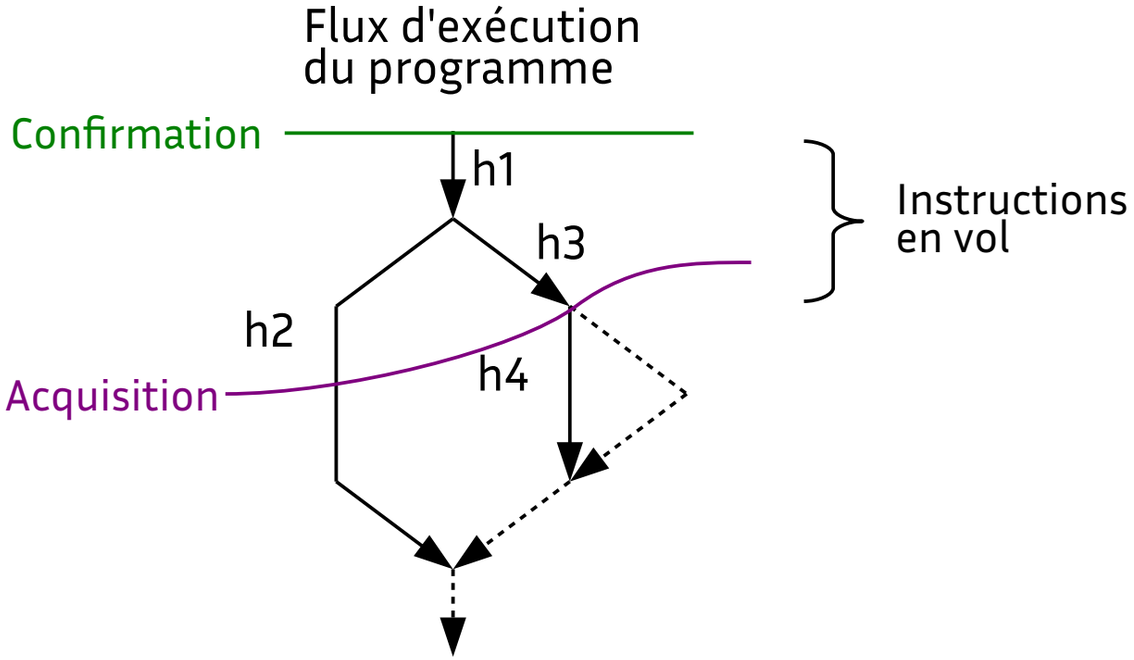


Table de chemins

Chemin	Masque
h1	1111
h2	1001
h3	0110
h4	0110

Gestion des chemins : exemple

- Divergence
  - h5 = h3 & 0100
  - h4 &= ~0100

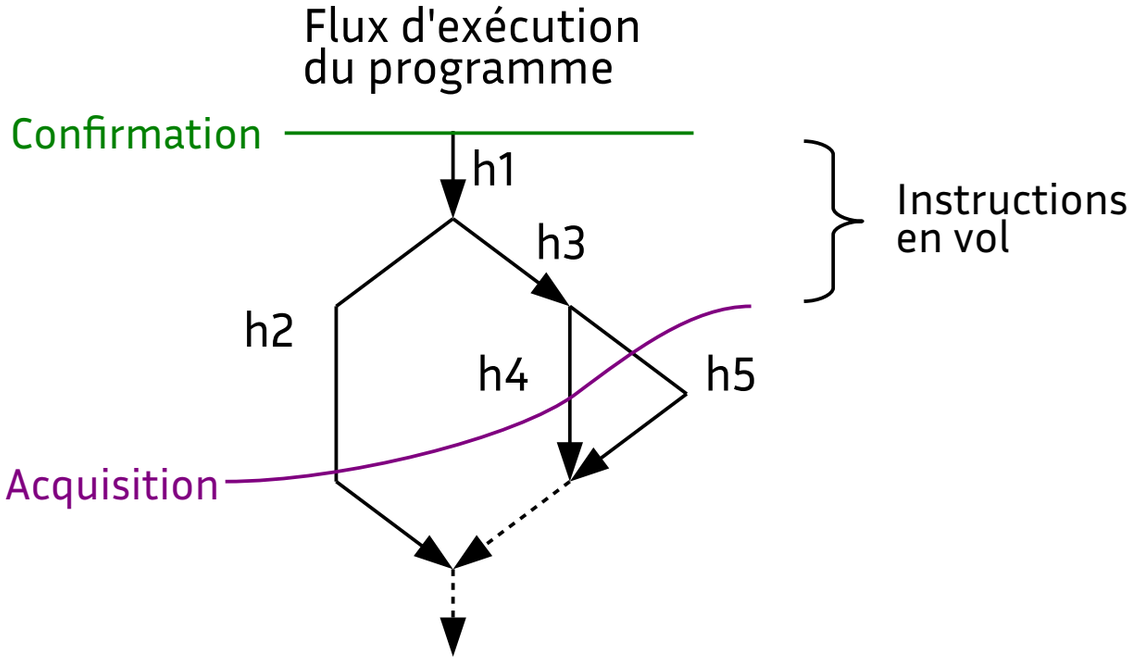


Table de chemins

Chemin	Masque
h1	1111
h2	1001
h3	0110
h4	0**0**10
h5	0**1**00

Gestion des chemins : exemple

- Convergence: h6 = h4 | h5

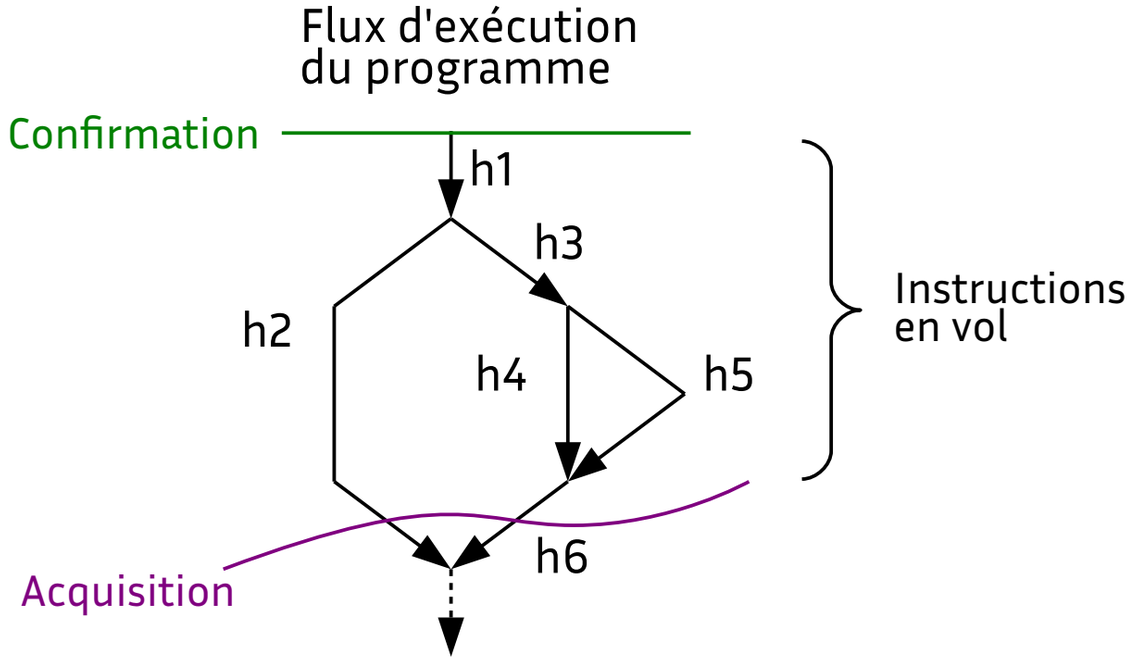


Table de chemins

Chemin	Masque
h1	1111
h2	1001
h3	0110
h4	0010
h5	0100
h6	0110

Gestion des chemins : exemple

- Libérer chemin h1 à la confirmation du branchement
- Convergence: h7 = h2 | h6

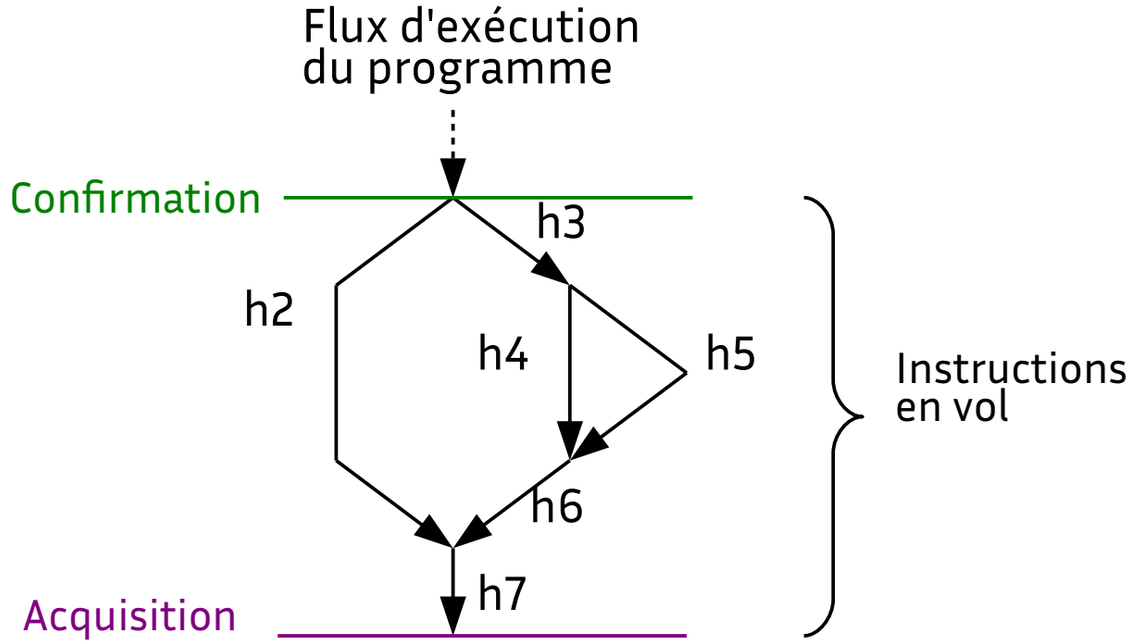
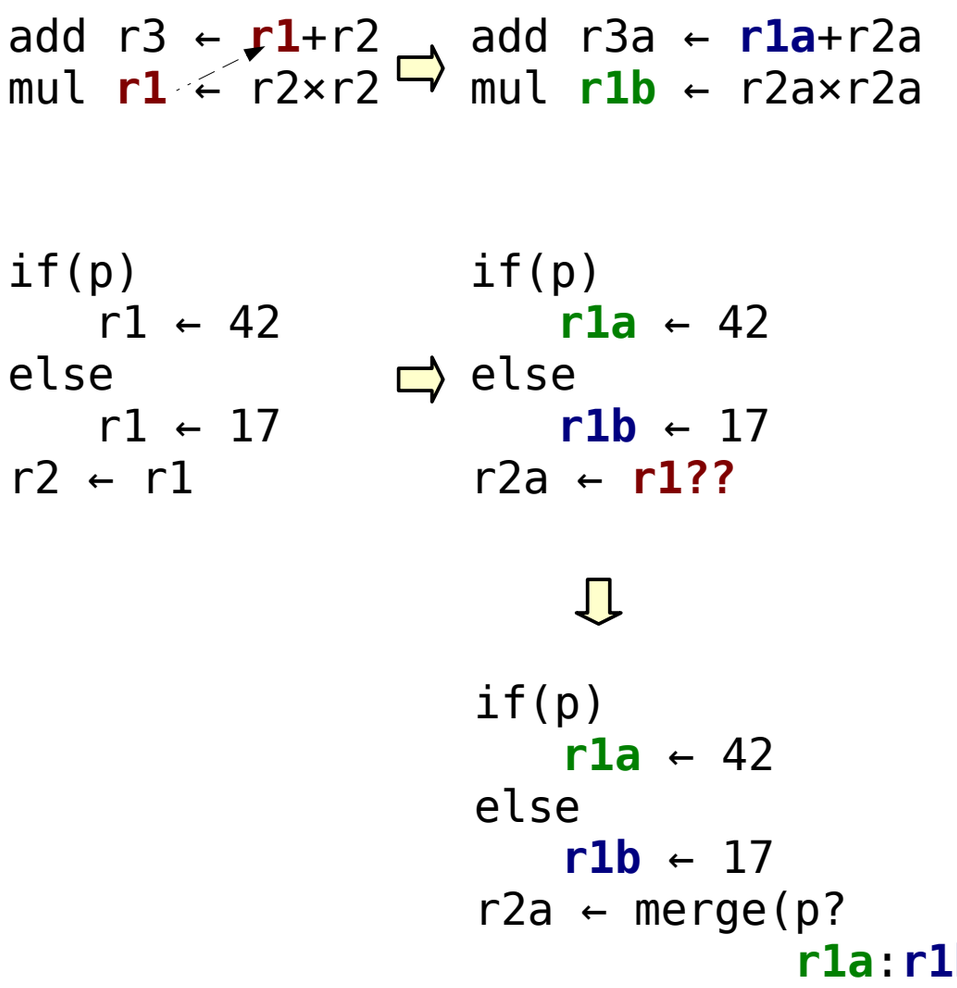


Table de chemins

Chemin	Masque
h2	1001
h3	0110
h4	0010
h5	0100
h6	0110
h7	1111

SIMT vs renommage : solution

- Renommage de registres pour éliminer les dépendances écriture-après-lecture
- Cas SIMT: dépendances partielles
  - Instructions implicitement masquées
  - Écriture partielle de registres vectoriels
  - Un registre peut avoir **plusieurs producteurs**
- Solution
  - **PIRAT** (Path-Identifying Register Alias Table)
    Différents chemins peuvent utiliser différents registres physiques
  - Injecter des **micro-opérations de fusion** à la demande après convergence



Renommage : deux chemins suffisent

Combien de registres physiques par registre architectural ?

- Autant que de threads/warp dans le pire cas
  - Bien trop pour réaliser en matériel
- Observation : 2 suffisent en pratique!
  - 1 pour le chemin actif courant, 1 pour les autres chemins
- En cas de dépassement, insérer des micro-opérations de fusion
  - Coût: dépendances entre instructions

```
r1 ← 42
if(p)
    r1 ← 17
r2 ← r1 + 1
```

```
p1 = 42
if(h1)
    p2 = 17
p4 = merge(h1?p2:p1)
p5 = p4 + 1
```

PIRAT

	Other	Active	
r1	**p1**	**p2**	h1
r2	p3		

Pathtable

	Mask
h1	1100

Se lit: r1 est dans **p2** pour les fils d'exec contenus dans **h1**, sinon dans **p1**

Renommage : deux chemins suffisent

Après fusion :

- L'entrée active de r1 est libérée
- L'autre entrée est utilisable comme source pour les instructions suivantes

```
r1 ← 42
if(p)
    r1 ← 17
r2 ← r1 + 1
```

```
p1 = 42
if(h1)
    p2 = 17
p4 = merge(h1?p2:p1)
p5 = p4 + 1
```

PIRAT

	Other	Active	
r1	**p4**		
r2	p5		

Pathtable

	Mask
h1	1100
h2	1111

Se lit: r1 est dans **p4** pour tous les fils d'exécution

Conclusions

- Contrôle séquentiel :
  prédiction de branchements
  - Pourquoi ? → aller plus vite
  - Comment ? → casser les dépendances de contrôle

- Contrôle parallèle :
  SIMT
  - Pourquoi ? → être plus efficace
  - Comment ? → factoriser le travail redondant

- Possible de combiner contrôle séquentiel et parallèle
  - Piste de recherche : étudier formellement propriétés de SIMT
- Outils similaires au niveau compilation : Static Single Assignment (SSA), gated SSA
  - Exposé de D. Demange le 15/02

Comment concilier contrôle et parallélisme ? Approches des architectures de processeurs généralistes et graphiques

8 février 2024
Collège de France

Caroline Collange
caroline.collange@inria.fr