

---

# Parallélisme d'instructions : Superscalaires versus VLIW

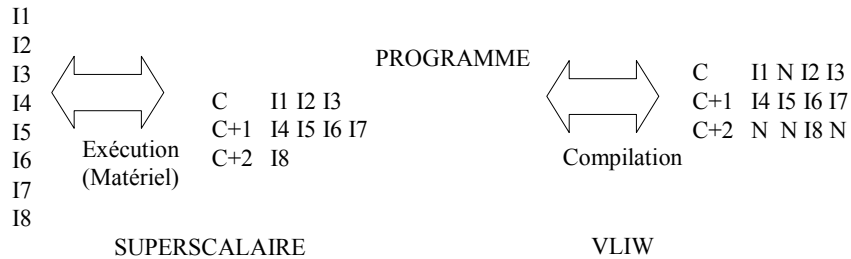
Daniel Etiemble  
de@lri.fr

## Résumé

---

- Lancement statique ou dynamique
- Superscalaires
  - Contrôle des dépendances
  - Exécution non ordonnée des instructions
- VLIW
  - Processeurs
    - Trimedia
    - C6x
    - IA-64
  - Pipeline logiciel

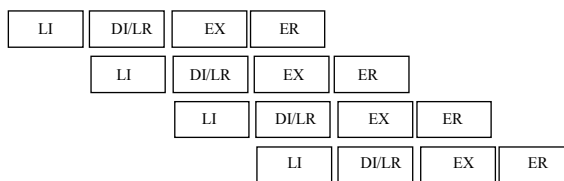
## Processeurs : superscalaire/VLIW



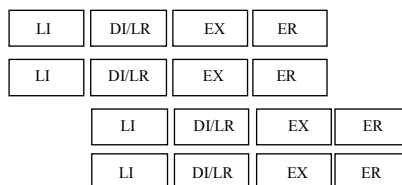
- **Superscalaire**
  - Le matériel est responsable à l'exécution du lancement parallèle des instructions
    - Contrôle des dépendances de données et de contrôle par matériel
- **VLIW**
  - Le compilateur est responsable de présenter au matériel des instructions exécutables en parallèle
    - Contrôle des dépendances de données et de contrôle par le compilateur

## Principes des superscalaires

pipeline



superscalaire



Superscalaire de degré n :  
n instructions par cycle

## Le contrôle des aléas

---

- Contrôle des dépendances de données
  - Réalisé par matériel
    - Scoreboard
    - Tomasulo
- Existe à la fois pour les processeurs scalaires (à cause des instructions multi-cycles) et les processeurs superscalaires (problèmes accentués)

## Problèmes liés aux superscalaires

---

- Nombre d'instructions lues
- Types d'instructions exécutables en parallèles
- Politique d'exécution
  - Exécution dans l'ordre
  - Exécution non ordonnée
- Dépendances de données
- Branchements

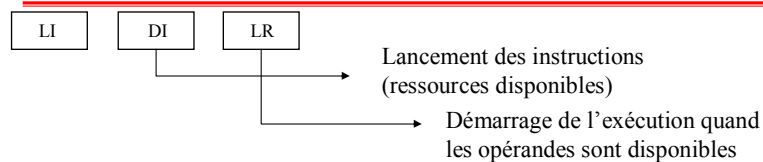
## Problèmes superscalaires (2)

---

- **Coût matériel accentué**
  - Plusieurs accès aux bancs de registres
  - Plusieurs bus pour les opérandes et les résultats
  - Circuits d'envoi (forwarding) plus complexes
- **Davantage d'unités fonctionnelles avec latences différentes**
  - plusieurs fonctions (Entiers, Flottants, Contrôle)
  - plusieurs unités entières
- **Plusieurs instructions exécutées simultanément**
  - Débit d'instructions plus important
  - Débit d'accès aux données plus important
  - Problème des sauts et branchements accentués

## Exécution superscalaire

---



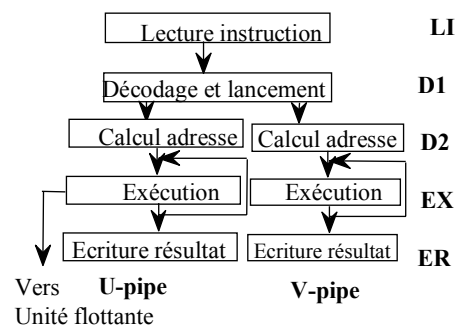
- **Exécution dans l'ordre**
  - Acquisition par groupe d'instructions
  - Traitement du groupe suivant quand toutes les instructions d'un groupe ont été lancées et démarrées.
- **Exécution non ordonnée**
  - Acquisition des instructions par groupe, rangées dans un tampon
  - Exécution non ordonnée des instructions, en fonction du flot de données
  - Fenêtre d'instructions liée à la taille du tampon.

## Exécution superscalaire x86

- Exécution directe du code x86
  - Pentium
- Traduction dynamique du code x86 en instructions de type RISC
  - Pentium Pro et successeur
  - AMD K6 et K7
- Emulation : traduction dynamique en code VLIW
  - Transmeta

## Exécution dans l'ordre : le Pentium

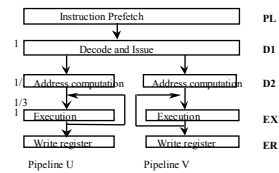
Deux pipelines U et V



## Pentium : phase exécution

### • REGLES

- Seules les instructions simples peuvent être appariées
- Quand une instruction est suspendue dans U, l'autre est suspendue dans V
- Quand une instruction est suspendue dans V, l'autre peut continuer, mais aucune autre instruction ne peut entrer dans EX avant que les précédentes n'aient atteint ER → Exécution au rythme de la plus lente.
- Quand deux instructions ont MEM pour destination, deux cycles supplémentaires sont nécessaires (5 au total)



mov reg, reg/mem/imm : 1  
 mov mem, reg/imm : 1  
 alu mem, reg/imm : 1/2/1  
 alu mem, reg/imm : 3  
 inc reg/mem : 1/3  
 dec reg/mem  
 push reg/mem : 1/2  
 pop reg/mem  
 lea reg, mem  
 jmp/call/jcc near : 1/2  
 nop : 1

## Exécution dans l'ordre (21164)

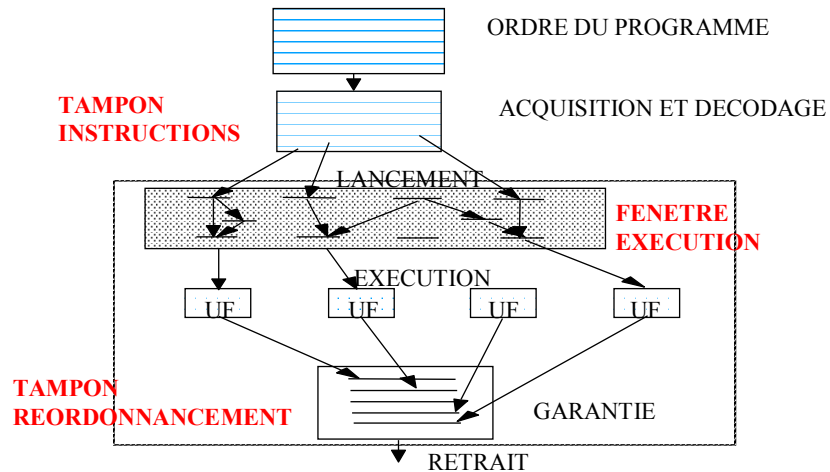
- 4 instructions acquises par cycle
- 4 pipelines indépendants

- 2 entiers
- 2 flottants

E0	E1	FA	FM
LD	LD	FADD	FMUL
ST	IBR	FDIV	
UAL	Jump	FBR	
CMOV	UAL		
COMP	CMOV		
	COMP		

- 1 groupe de 4 instructions considéré à chaque cycle
  - si les 4 instructions se répartissent dans E0, E1, FA et FM, elles démarrent leur exécution. Sinon, certaines utilisent les cycles suivants jusqu'à ce que les 4 aient démarré.
  - Les 4 suivantes sont traitées quand les 4 précédentes ont démarré.

## Exécution non ordonnée

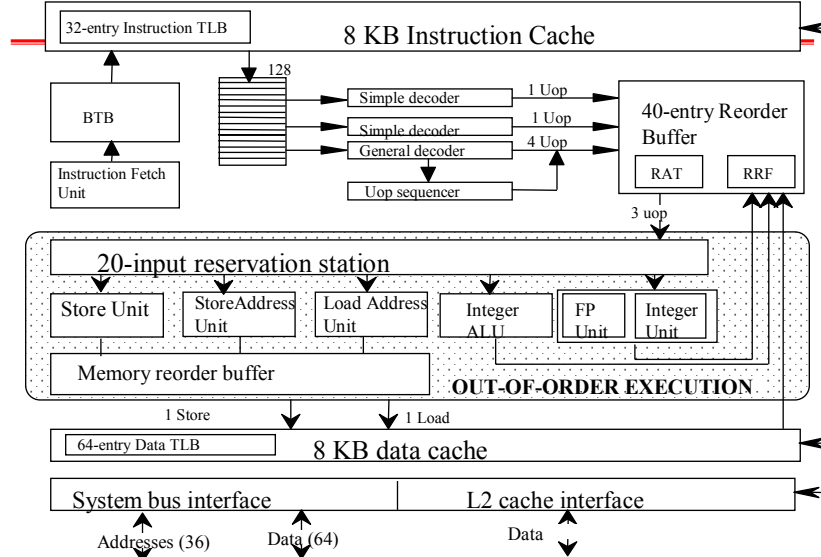


M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

13

## Schéma fonctionnel du Pentium Pro



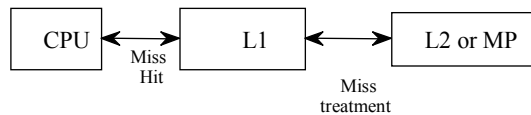
M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

14

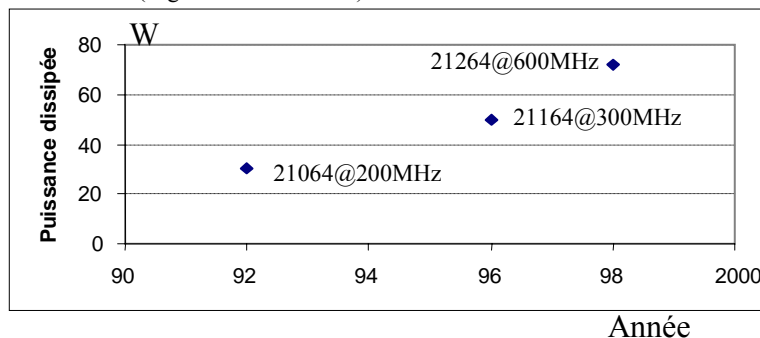
## Caches non bloquants

- Pour permettre l'exécution non ordonnée des instructions, le processeur doit pouvoir continuer à exécuter des Loads alors qu'un Load précédent a provoqué un défaut de cache.
- Un cache non bloquant permet plusieurs accès cache simultanés, et notamment de traiter des succès pendant le traitement d'un échec.



## Evolution de la puissance dissipée

- Exemple des processeurs Alpha
  - 21064 (degré 2 - dans l'ordre)
  - 21164 (degré 4 - dans l'ordre)
  - 21264 (degré 4 - non ordonné)



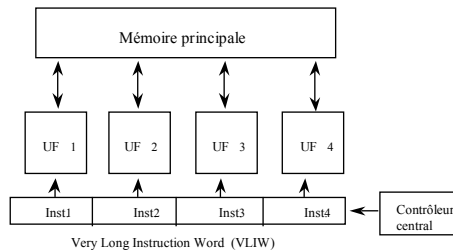
## du R5000 au R10000

### ***DIMINUSHING RETURN***

Caractéristique	R10000	R5000	Différence
Fréquence	200 MHz	180 MHz	équivalent
CMOS	0.35/4M	0.35/4M	
Cache	32K/32K	32K/32K	
Etages pipeline	5-7	5	
Modèle	Non ordonné	Dans l'ordre	
Lancement	4	1 + FP	
Transistors	5.9 millions	3.6 millions	+64%
Taille	298 mm <sup>2</sup>	84 mm <sup>2</sup>	
Développement	300 h x a	60 h x a	
SPECint95	10.7	4.7	+128%
SPECfp95	17.4	4.7	+270%
Power	30W	10W	200%
SPEC/Watt	0.36/0.58	0.47/0.47	-31%/23%

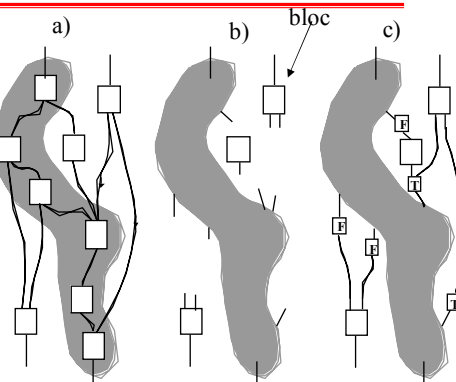
## ARCHITECTURE VLIW

- PRINCIPE
  - Plusieurs unités fonctionnelles
  - ORDONNANCEMENT STATIQUE (compilateur)
    - Aléas de données
    - Aléas de contrôle
  - Exécution pipelinée des instructions
  - Exécution parallèle dans N unités fonctionnelles
- Tâches du compilateur
  - Graphe de dépendances locales : Bloc de base
  - Graphe de dépendance du programme : Ordonnancement de traces
- Pour:
  - Matériel simplifié (contrôle)
- Contre:
  - La compatibilité binaire entre générations successives est difficile ou impossible



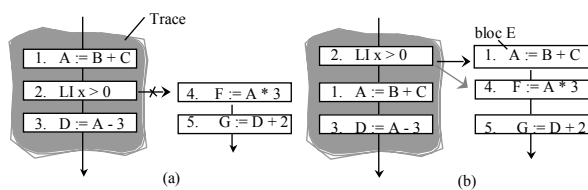
## VLIW: “Ordonnancement de traces”

- Ordonnancement des blocs de base
- Ordonnancement à travers les blocs
  - Sélectionner une trace
  - Optimiser la trace, avec les problèmes d’entrée et sortie de la trace
  - Ajouter des blocs pour connecter les autres chemins à la trace
    - Sortant de la trace
    - Entrant dans la trace

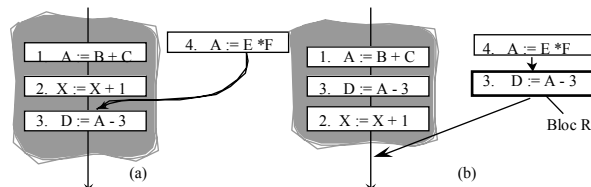


## VLIW: “Ordonnancement de traces”

### Blocs sortant de la trace

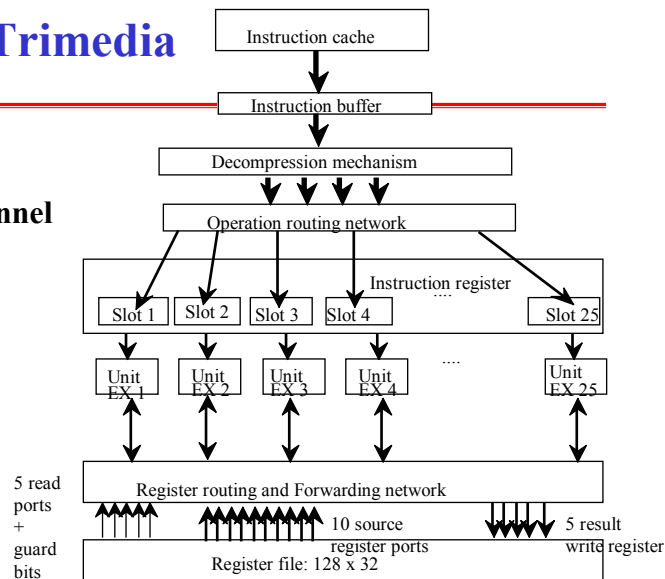


### Blocs entrant dans la trace



## VLIW - Trimedia (Philips)

### Schéma fonctionnel

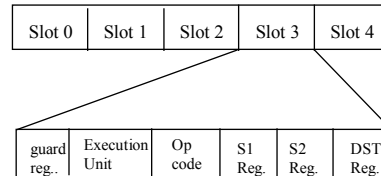


## Jeu d'instructions Trimédia

- VLIW : 5 instructions de n bits
- 128 registres généraux
  - $r0 = 0$
  - $r1 = 1$
- Exécution conditionnelle de toute instruction, avec spécification d'un registre général contenant le prédicat (vrai si 1; faux si 0)

## VLIW: Trimedia (Philips)

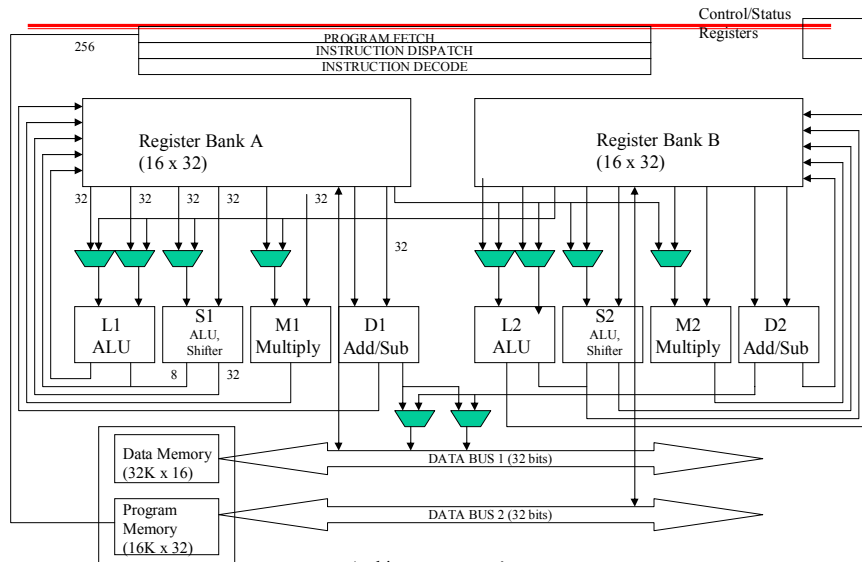
- Format d'instructions
  - Registres de garde : indiquent si l'opération est exécutée (opérations conditionnelles)
  - Techniques de compression
    - Supprimer les NOP des opérations non utilisées
  - Branchements retardés
    - 3 cycles
- Objectifs
  - VLIW + DSP
  - Multimédia



## Jeu d'instructions C6x

- VLIW : 8 x 32 bits
- Instructions 32 bits, correspondant à 4 x 2 unités fonctionnelles
  - voir schéma fonctionnel
- 2 ensemble A et B de 16 registres généraux
- Instructions à exécution conditionnelle
  - 5 registres généraux A1, A2, B0, B1, B2 peuvent contenir la condition
    - différent zéro
    - égal 0

## Schéma fonctionnel du C6x



M2 SET1  
2005-2006

Architectures avancées  
Daniel Etiemble

25

## Modes d'adressage

Type d'adressage	No modification	Préincrément ou Décrément du registre adresse	Postincrément ou Décrément du registre adresse
Registre Indirect	*R	*++R *--R	*R++ *R--
Registre + déplacement	*+R[ucst5] *-R[ucst5]	*++R[ucst5] *--R[ucst5]	*R++[ucst5] *R--[ucst5]
Base + Index	*+R[offsetR] *-R[offsetR]	*++R[offsetR] *--R[offsetR]	*R++ *R--

M2 SET1  
2005-2006

Architectures avancées  
Daniel Etiemble

26

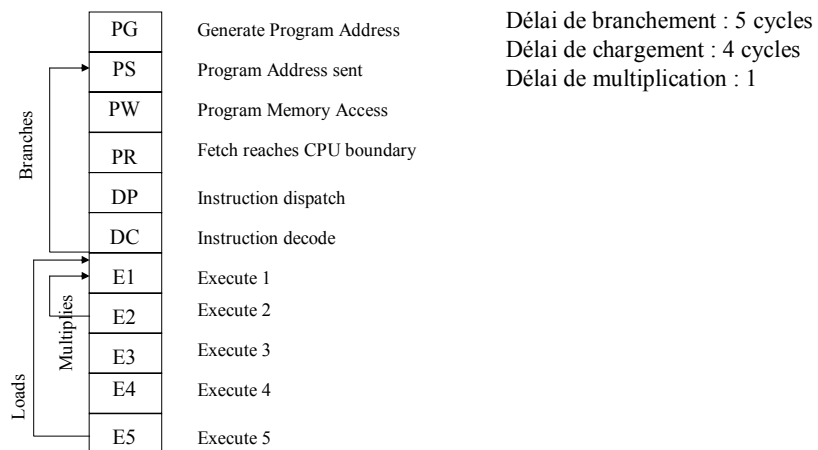
## Instructions et unités fonctionnelles

---

.L Unit		.M Unit	.S Unit		.D Unit
ABS	SADD	MPY	ADD	MVKH	ADD
ADD	SAT	SMPY	ADDK	NEG	ADDA
AND	SSUB	MPYH	ADD2	NOT	LD mem
CMPEQ	SUB		AND	OR	LD mem (15 bit) (D2)
C.PGT	SUBC		B disp	SET	MV
CMPGTU	XOR		B IRP	SHL	NEG
CMPLT	ZERO		B NRP	SHR	ST mem
CMPLTU			Breg	SHRU	ST mem (15 bit) (D2)
LMBD			CLR	SSHL	SUB
MV			EXT	SUB	SUBA
NEG			EXTU	SUB2	ZERO
NORM			MVC (S2)	XOR	
NOT			MV	ZERO	
OR			MVK		

## PIPELINE C6x

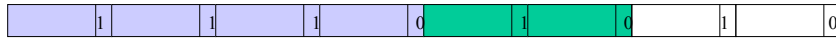
---



## C6x: Acquisition des paquets

---

32 bits



Le bit de poids faible de chaque instruction indique si l'instruction peut être exécutée en parallèle avec l'instruction qui suit.

Toujours 0

## Programmation VLIW (C6x)

---

- Allocation des ressources
  - Instructions et unités fonctionnelles
- Graphes de dépendances
  - Latences (Chargement, Multiplication, Branchement)
- Déroulage de boucle
- Pipeline logiciel
  - Intervalle d'itérations
  - Cas des conditionnelles
  - Durée de vie des registres

## Programmation C6x : produit scalaire (1)

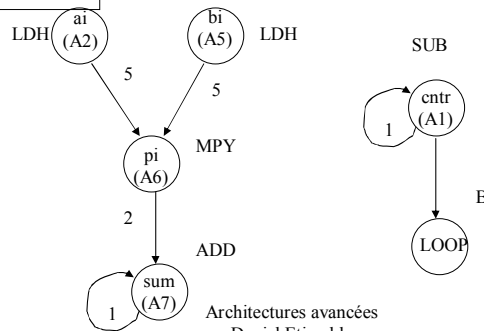
Code C

```
int dotp(short[a], short[b])
{
  int sum, i;
  sum=0;
  for (i=0; i<100;i++)
    sum+=a[i]*b[i];
  return (sum);
}
```

Code assembleur

```
LOOP:  LDH    .D1    *A4++,A2 ; load ai
        LDH    .D1    *A3++,A5 ; load bi
        MPY    .M1    A2, A5, A6 ; ai*bi
        ADD    .L1    A6, A7,A7 ; sum+=(ai*bi)
        SUB    .S1    A1,1,A1 ; decrement loop counter
[A1]   B      .S2    LOOP ; branch to loop
```

GRAPHE  
DEPENDANCE



M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

31

## Programmation C6x : produit scalaire (2)

CODE ASSEMBLEUR NON PARALLELE

```
        MVK    .S1    100,A1 ; set up loop counter
        ZERO   .L1    A7 ; zero out accumulator
LOOP    LDH    .D1    *A4++,A2 ; load ai
        LDH    .D1    *A3++,A5 ; load ai
        NOP    4 ; delay slots for LDH
        MPY    .M1    A2, A5, A6 ; ai*bi
        NOP    1 ; delay slot for MPY
        ADD    .L1    A6, A7,A7 ; sum+=(ai*bi)
        SUB    .S1    A1,1,A1 ; decrement loop counter
[A1]   B      .S2    LOOP ; branch to loop
        NOP    5 ; delay slots for branch
```

16 cycles/itération

M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

32

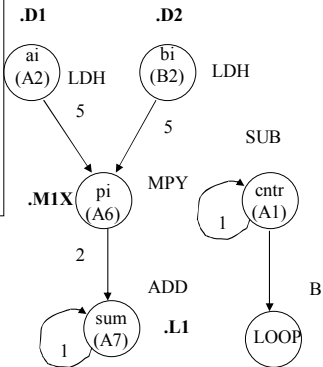
## Programmation C6x : produit scalaire (3)

```

||      MVK      .S1      100,A1      ; set up loop counter
||      ZERO     .L1      A7          ; zero out accumulator
LOOP:
||      LDH      .D1      *A4++,A2    ; load ai
||      LDH      .D2      *B4++,B2    ; load bi
||      SUB      .S1      A1,1,A1     ; decrement loop counter
[A1]   B        .S2      LOOP        ; branch to loop
||      NOP      2          ; delay slots for LDH
||      MPY      .M1      A2, A5, A6   ; ai*bi
||      NOP      1          ; delay slot for MPY
||      ADD      .L1      A6, A7,A7    ; sum+=(ai*bi)
; branch occurs here

```

8 cycles for each iteration



M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

33

## Programmation C6x : produit scalaire (4)

```

int dotp(short[a], short[b])
{
  int sum, sum0, sum1, i;
  sum0=0;
  sum1=0
  for (i=0; i<100;i+=2){
    sum0+=a[i]*b[i];
    sum1+=a[i+1]*b[i+1];
  }
  sum=sum0+sum1;
  return (sum);
}

```

```

LDW      .D1      *A4++,A2    ; load ai and ai+1
LDW      .D2      *B4++,B2    ; load bi and bi+1
MPY      .M1X     A2, B2, A6   ; ai*bi
MPYH     .M2X     A2, B2, B6   ; ai+1*bi+1
ADD      .L1      A6, A7,A7    ; sum0+=(ai*bi)
ADD      .L2      B6, B7,B7    ; sum1+=(ai+1*bi+1)
SUB      .S1      A1,1,A1     ; decrement loop counter
[A1]   B        .S2      LOOP        ; branch to loop

```

M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

34

## Programmation C6x : produit scalaire (5)

---

	MVK	.S1	100,A1	; set up loop counter
	ZERO	.L1	A7	; zero out sum0
	ZERO	.L2	B7	; zero out sum1
LOOP:				
	LDW	.D1	*A4++,A2	; load ai and ai+1
	LDW	.D2	*B4++,B2	; load bi and bi+1
	SUB	.S1	A1,1,A1	; decrement loop counter
[A1]	B	.S2	LOOP	; branch to loop
	NOP	2		; delay slots for LDH
	MPY	.M1X	A2, B2, A6	; ai*bi
	MPYH	.M2X	A2, B2, B6	; ai+1*bi+1
	NOP	1		; delay slot for MPY
	ADD	.L1	A6, A7,A7	; sum0+=(ai*bi)
	ADD	.L2	B6,B7,B7	; sum1+=(ai+1*bi+1)
; branch occurs here				
	ADD	.L1X	A7,B7,A4	; sum=sum0+sum1

8 cycles for 2 iterations

## Programmation C6x : produit scalaire (6)

### Table des intervalles entre itérations

Unit/cy	0	1	2	3	4	5	6	7
.D1	LDW							
.D2	LDW							
.M1						MPY		
.M2						MPYH		
.L1								ADD
.L2								ADD
.S1		SUB						
.S2			B					

Unit/cy	0	1	2	3	4	5	6	7
.D1	LDW	LDW1	LDW2	LDW3	LDW4	LDW5	LDW6	LDW7
.D2	LDW	LDW1	LDW2	LDW3	LDW4	LDW5	LDW6	LDW7
.M1						MPY	MPY1	MPY2
.M2						MPYH	MPYH1	MPYH2
.L1								ADD
.L2								ADD
.S1		SUB	SUB1	SUB2	SUB3	SUB4	SUB5	SUB6
.S2			B	B1	B2	B3	B4	B5

## Programmation C6x : produit scalaire (7)

### Pipeline logiciel

---

```
LOOP:
    LDW    .D1    *A4++,A2 ; load ai and ai+1 (+7)
    LDW    .D2    *B4++,B2 ; load bi and bi+1 (+7)
    [[A1]  SUB    .S1    A1,1,A1 ; decrement loop counter (+6)
    [[A1]  B      .S2    LOOP ; branch to loop (+5)
    MPY    .M1X   A2, B2, A6 ; ai*bi (+2)
    MPYH   .M2X   A2, B2, B6 ; ai+1*bi+1 (+2)
    ADD    .L1    A6, A7,A7 ; sum0+=(ai*bi)
    ADD    .L2    B6,B7,B7 ; sum1+=(ai+1*bi+1)

; branch occurs here
    ADD    .L1X   A7,B7,A4 ; sum=sum0+sum1
```

1 cycle /iteration

PROLOGUE  
EPILOGUE

## EPIC (IA64)

---

- **Parallélisme explicite**
  - Parallélisme d'instructions explicite dans le code machine (approche VLIW)
  - Le compilateur fait l'ordonnement à grande échelle
  - Compatibilité binaire (x86, HP-PA)
- **Les caractéristiques qui augmente le parallélisme d'instructions**
  - Prédication
  - Spéculation
  - Autres...
- **Ressources pour l'exécution parallèle**
  - Beaucoup de registres (128 entiers, 128 flottants)
  - Beaucoup d'unités fonctionnelles...

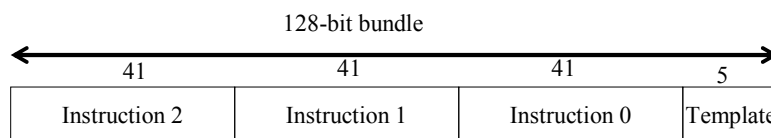
## Jeu d'instructions IA64

---

- VLIW de 128 bits, contenant 3 instructions
- 128 registres entiers et 128 registres flottants
- 64 registres 1 bit pour les prédicats
- Exécution conditionnelle de toute instruction, en fonction d'un registre prédicat
- Chargements spéculatifs
- Support pour pipeline logiciel

## Format d'instructions IA64

---



Type instruction	Description	Type unité exécution
A	UAL entiers	Unité I ou M
I	Entiers non-UAL	Unité I
M	Mémoire	Unité M
F	Flottant	Unité F
B	Branchements	Unité B
L+X	Etendu	Unité I

## Templates

---

Arrêts architecturaux

Mappage des emplacements d'instructions sur les unités d'exécution

MII  
MLX  
MMI  
MFI  
MMF  
MIB  
MBB  
BBB  
MMB  
MFB

## Organisation des registres

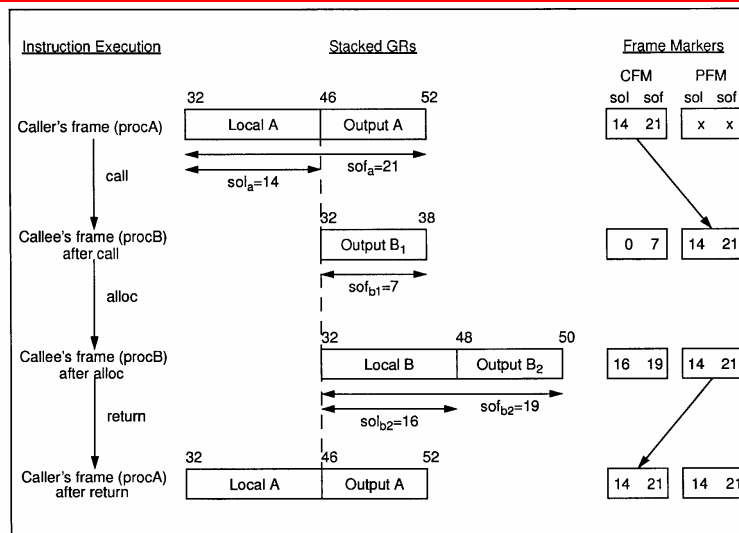
---

- Registres généraux : 128 x 65 bits; GR0-GR127
  - 64 bits données + 1 bit (*NaT: Not a Thing*)
  - Registres statiques : GR0-GR31
    - GR0 = 0
    - Variables globales
  - Registres en pile
    - GR32-GR127
    - Allocation d'un nombre programmable de registres locaux et de sortie

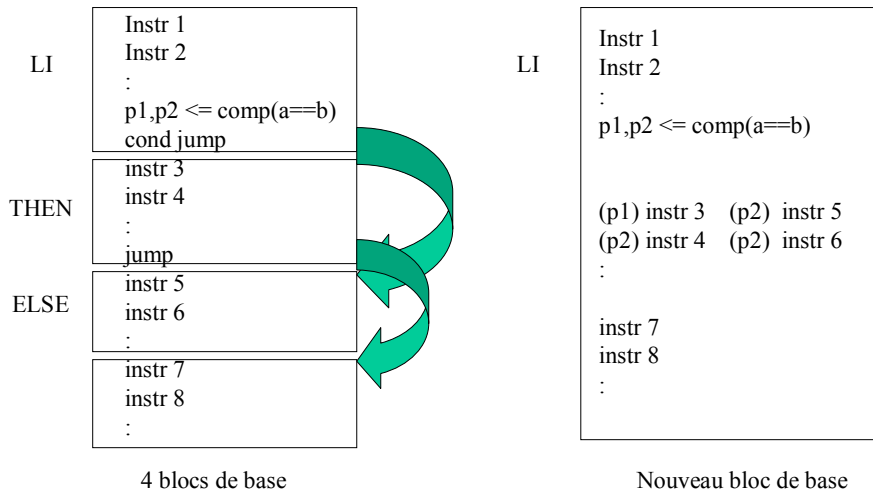
## Organisation des registres (2)

- Registres flottants : 128 x 82 bits
  - Registres statiques FR0-FR31
    - FR0 = +0.0
    - FR1 = +1.0
  - Registres tournants FR32-FR127
- Registres prédicat : 64 x 1bit
  - Registres statiques : PR0-PR15
    - PR0=1
  - Registres tournants PR16-63
- Registres de branchements: 8 x 64 bits

## Passage de paramètres



## L'utilisation des prédicats



## Les prédicats dans l'IA64

**p1, p2 ← if (qp) Compare (S1,S2)**      (comp) = Comparaison (S1,S2)  
 (qp) = prédicat courant

		<b>p1</b>		<b>p2</b>	
Normal	-	(qp)	(comp)	(qp)	(¬ comp)
UNC		(qp) (¬qp)	(comp) 0	(qp) (¬ qp)	(¬ comp) 0
AND	and andcm	(qp& ¬ comp) (qp&comp)	0 0	(qp& ¬ comp) (qp&comp)	0 0
OR	or orcmm	(qp&comp) (qp& ¬ comp)	1 1	(qp&comp) (qp& ¬ comp)	1 1
De Morgan	orandcm and.orcm	(qp&comp) (qp& ¬ comp)	1 0	(qp&comp) (qp& ¬ comp)	0 1

## Exemple

```

if (r1)
    r1=r3+r4;
else
    r7=r6-r5;
    
```

	comp.eq p1,p2=r1,r0	//0
(p1)	br.cond else;	//0
	add r1=r3,r4	//1
	br end_if	//1
else	sub r7=r6,r5	//1
	end_if	

	comp.eq p1,p2=r1,r0	//0
(p1)	add r2=r3,r4	//1
(p2)	sub r7=r6,r5	//1

30% de mauvaises prédictions  
10 cycles par mauvaise prédiction  
5 cycles

## Equilibrage des chemins

```

if (r4) //2
    r3=r2+r1;
else //18
    r3=r2*r1;
    
```

setf : 8 cycles      Coût mauvaise prédiction =  
xma : 6                      10 cycles  
getf : 2

		T	F
cmp.ne	p1, p2=r3, r0 ;	// 0	0
(p1)	add r3=r2,r1	//1	1
(p2)	setf f1=r1	//1	1
(p2)	setf f2=r2 ;;	//1	1
(p2)	xma.1 f3=f1,f2,f0 ;;/9	//9	2
(p2)	getf r3=f3 ;;	//15	3
(p2)	use of r3	//17	4
		18	5

% if pris	mauvaise prédiction
50%	0%
Cas 1 : 2*0.5 + 18*0.5 = <b>10</b>	
Cas 2 : 5*0.5 + 18*0.5 = 11.5	
70%	10%
Cas 1 : 2*0.7 + 18*0.3+10*0.1 = <b>7.8</b>	
Cas 2 : 5*0.7 + 18*0.3 = 8.9	
30%	30%
Cas 1 : 2*0.3 + 18*0.7+10*0.3 = 16.2	
Cas 2 : 5*0.3 + 18*0.7 = <b>14.1</b>	

## Parallel Compare

```
if (rA || rB || rC || rD) {
    /* if-block instructions */
}
```

```
cmp.ne p1,p0=r0,r0 ;; // p1=0
cmp.ne.or p1,p0=rA,r0
cmp.ne.or p1,p0=rB,r0
cmp.ne.or p1,p0=rC,r0
cmp.ne.or p1,p0=rD,r0
(p1) br.cond if-block
```

```
if ((rA<0)&&(rB==-15)&&(rC>0))
    /* if-block instructions */
```

```
cmp.eq p1,p0=r0,r0 ;; // p1=1
cmp.ge.and p1,p0=rA,r0
cmp.ne.and p1,p0=rB,-15
cmp.le.and p1,p0=rC,r0
(p1) br.cond if-block
```

```
if ((rA==0) || (rB==10))
    r1=r3+r4;
else
    r7=r6-r5;
```

```
cmp.ne p1,p2=r0,r0 ;; // p1=1
cmp.eq.or.andcm p1,p2=rA,r0
cmp.eq.or.andcm p1,p2=rB,10
(p1) add r1=r3,r4
(p2) sub r7=r6,r5
```

## Déplacement de code

Vers le bas

Vers le haut

```
(p1) br.cond some_label //0
st4 [r34]=r23 //1
ld4 r5 = [r56] //1
ld4 r6 = [r57] //2
```

p2 = -p1

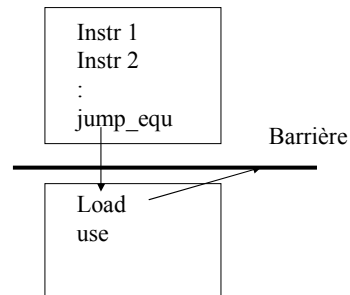
```
(p2) st4 [r34]=r23 //0
(p2) ld4 r5 = [r56] //0
(p1) br.cond some_label //0
ld4 r6 = [r57] //1
```

```
ld8 r56 = [r45] //0
st4 [r23]=r56 //2
Etiqu_A: //3
add...
add...
add... ;;
```

```
// point qui domine Label_A
cmp.ne p1,p0=r0,r0 ;; // p1=0
// autres instructions
cmp.eq p1,p0=r0,r0 ;; // p1=1
ld8 r56 = [r45] //0
Etiqu_A: //1
add...
add...
add...
(p1) st4 [r23]=r56 //2
```

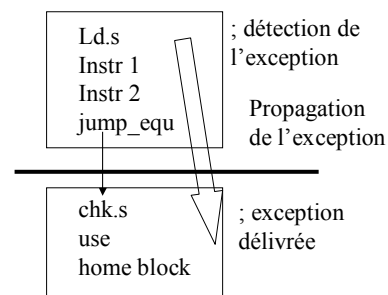
## La latence mémoire

- Les chargements influent sur la performance
  - Souvent la première instruction d'une chaîne d'instructions dépendantes
  - Peuvent provoquer des latences importantes (échecs cache)
- Les chargements peuvent provoquer des exceptions



## Les chargements spéculatifs

- Distinguer le comportement du chargement de celui de l'exception
  - L'instruction de chargement spéculatif (ld.s) débute une opération de chargement et détecte les exceptions
  - Propage un "jeton" d'exception (inclus dans le registre destination) de ld.s à chk.s
  - L'instruction de test d'exception spéculative (chk.s) provoque les exceptions détectées par ld.s



## Spéculations sur les données et instructions

---

### Récupération sur données spéculées à l'aide de ld.c

```
// autres instructions
std8 [r4]=r12
ld8 r6, [r8];;
add r5=r6, r7;;
st8 [r18]=r5
```

```
ld8.a r6, [r8];; // chargement avancé
// autres instructions
std8 [r4]=r12
ld8.c.clr r6, [r8];; //test du chargement
add r5=r6, r7;;
st8 [r18]=r5
```

ALAT: *Advanced Load Address Table*  
Entrée allouée par ld.a  
Entrée testée par ld.c

## Spéculations sur les données et instructions (2)

---

### Récupération sur données spéculées à l'aide de chk.a

```
// autres instructions
std8 [r4]=r12
ld8 r6, [r8];;
add r5=r6, r7;;
st8 [r18]=r5
```

```
ld8.a r6, [r8];;
// autres instructions
add r5=r6, r7;;
// autres instructions
std8 [r4]=r12
chk.a.clr r6, recover
Back : st8 [r18], r5
```

```
//ailleurs
Recover :
ld8 r6, [r8];;
add r5=r6, r7
br back
```

## Optimisation des boucles

---

```
L1:  ld4 r4=[r5],4 ;; // 0
      add r7=r4,r9;; // 2
      st4 [r6]=r7,4;; //3
      br.cloop L1;; //3
```

```
Stage 1 : ld4 r4=[r5],4
Stage 2 : -----
Stage 3 : add r7=r4,r9
Stage 4 : st4 [r6]=r7,4
```

### PIPELINE LOGICIEL

1	2	3	4	5	Cycle	
ld4					X	PROLOGUE
	ld4				X+1	
add		ld4			X+2	NOYAU
st4	add		ld4		X+3	
	st4	add		ld4	X+4	EPILOGUE
		st4	add		X+5	
			st4	add	X+6	
				st4	X+7	

## Renommage de registres explicite

---

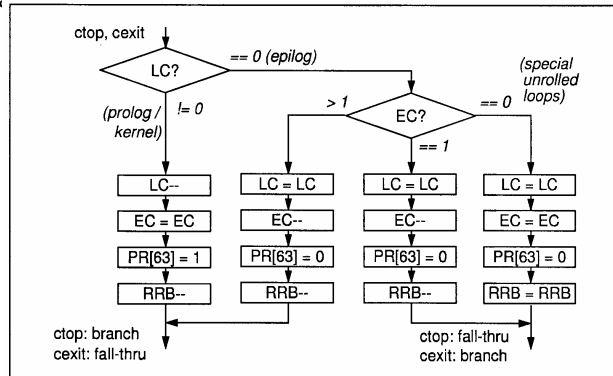
- Renommage de registres
  - N° registre = N° registre + rrb (*Rotating Register Base*)
  - rrb est décrémenté par des instructions de branchement spéciales
- p16-p63, f32-f127, et une zone programmable de registres entiers peuvent “tourner” (rotation de registres)
  - rrb.gr, rrb.fr, rrb.pr
  - Décrémentés simultanément par les branchements de boucle pour pipeline logiciel

```
L1: ld4 r35=[r4], 4
      st4 [r5]=r37, 4
      swp_branch L1;;
```

Latence de 2 cycles

## Branchements de boucle

- *Loop Count (LC)*
- *Epilog Count (EC)*



M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

57

## Exemple de boucle avec compteur

Stage 1 : (p16) ld4 r4=[r5],4  
 Stage 2 : (p17)-----  
 Stage 3 : (p18) add r7=r4,r9  
 Stage 4 : (p19) st4 [r6]=r7,4

```

mv lc=199          // LC = loop count - 1
mv ec=4           // EC= epilog stages +1
mov pr.rot =1<<16;; //PR16=1, rest=0
    
```

```

L1:
(p16) ld4 r32=[r5],4 //0
(p18) add r35=r34,r9 //0
(p19) st4 [r6]=r36,4 //0
      br.ctop L1;;
    
```

AVEC REGISTRES TOURNANTS

M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

58



## Exemple de boucle *While*

```
L1: ld4 r4=[r5],4 ;; // 0
    st4 [r6]=r4,4 //2
    cmp.ne p1,p0=r4,r0 //2
    (p1)br L1;; //2
```

```
Stage 1 : ld4 r4=[r5],4
Stage 2 : -----
Stage 3 : st4 [r6]=r4,4
          cmp.ne p1,p0=r4,r0
          (p1)br L1
```

### PIPELINE LOGICIEL

1	2	3	4	5	Cycle
ld4					X
	ld4.s				X+1
scb		ld4.s			X+2
	scb		ld4.s		X+3
		scb		ld4.s	X+4
			scb		X+5
				scb	X+6

## Exemple de boucle *While* (2)

```
mv ec=2 // EC= epilog stages +1
mov pr.rot =1<<16;; //PR16=1, rest=0
```

```
L1: ld4.s r32=[r5], 4 //0
    (p18) chk.s r34, recovery //0
    (p18) cmp.ne p17, p0=r34,r0 //0
    (p18) st4 [r6]=r34,4
    (p17) br.wtop.sptk L1;; //0
```

1	2	3	4	Cycle
ld4				X
	ld4.s			X+1
scb		ld4.s		X+2
	scb		ld4.s	X+3
		scb		X+4
			scb	X+5

## Exemple de boucle *While* (3)

Cycle	Port/instructions					State before br.wtop			
	M	I	I	M	B	p16	p17	p18	EC
0	ld4.s				br.wtop	1	0	0	2
1	ld4.s				br.wtop	0	1	0	1
2	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
3	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
...	...	...	...	...	...	...	...	...	...
100	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
...	...	...	...	...	...	...	...	...	...
199	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
200	ld4.s	cmp	chk	st4	br.wtop	0	1	1	1
201	ld4.s	cmp	chk	st4	br.wtop	0	0	1	1
						0	0	0	0

M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

63

## Exemples

```
for (i=0; i<N;i++)
  A[i]=B[i]+C[i];
```

```
L1: ldf f5=[r5], 8
     ldf f6=[r6], 8
     fadd f7=f5,f6
     stf [r7]=f7,8
     br.cloop L1
```

	p16	p17	p18	p19	p20	p21	p22	p23	p24
		1	3	5	7	9	11	13	15
						fadd	fadd	fadd	fadd
									stf
0	2	4	6	8	10	12	14	16	
ldf	ldf	ldf	ldf	ldf	ldf	ldf	ldf	ldf	ldf
ldf	ldf	ldf	ldf	ldf	ldf	ldf	ldf	ldf	ldf

Latences

ldf : 9 cycles  
fadd : 5 cycles

2 cycles/itération  $\Pi=2$

```
L1 : (p24) stf [r7]=f60,8
      (p21) fadd f57=f37,f47 ;
      (p16) ldf f32=[r5],8
      (p16) ldf f42=[r6],8
      br.ctop L1;;
```

M2 SETI  
2005-2006

Architectures avancées  
Daniel Etiemble

64

## Exemple (2)

```
for (i=0; i<N;i+=2){
  A[i]=B[i]+C[i];
  A[i+1]=B[i+1]+C[i+1];}
```

Déroulage d'ordre 2

```
L1 : (p16) ldfp f32,f33=[r5],16
      (p16) ldfp f34,f35=[r6], 16;
      (p20) fadd f40=f36,f38
      (p20) fadd f41= f37,39;;
      (p22) stf [r7] = f42, 8
      (p22) stf [r7] = f43,8
      br.ctop L1;;
```

p16	p17	p18	p19	p20	p21	p22	
	1	4	7	10	13	16	19
						stf	
						stf	
	2	5	8	11	14	17	20
				fadd	fadd	fadd	fadd
				fadd	fadd	fadd	fadd
0	3	6	9	12	15	18	21
ldfp	ldfp	ldfp	ldfp	ldfp	ldfp	ldfp	ldfp
ldfp	ldfp	ldfp	ldfp	ldfp	ldfp	ldfp	ldfp

$\Pi=3$

3 cycles/2 itérations