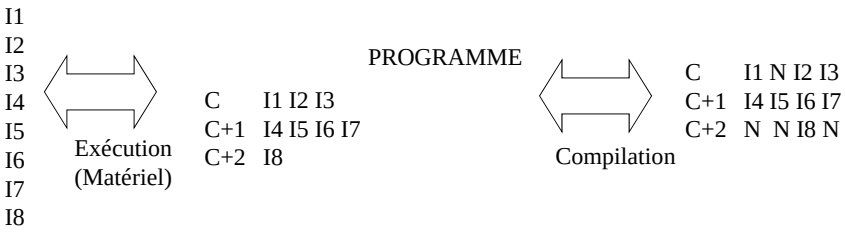
Parallélisme d'instructions : Superscalaires versus VLIW

Daniel Etiemble
de@lri.fr

Parallélisme d'instructions

- Pipeline simple et superpipeline
  - 1 instruction par cycle. $CPI_{max}=1$
- Amélioration de la performance
  - Exécuter plusieurs instructions en parallèle
    - IPC = 1/CPI (Instructions par cycle)
- Deux approches
  - Superscalaires
  - VLIW (mots d'instructions très long)

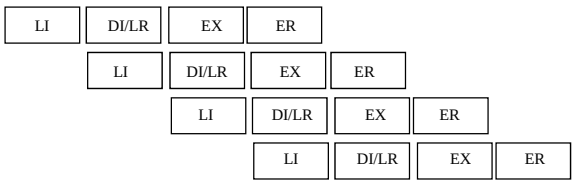
Processeurs : superscalaire/VLIW



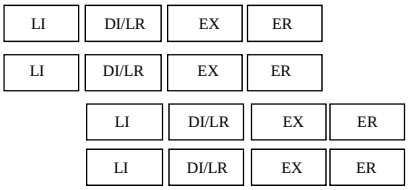
- Superscalaire
  - Le matériel est responsable à l'exécution du lancement parallèle des instructions
    - Contrôle des dépendances de données et de contrôle par matériel
- VLIW
  - Le compilateur est responsable de présenter au matériel des instructions exécutables en parallèle
    - Contrôle des dépendances 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
    - Tableau de marques (Scoreboard)
    - Tomasulo
- Existe à la fois pour les processeurs scalaires (à cause des instructions multi-cycles) et les processeurs superscalaires (problèmes accentués)

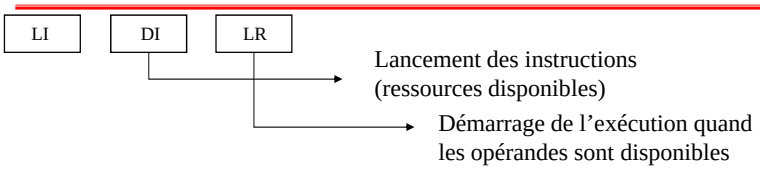
Problèmes liés aux superscalaires

- 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.

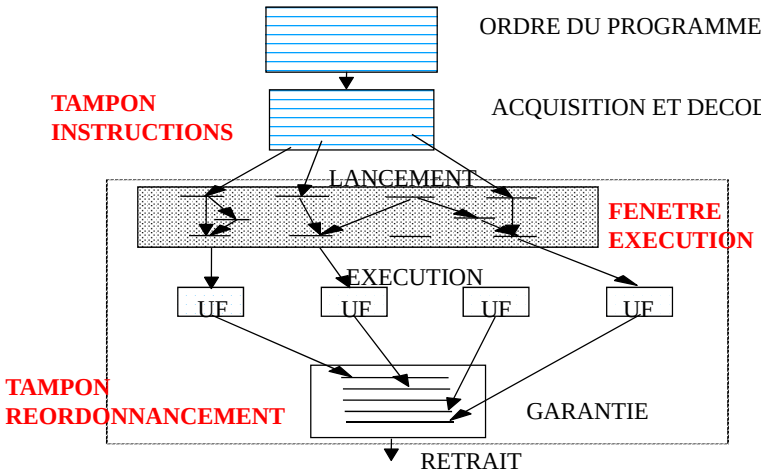
Exécution dans l'ordre (21164)

- 4 instructions acquises par cycle
- 4 pipelines indépendants
  - 2 entiers
  - 2 flottants

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

- 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



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

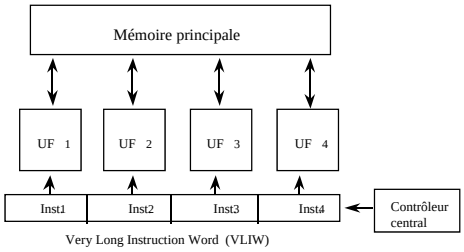
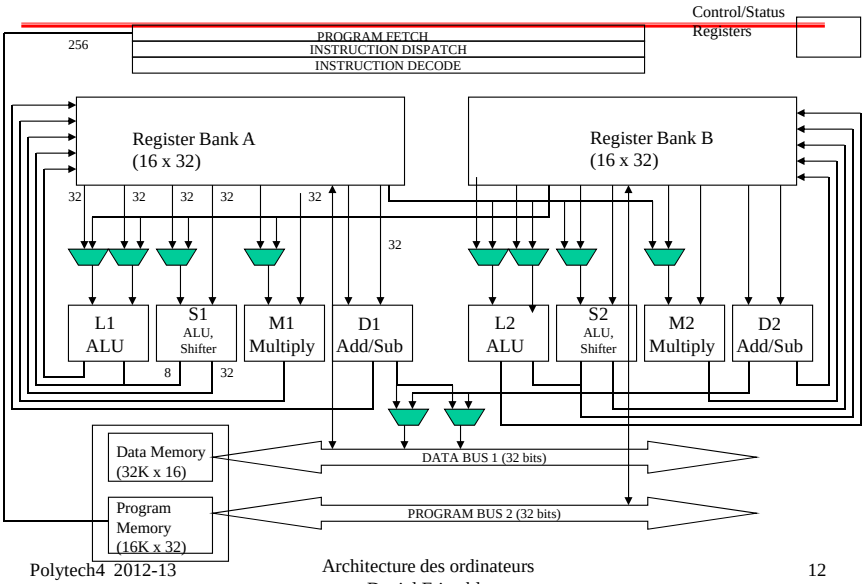


Schéma fonctionnel du C6x



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

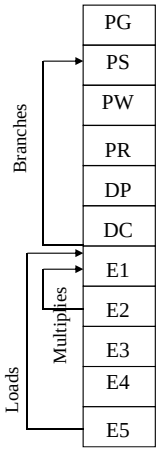
Modes d’adressage

Type d’adressage	No modification	Préincrement ou Décrement du registre adresse	Postincrément ou Décrement 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--

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



Generate Program Address
Program Address sent
Program Memory Access
Fetch reaches CPU boundary
Instruction dispatch
Instruction decode
Execute 1
Execute 2
Execute 3
Execute 4
Execute 5

Délai de branchement : 5 cycles
Délai de chargement : 4 cycles
Délai de multiplication : 1

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)

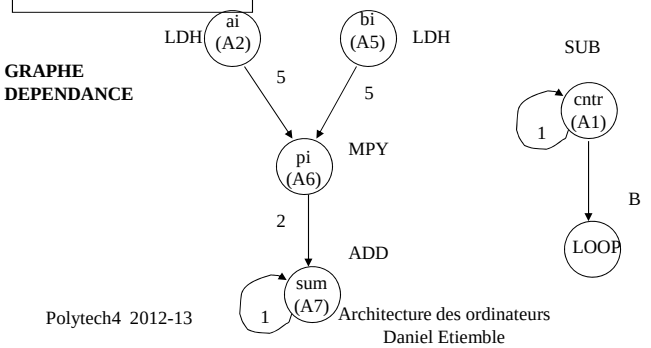
Code C

```
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



Programmation C6x : produit scalaire (2)

CODE ASSEMBLEUR NON PARALLELE

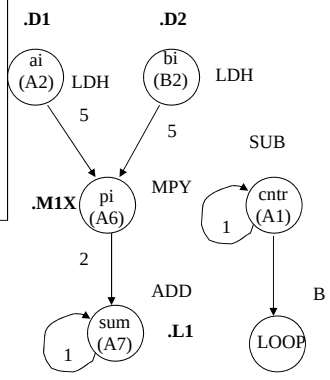
```
            MVK     .S1     100,A1      ; set up loop counter
            ZERO    .L1     A7          ; zero out accumulator
LOOP
            LDH     .D1     *A4++,A2    ; load ai
            LDH     .D1     *A3++,A5    ; load ai
            NOP     4                   ; delay slots for LDH
            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

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/itération



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
```


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/2 itérations

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 /itération

PROLOGUE
EPILOGUE