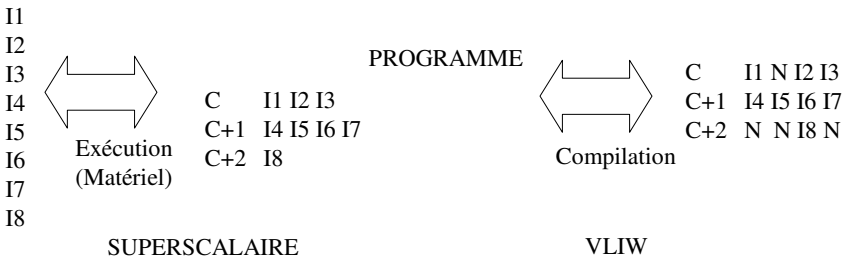
Parallélisme d'instructions : Superscalaires versus VLIW

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

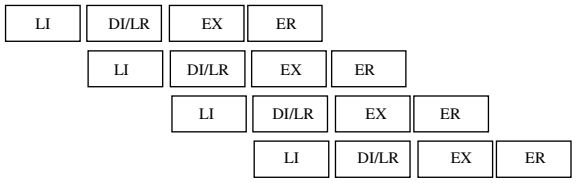
Processeurs : superscalaire/VLIW



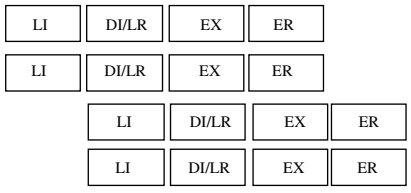
- Superscalaire
  - Le matériel est responsable à l'exécution du lancement parallèle des instructions
    - Contrôle des dépendances de données et de contrôle par matériel
- VLIW
  - Le compilateur est responsable de présenter au matériel des instructions exécutables en parallèle
    - Contrôle des dépendances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

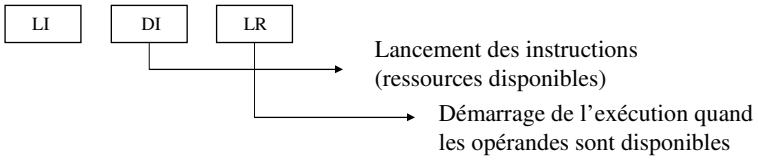
Problèmes liés aux superscalaires

- 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 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

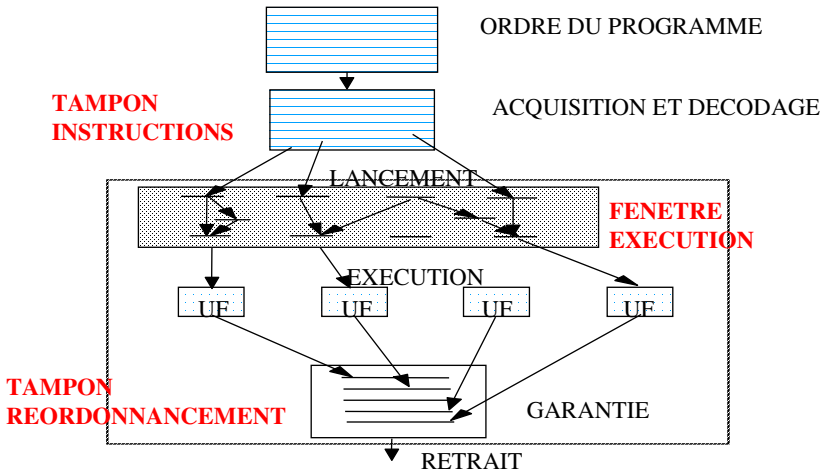
Exécution dans l'ordre (21164)

- 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

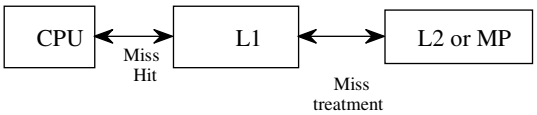
- 1 groupe de 4 instructions considéré à chaque cycle
  - si les 4 instructions se répartissent dans E0, E1, FA et FM, elles démarrent leur exécution. Sinon, certaines utilisent les cycles suivants jusqu'à ce que les 4 aient démarré.
  - Les 4 suivantes sont traitées quand les 4 précédentes ont démarré.

Exécution non ordonnée



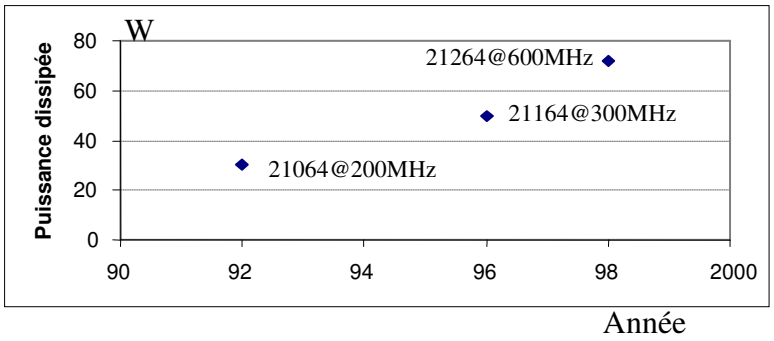
Caches non bloquants

- 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)



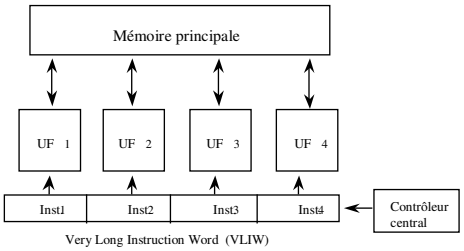
du R5000 au R10000

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 mm2	84 mm2	
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%

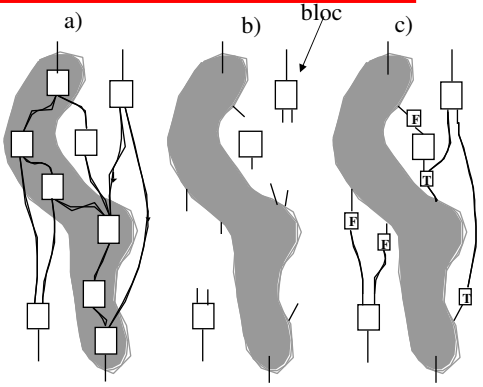
ARCHITECTURE VLIW

- PRINCIPE
  - Plusieurs unités fonctionnelles
  - ORDONNANCEMENT STATIQUE (compilateur)
    - Aléas de données
    - Aléas de contrôle
  - Exécution pipelinée des instructions
  - Exécution parallèle dans N unités fonctionnelles
- Taches 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



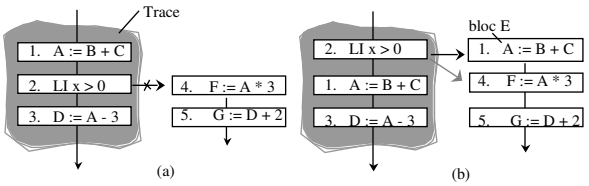
VLIW: "Ordonnancement de traces"

- 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

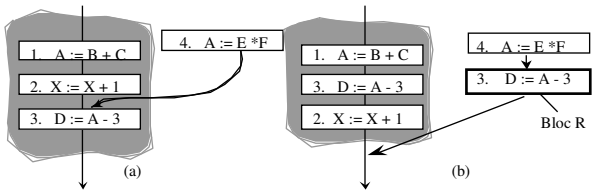


VLIW: “Ordonnancement de traces”

Blocs sortant de la trace

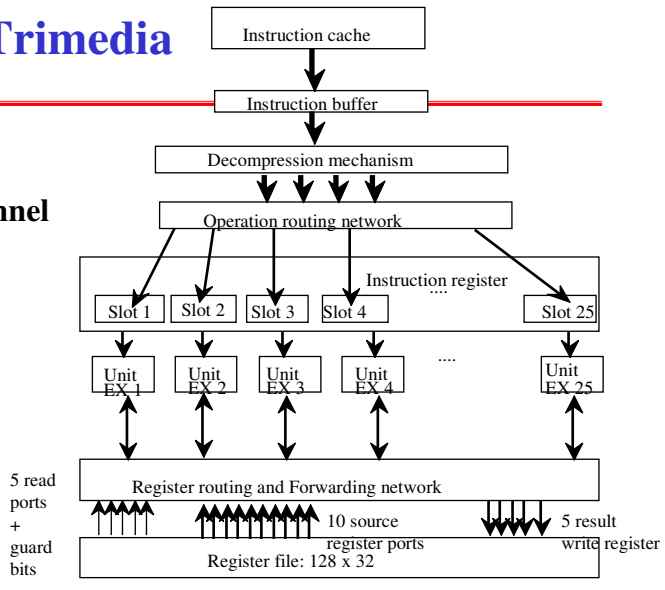


Blocs entrant dans la trace



VLIW - Trimedia
(Philips)

Schéma fonctionnel

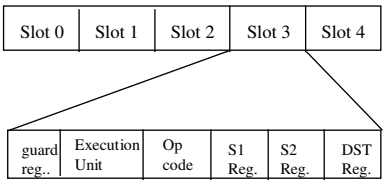


Jeu d'instructions Trimédia

- 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)

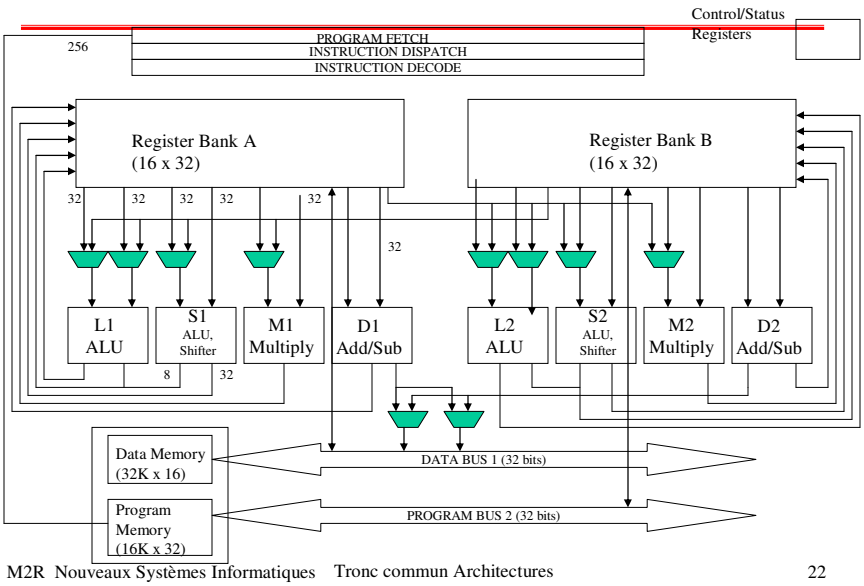
- Format d'instructions
  - Registres de garde : indiquent si l'opération est exécutée (opérations conditionnelles)
  - Techniques de compression
    - Supprimer les NOP des opérations non utilisées
  - Branchements retardés
    - 3 cycles
- Objectifs
  - VLIW + DSP
  - Multimédia



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



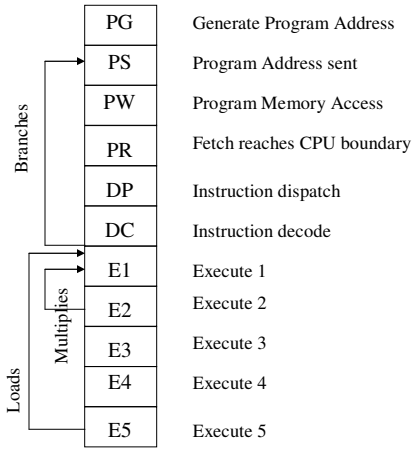
Modes d'adressage

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		

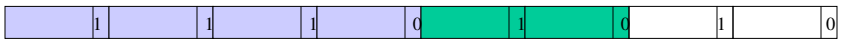
PIPELINE C6x



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

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)

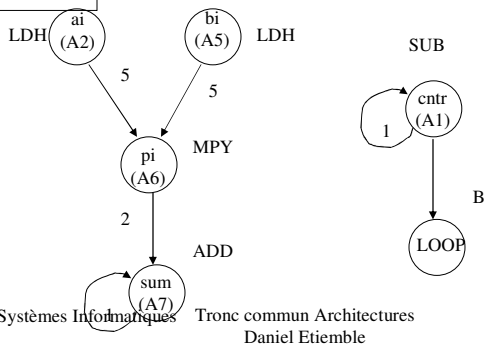
Code C

```
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



Programmation C6x : produit scalaire (2)

CODE ASSEMBLEUR NON PARALLELE

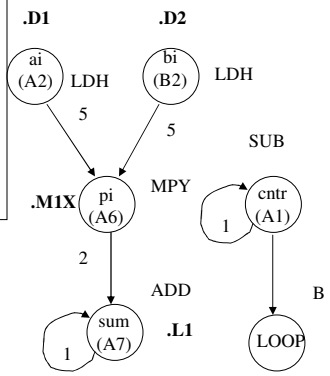
```
            MVK     .S1     100,A1      ; set up loop counter
            ZERO    .L1     A7          ; zero out accumulator
LOOP
            LDH     .D1     *A4++,A2    ; load ai
            LDH     .D1     *A3++,A5    ; load ai
            NOP     4                   ; delay slots for LDH
    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

Programmation C6x: si – alors - sinon

Code C

```
int if_then(short[a], short theta,
int codeword, int mask)
{
int sum, i, cond;
sum=0;
for (i=0; i<32;i++)
cond= codeword&mask
if (theta==!(!(cond)))
                sum+=a[i];
else
                sum-=a[i];
mask=mask<<1;
}
return (sum);
}
```

Code assembleur

```
                MVK     32, cntr
                ZERO    sum
LOOP:
                AND     .SX2    cword, mask, cond
[cond]          MVK     .S2     1,cond
                CMPEQ   .L2     theta, cond, if
                LDH     .D1     *a++,ai
[if]            ADD     .L1      sum, ai, sum
[!if]           SUB     .D1      sum, ai, sum
                SHL     .S1     mask,1,mask
[cntr]          ADD     .L2     -1,cntr,cntr
[cntr]          B       .S1     LOOP
```


Si – alors - sinon : Table des intervalles entre itérations

Unit/cy	0	2	4	6	8	10	12	14
.D1				SUB ai	SUB ai	SUB ai	SUB ai	SUB ai
.D2								
.M1								
.M2								
.L1			ZERO	ADD ai	ADD ai	ADD ai	ADD ai	ADD ai
.L2	ADD cntr		CMPEQ	CMPEQ	CMPEQ	CMPEQ	CMPEQ	CMPEQ
.S1		SHL	SHL	SHL	SHL	SHL	SHL	SHL
.S2		AND	AND	AND	AND	AND	AND	AND
Unit/cy	1	3	5	7	9	11	13	15
.D1	LDH	LDH	LDH	LDH	LDH	LDH	LDH	LDH
.D2								
.M1								
.M2								
.L1								
.L2	ADD cntr	ADD cntr	ADD cntr	ADD cntr	ADD cntr	ADD cntr	ADD cntr	ADD cntr
.S1	B Loop	B Loop	B Loop	B Loop	B Loop	B Loop	B Loop	B Loop
.S2		MVK	MVK	MVK	MVK	MVK	MVK	MVK

Intervalle entre itérations =2

Programmation C6x: Si – alors - sinon
(pipeline logiciel)

```
LOOP:
[B0]      ADD     .L2     -1,B0,B0
|| [B2]   MVK     .S2     1,B2
|| B0]    B       .S1     LOOP
||        LDH     .D1     *A4++,A7

[B1]      ADD     .L1      A7, A5, A7
|| [!B1]  SUB     .D1      A7, A5, A7
||        CMPEQ   .L2      B6, B2, B1
||        SHL     .S1      A6,1,A6
||        AND     .SX2     B4, A6, B2
```

2 cycles/itération

IA-64 (EPIC)

- Voir « superscalaires versus VLIW » dans le cours M1 Architectures avancées

http://www.lri.fr/~de/ArchiM1-1011.htm