Jeux d'instructions

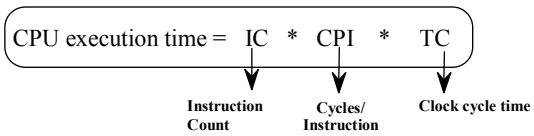
Daniel Etiemble
de@lri.fr

Plan

- Jeux d'instructions
  - Impact performance
  - Impact taille du code
- Pipelines
  - Simple, superpipeline et superscalaire : impact sur la performance
  - Impact des branchements
    - Prédicteurs
    - Alternative : instructions conditionnelles et prédicats (ARM, IA-64)
- Instructions SIMD
- Hiérarchie mémoire
  - Caches et impact sur la performance
    - Monoprocesseur
    - Cohérence des caches dans les multiprocesseurs
  - Alternatives aux caches
    - Processeurs vectoriels
    - Mémoire scratch-pad
- DSP et microcontrôleurs
- Graphique
  - Cartes graphiques
  - Processeurs graphiques et processeurs multimédia

PERFORMANCE

- L'équation



- Les composantes de la performance
  - Nombre d'instructions
    - Jeu d'instructions et compilateur
  - CPI
    - Microarchitecture
  - T_c
    - Technologie CMOS et Microarchitecture

L'exécution d'une instruction

- Les étapes fondamentales

Instructions UAL	*Instructions Mémoire*	*Instructions Branchement*
Lecture instruction	Lecture instruction	Lecture instruction
Incrémentation CP	Incrémentation CP	Incrémentation CP
Décodage de l'instruction	Décodage de l'instruction	Décodage de l'instruction
Lecture des opérandes	Calcul de l'adresse mémoire	Calcul de l'adresse de branchement
Exécution	Accès mémoire	Exécution
Ecriture du résultat	Rangement du résultat	

Les différentes étapes

- Instructions entières
  LI/CP DI/LR EX ER
- Instructions flottantes
  LI/CP DI/LR EX1 EX2 ... ER
- Instructions mémoire
  LI/CP DI/LR CA AM ER
- Instructions de branchement
  LI/CP DI/CAB/EX

Les objectifs

- Performance
  - Pipeline efficace
    - Instructions de longueur fixe
    - Décodage simple
    - Modes d'adressage simples
- Taille du code
  - Minimiser la taille des instructions
    - Instructions de longueur variable (ou fixe)
  - Accès aux données efficace
    - Modes d'adressage complexes et efficaces pour applications visées
- Compatibilité binaire avec les générations précédentes
  - Exemple IA-32 (x86)

Modèles d'exécution

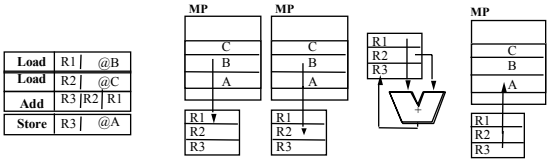
- Modèles d'exécution (n,m)
  - n : nombre d'opérandes par instruction
  - m : nombre d'opérandes mémoire par instruction
- Les différents modes
  - RISC : (3,0)
    - Instructions de longueur fixe
    - Load et Store : seules instructions mémoire
  - IA-32 : (2,1)
  - Pile (0,0)
    - Tous les opérandes sont accédés via la pile

Modèle d'exécution RISC

(n,m)

n : nombre d'opérandes par instruction
m : nombre d'opérandes mémoire par instruction
Ex : A : = B + C

LOAD-STORE (3,0)



Instructions de longueur fixe
Seules les instructions Load et Store accèdent à la mémoire

Registres: organisation RISC

- 32 registres généraux (entiers) R0 à R31
- 32 registres flottants
- Instructions UAL et mémoire
  - Registre – registre
    - $R_d \leftarrow R_{s1}$ op R_{s2}
  - Registre – immédiat
    - $R_d \leftarrow R_{s1}$ op immédiat
    - Rd ↔ Mémoire (R_{s1} + dépl.)

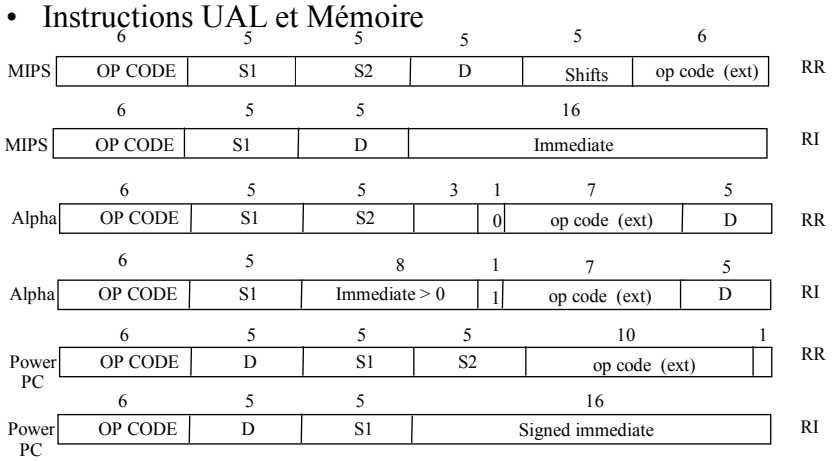
General Purpose Registers

R0	always zero	local var A	R16
R1		local var B	R17
R2		local var C	R18
R3		local var D	R19
R4		local var E	R20
R5			R21
R6			R22
R7			R23
R8	compiler temp 1		R24
R9	compiler temp 2		R25
R10	compiler temp 3		R26
R11	compiler temp 4		R27
R12			R28
R13		stack pointer	R29
R14			R30
R15			R31

Typical RISC Processor

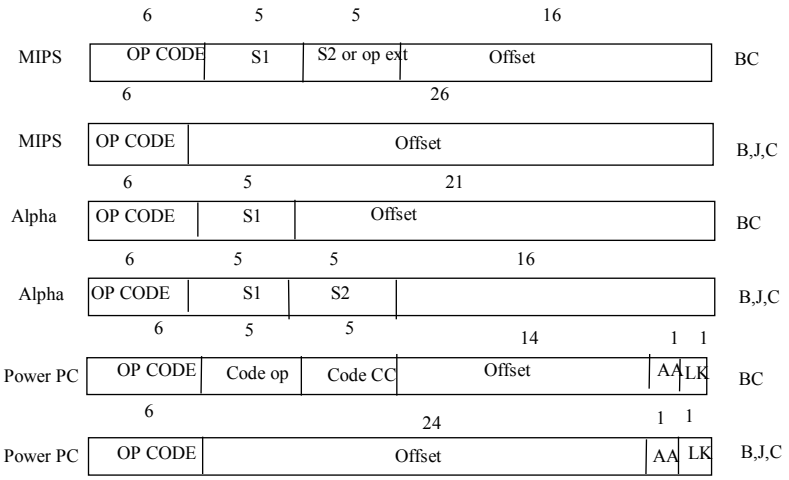
Formats d'instructions RISC

- Instructions UAL et Mémoire



Formats d'instructions RISC

- Sauts et branchements



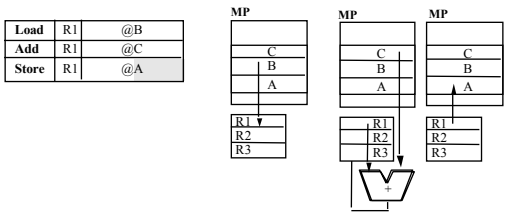
Modèle (2,1)

(n,m)

n : nombre d'opérandes par instruction
m : nombre d'opérandes mémoire par instruction

Ex : A : = B + C

REGISTRE-MEMOIRE (2,1)



CISC compatible avec la technologie MOS des années 75-80

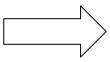
Caractéristiques IA-32

- Instructions de longueur variable

Op code	Reg. et M	Déplacement	Immédiat	
1 or 2	1 or 2	0, 1, 2 or 4	0, 1, 2 or 4	octets

  – Inst dest, source

REG	**REG**
REG	**MEM**
REG	**IMM**
MEM	**REG**
MEM	**IMM**



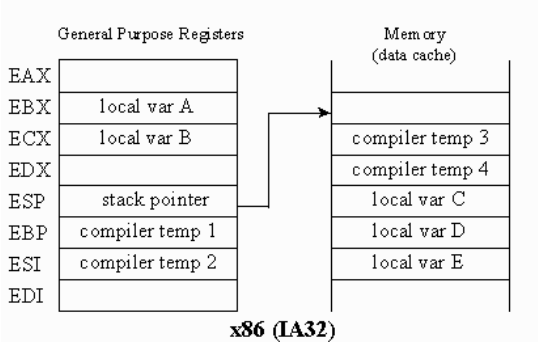
Lecture mémoire,
Exécution,
Ecriture mémoire

- Instructions complexes
  – Rep
- Modes d'adressage complexes

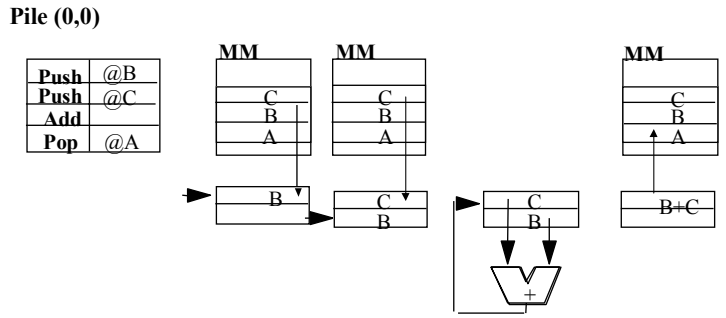
Adresse mémoire = Rb + RI x f + déplacement

Registres : organisation IA-32

- Organisation non homogène
  – 8 registres «généraux» avec rôle spécifique
  – Registres flottants fonctionnant en pile (x87)
  – Registres «SIMD» MMX, SSE-SSE2-SSE3)

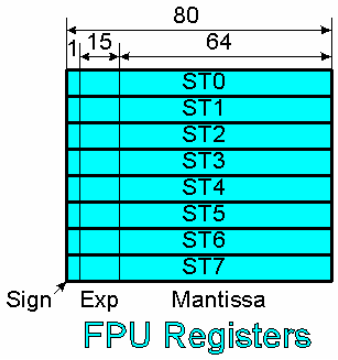


Modèle pile (0,0)



- Utilisé dans certains CPU (Transputer)
- Pile flottante x87

Pile flottante x87



- **Matériel pour calcul rapide sur nombres flottants**
  - Opérations flottantes en parallèle avec les opérations entières
  - Historiquement implanté comme un coprocesseur (8087-80387)
  - CPU et FPU échangent les données via la mémoire
- **8 registres spécifiques 80 bits (précision étendue)**

Instructions flottantes IA-32 (x87)

- **Exemples de codes opérations flottants 80x87 FPU**
  - FADD: Addition
  - FMUL: Multiplication
  - FDIV: Division
  - FDIVR: Division
  - FSIN: Sinus (radians)
  - FCOS: Cosinus (radians)
  - FSQRT: Racine carrée
  - FSUB: Soustraction
  - FABS: Valeur absolue
  - FYL2X: Calcule Y * $\log_2(X)$
  - FYL2XP1: Calcule Y * $\log_2(X)+1$

Programmer la pile

- ST=ST(0)=sommet de pile
  Utilisation de ST(0) est implicite si non spécifié
- Restriction : un registre doit être ST(0)
- **FLD MemVar**: Charge ST(0) avec MemVar & Empile ST(i) dans ST(i+1) pour i=0..6
- **FSTP MemVar**: Sommet de pile en mémoire & Empile ST(i) dans ST(i-1) pour i=1..7
- **FST MemVar**: Sommet de pile en mémoire. Laisse résultat en ST(0)
- **FSTP:** Range résultat et dépile

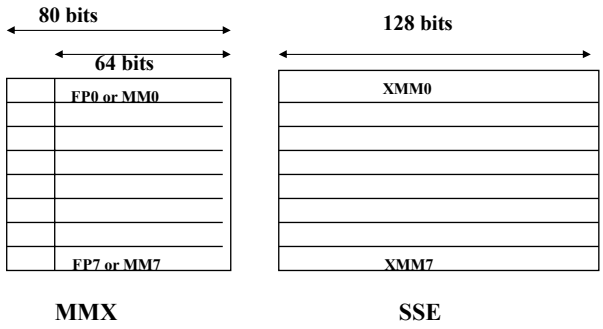
Programmer avec la pile

```
FLD VarX
FLD VarY
FLD VarZ
FMUL
FADD
FSTP MRslt
```

Programmer avec des registres

```
FLD VarX
FLD VarY
FMUL ST(1),ST(0)
FSIN ST(0)
```

Registres MMX, SSE et SSE2 du Pentium 4



- Les registres MMX sont partagés avec les registres flottants
- Les registres XMM sont séparés

Le débat RISC-CISC pour les PC

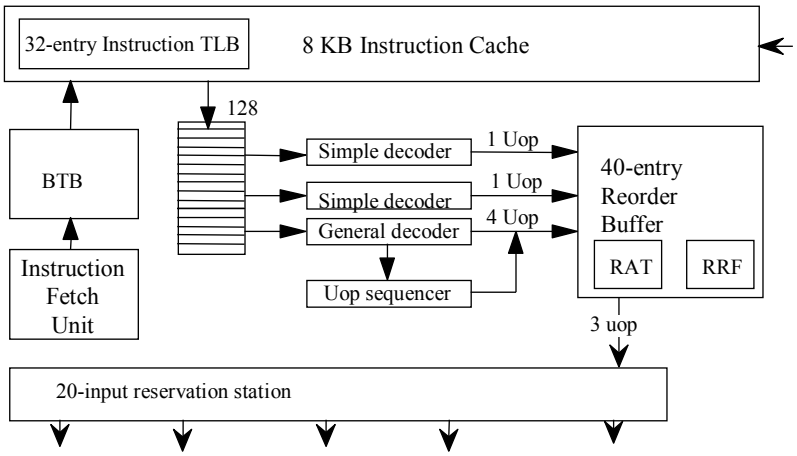
- Définition
  - RISC : modèle (3,0)
  - CISC : tous les autres
- RISC et pipeline
  - Les jeux d'instructions RISC facilitent la réalisation de pipelines performants
- « Solution » Intel et AMD pour IA-32
  - Convertir les instructions CISC en instructions RISC lors du décodage des instructions (conversion dynamique)
    - On conserve la compatibilité binaire
    - On a l'efficacité des pipelines « RISC »

Traduction des instructions x86

Pentium Pro, PII, PIII, P4

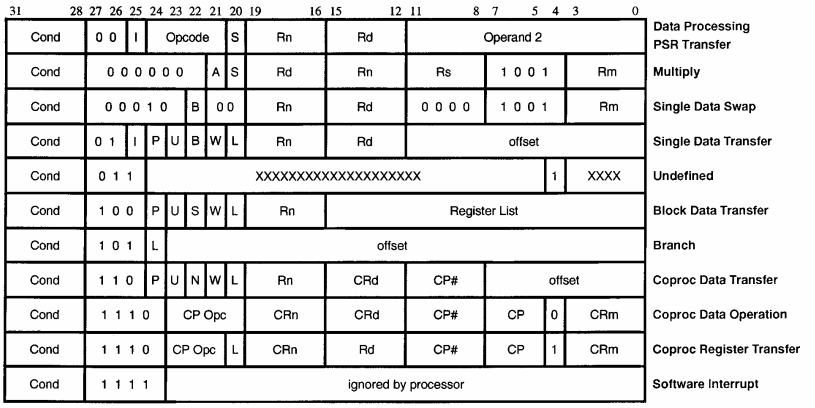
Instructions x86	**Opérations RISC**
add EAX, [EBP +d8]	load temp, [EBP + d8] add EAX, temp
add [EBP +d8], EAX	load temp, [EBP + d8] add EAX, temp store EAX, [EBP+8]
cmp EAX, imm32	cmp EAX, imm32
push ECX	sub ESP, 4 store [ESP], ECX

Les décodeurs du Pentium Pro



Le jeu d'instruction ARM

JEU D 'INSTRUCTIONS RISC 32 BITS



ARM ISA et densité du code

- Toutes les instructions sont conditionnelles
  - suppression de branchement
- Modes d 'adressage complexes
  - registre + déplacement 8 bits (avec rotation)
  - registre + registre avec décalage
  - pré ou post indexation
- Transfert de blocs de registres
- Combinaison d 'opérations UAL avec décalage/rotation sur le deuxième opérande
  - multiplication par constante

ARM : Modèle de programmation

- 31 registres 32 bits + 6 registres d'état
  - 16 registres R0 à R15 visible par le programmeur
    - R15 = CP
    - R14 = Registre de lien (retour de procédure)
    - R13 = Pointeur de pile
  - Autres registres utilisés pour interruptions, système, etc
- Format général des instructions
  - si condition Rd ⇐ Rn OP Shift/Rotate (Operand 2)

Réduction de la taille de code

- MIPS16
  - Version 16 bits du code MIPS32
- Thumb
  - Définition d 'instructions 16 bits compatibles avec les instructions ARM 32 bits
    - suppression de la condition
    - 8 registres (R0 à R7) au lieu de 16
    - Immédiats sur 8 bits au lieu de 12 bits
    - Format à 2 opérandes au lieu de 3 opérandes
  - Compression au niveau procédures
  - Exécuter un code 16 bits sur un processeur 32 bits
    - Traduction au décodage Thumb vers ARM

ARM et THUMB

Simple C routine

Here is a simple C routine that demonstrates the différences between Thumb and ARM code. This routine returns the absolute value of the C integer passed to it as a parameter. The C code is:

```
 if (X>=0)
            return x;
else
            return -x;
```

Codes	Instructions	Taille (Octets)
ARM	3	12
Thumb	4	8

The equivalent ARM assembly version is (excluding preamble):

```
iabs   CMP   ro,#o       ;Compare r0 to zero
       RSBLT r0,r0,#0    ;If r0<0 (less than-LT)
                           then r0= 0-r0
       MOV   pc,lr       ;Move Link Register,to PC
                           (Return)
```

The Thumb assembly version is:

```
       CODE16          ;Directive specifying 16-bit
                         (Thumb)
iabs   CMP r0, #0        ; Compare r0 to zero
       BGE return;      Jump to Return îf greater or
                         ;equal to zero
       NEG      r0, r0 ;If not, negate r0
return MOV      pc,lr ; Move Link register to PC
                         (Return)
```


Codes ARM et Thumb

CONVERSION BINAIRE HEXADECIMAL

ARM code:

```
       MOV   r1,r0          ;r0 holds value to convert
                            ;Load r1 with value
       MOV   r2,#8          ;Load r2 with decimal 8
Loop:  MOV   r1,r1,ROR #28  ;Rotate r1 right by 28 and
                            ;put result in r1
       AND   r0,r1,#15      ;AND r1 with decimal 15
       CMP   r0,#10         ;Compare r0 with decimal 10 and
       ADDLT r0,r0,#'0'     ;if it's less than 10 then
                            ;ADD ASCII 0 to r0
       ADDGE r0,r0,#'A'     ;otherwise (greater or equal)
                            ;ADD ASCII A to r0
       SWI 0                ;routine to write char to scree
       SUBS  r2,r2,#1       ;Subtract 1 from r2
       BGT   Loop           ;and loop back to 1 if r2 is
                            ;still greater than zero
       MOV   pc,lr          ;Load PC with Link reg (Return)
```

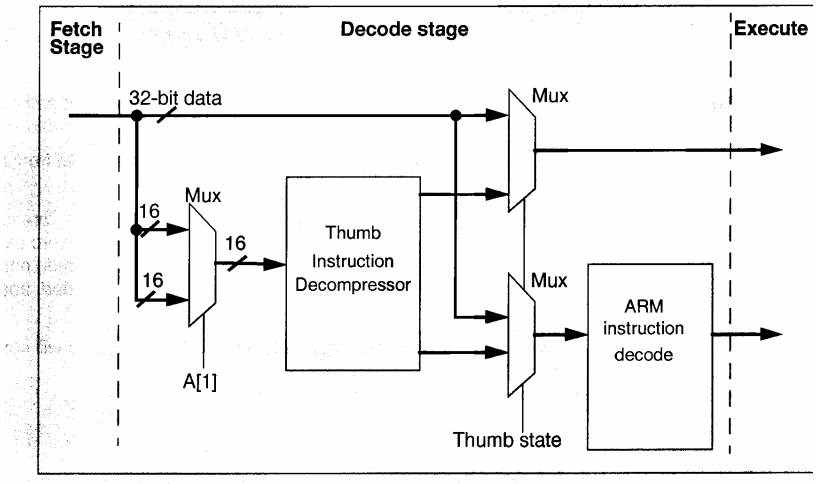
Thumb code:

```
       MOV   r1,r0          ;Value to convert in r0
                            ;Load r1 with value
       MOV   r2,#8          ;Put 8 in r2
Loop1  LSR   r0,r1,#28      ;Do logical shift right on r1
                            ;by 28 places and place in r0
       LSL   r1,r1,#4       ;Do logical shift left on r1
                            ;by 4 places
       CMP   r0,#10         ;Compare r0 with 10 and
       BLT   Loop2          ;if less than 10, branch to Loop2
       ADD   r0,#'A'-'0'-10;ADD ASCII A-0-10 (7) to r0
Loop2  ADD   r0,#'0'        ;ADD ASCII 0 (48) to r0
       SWI 0                ;routine to write char to screen
       SUB   r2,#1          ;subtract 1 from r2
       BNE   Loop1          ;if unfinished loop1
       MOV   pc,lr          ;else,load PC with Link register
                            ;(Return)
```

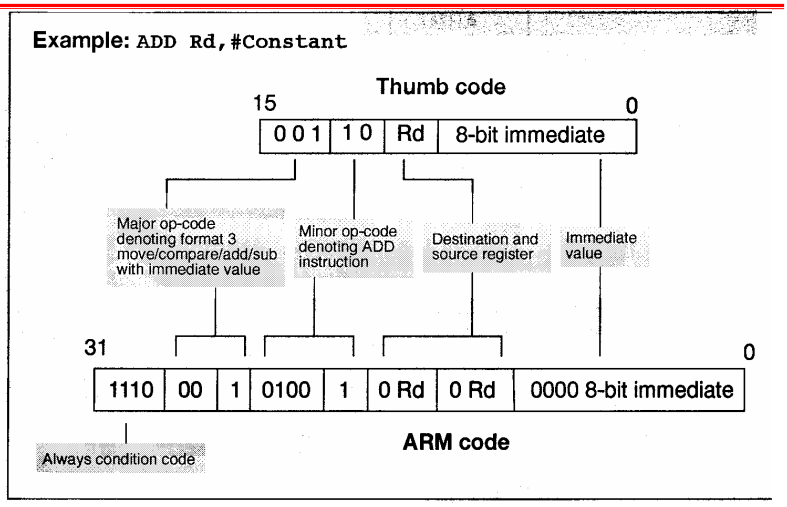
11 instructions
44 octets

12 instructions
24 octets

Décodage et décompression Thumb



Décompression Thumb vers ARM



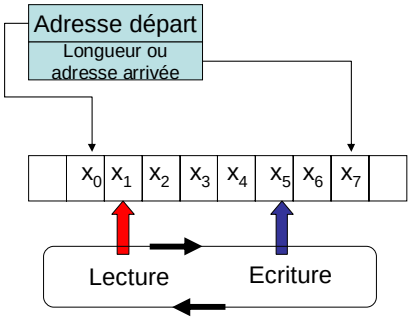
Instructions particulières liées aux applications

- Produit scalaire en calcul scientifique
  - S+= a[i]*b[i]
  - Power et PowerPC
    - Instructions Multiplication-Addition et Multiplication-soustraction (simple ou double précision)
- Produit scalaire en traitement du signal
  - Calcul des filtres FIR et IIR
  - DSP
    - Instruction Multiplication - Addition entière 16 bits vers 32 bits
    - Instruction DOTP (produit scalaire) dans le TMS 320C64x

$$y(n) = \sum_{k=0}^{M} b_k x(n-k)$$

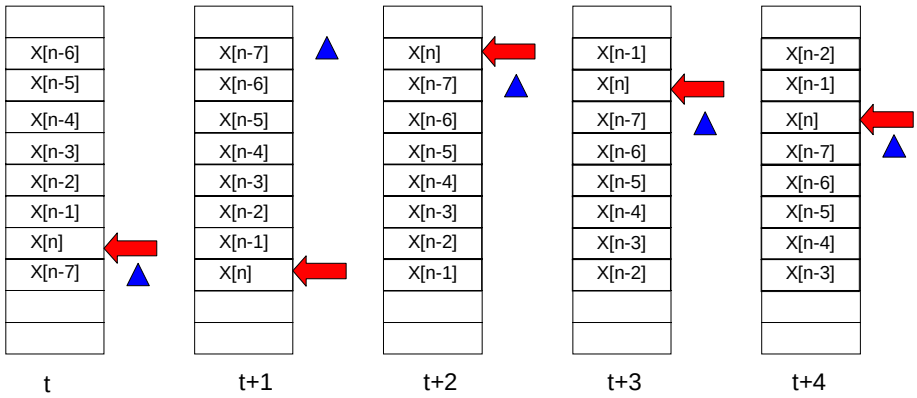
Modes d'adressage liés aux applications

- Adressage circulaire
  - But : ne conserver que les dernières valeurs des entrées ou d'un calcul précédent



Exemple d'adressage circulaire

Mémoire



: écriture donnée

: donnée la plus ancienne

ADD Ri, M (Rx) accède toujours à X[n]

Algorithme de Cooley-Tukey (DFT)

- Algorithme DFT pour une puissance de 2 $N = 2^{\nu}$

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk} = \sum_{n=0}^{N-1} x[n] e^{-j2\pi nk/N} ; W_N = e^{-j2\pi/N}$$

- Sommes séparées pour les valeurs paires et impaires de *n:*

$$X[k] = \sum_{n\ pair} x[n] W_N^{nk} + \sum_{n\ impair} x[n] W_N^{nk}$$

- Avec $n = 2r$ pour n pair et $n = 2r+1$ pour n impair

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{(2r+1)k}$$

Algorithme de Cooley-Tukey (DFT)

mais $W_N^2 = e^{-j2\pi 2/N} = e^{-j2\pi/(N/2)} = W_{N/2}$ et $W_N^{2rk} W_N^k = W_N^k W_{N/2}^{rk}$

et …

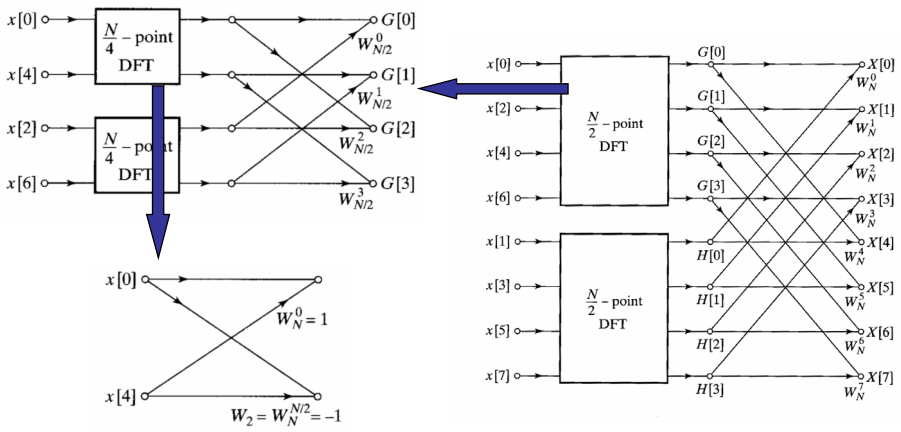
$$X[k] = \underbrace{\sum_{n=0}^{(N/2)-1} x[2r] W_{N/2}^{rk}}_{\text{DFT } N/2 \text{ points de } x[2r]} + W_N^k \underbrace{\sum_{n=0}^{(N/2)-1} x[2r+1] W_{N/2}^{rk}}_{\text{DFT } N/2 \text{ points de } x[2r+1]}$$

$$X[k] = \sum_{k=0}^{N-1} x[n] W_N^{nk}$$

$$= \sum_{n=0}^{(N/2)-1} x[2r] W_{N/2}^{rk} + W_N^k \sum_{n=0}^{(N/2)-1} x[2r+1] W_{N/2}^{rk}$$

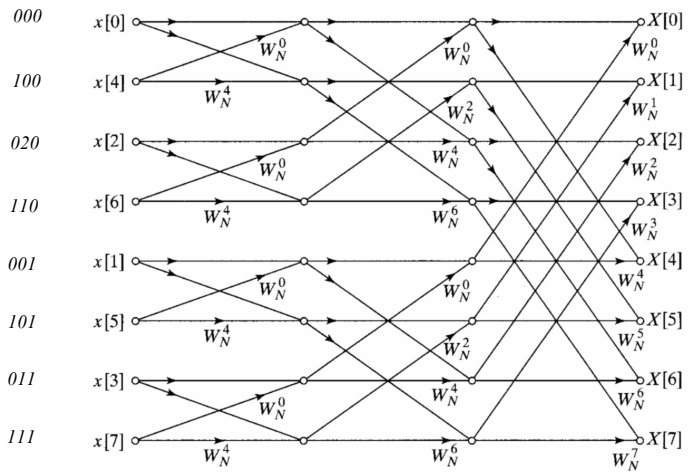
Représentation d'une DFT 8 points

$$X[k] = G[k] + W_N^k H[k]$$



La FFT complète 8 points

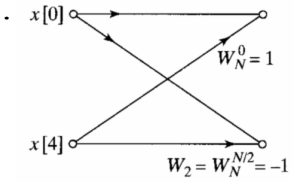
Ordre opposé des bits en entrée



La DFT 2 points

- Expression : $$X[k] = \sum_{n=0}^{1} x[n] W_2^{nk} = \sum_{n=0}^{1} x[n] e^{-j2\pi nk/2}$$
- Avec $k = 0,1$ on obtient

$$X[0] = x[0] + x[1]$$
$$X[1] = x[0] + e^{-j2\pi 1/2} x[1] = x[0] - x[1]$$



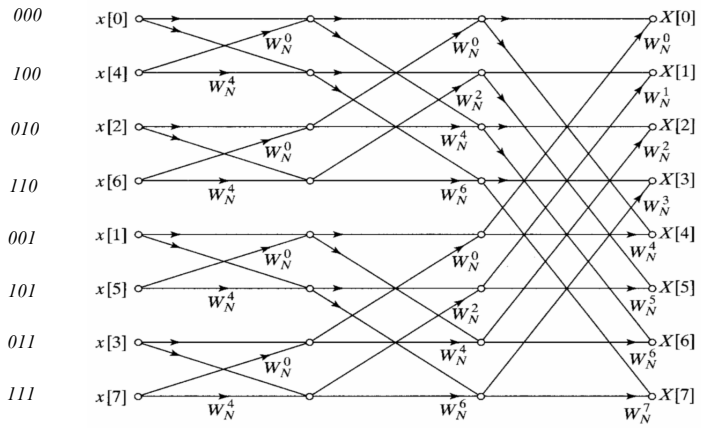
Papillon FFT

FFT : Version « Embree »

```
void fft_c(int n,COMPLEX *x,COMPLEX *w)
{COMPLEX u,temp,tm;
int i,j,le,windex;
windex = 1;
  for(le=n/2 ; le > 0 ; le/=2) {
    wptr = w;
    for (j = 0 ; j < le ; j++) {
      u = *wptr;
      for (i = j ; i < n ; i = i + 2*le) {
        xi = x + i;
        xip = xi + le;
        temp.real = xi->real + xip->real;
        temp.imag = xi->imag + xip->imag;
        tm.real = xi->real - xip->real;
        tm.imag = xi->imag - xip->imag;
       xip->real = tm.real*u.real - tm.imag*u.imag;
       xip->imag = tm.real*u.imag + tm.imag*u.real;
        *xi = temp; }
      wptr = wptr + windex;}
    windex = 2*windex;}}
```

- Embree, C algorithms for Real-Time DSP, Prentice Hall
- Caractéristiques
  - POINTEURS
  - STRUCTURE
  - PAS NON UNITAIRES

Modes d'adressage liés aux applications

- Adressage avec inversion de l'ordre des bits (FFT)

Ordre opposé des bits en entrée