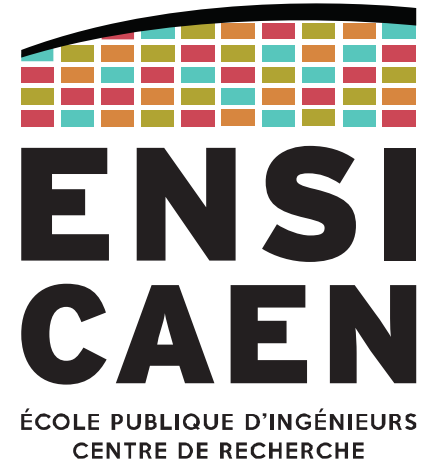


Chapitre 3
Assembleur
C6678

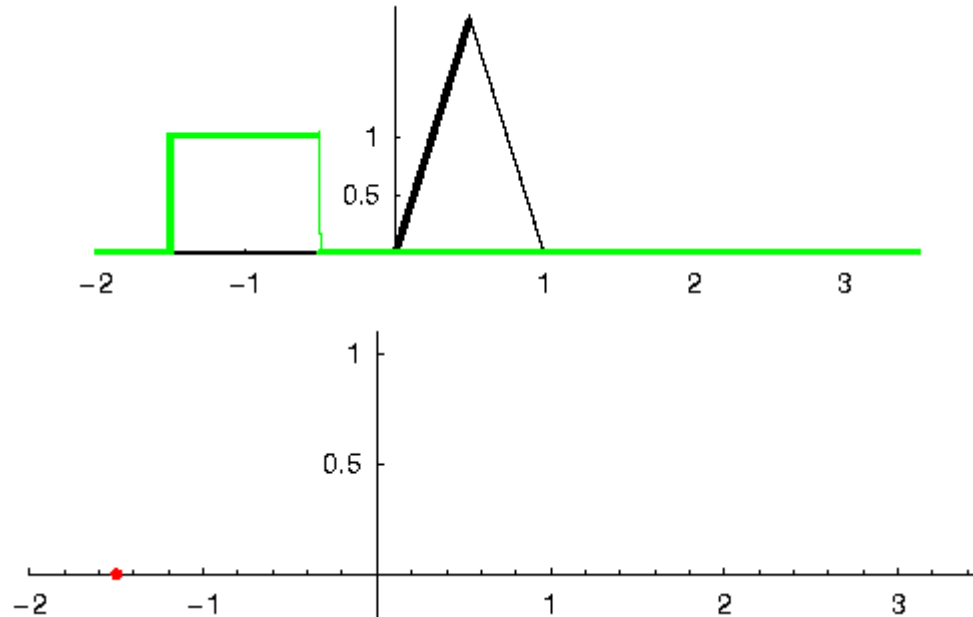


EXEMPLE D'ALGORITHME



Les TP se feront autour d'un algorithme bien connu : la **convolution discrète**.

Cet algorithme a une structure très simple, mais est très difficile à accélérer sans *refactoring* mathématique.



Voici la définition mathématique de la convolution discrète :

$$y(k) = \sum_{k=0}^Y \sum_{j=0}^N a(j) \cdot x(k-j)$$

Avec :

- $x()$ le vecteur d'échantillons en entrée
- $y()$ le vecteur d'échantillons en sortie
- $a()$ le vecteur des coefficients
- Y la taille du vecteur de sortie
- N le nombre de coefficients a
- k l'indice de l'échantillon en cours de traitement

Valider l'implémentation de l'algorithme

Avant de programmer en C le processeur cible, l'algorithme est généralement validé par prototypage et simulation, avec des outils tels que Matlab/Simulink.

Validé l'algorithme consiste à coder son implémentation canonique et vérifier les valeurs des vecteurs d'entrée et de sortie.



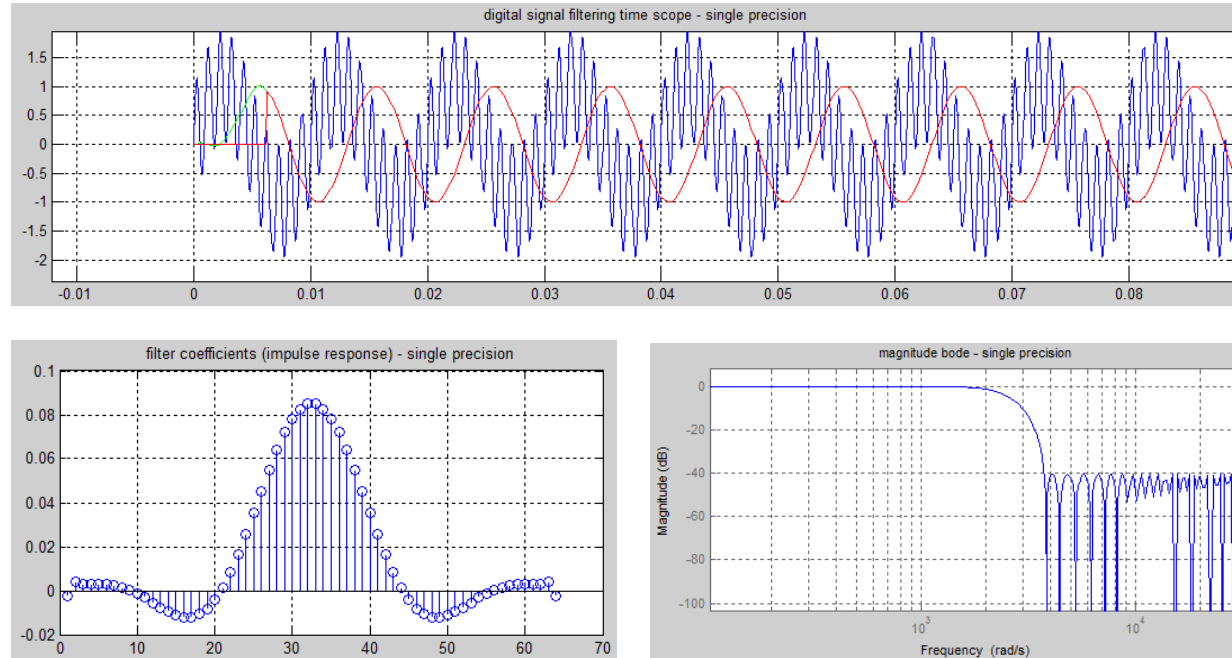
Voici donc l'implémentation Matlab de l'algorithme de convolution discrète.

```
function yk = fir_sp(xk, coeff, coeffLength, ykLength)
    yk = single(zeros(1,ykLength));    % output array preallocation

    % output array loop
    for i=2:ykLength
        yk(i) = single(0);

        % FIR filter algorithm - dot product
        for j=1:coeffLength
            yk(i) = single(yk(i)) + single(coeff(j)) * single(xk(i+j-1));
        end
    end
end
```

... et les valeurs obtenues par simulation, pour un filtre FIR d'ordre 64.



Ce code est fourni avec les ressources du TP.

Une fois l'algorithme validé, il est implémenté sur le processeur cible.

D'abord, implémentation en C canonique utilisant des **float simple-précision IEEE-754**.

```
void fir_sp (    const float * restrict xk,  \
                const float * restrict a,  \
                float * restrict yk,       \
                int na,                    \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```


Une autre version de l'algorithme en C canonique.

Celle-ci est fournie par Texas Instruments via sa librairie **dsplib**.

```
#pragma CODE_SECTION(DSPF_sp_fir_gen_cn, ".text:ansi");

#include "DSPF_sp_fir_gen_cn.h"

void DSPF_sp_fir_gen_cn(const float *x,
    const float *h,
    float *y,
    int nh,
    int ny)
{
    int i, j;
    float sum;

    for(j = 0; j < ny; j++)
    {
        sum = 0;

        // note: h coeffs given in reverse order: { h[nh-1], h[nh-2], ..., h[0] }
        for(i = 0; i < nh; i++)
            sum += x[i + j] * h[i];

        y[j] = sum;
    }
}
```

Encore du C canonique issu de la librairie dsplib de Texas Instruments.
Mais cette fois avec utilisation d'**entiers signés 16-bit** en format **Q1.15**.

```
#pragma CODE_SECTION(DSP_fir_gen_cn, ".text:ansi");

#include "DSP_fir_gen_cn.h"

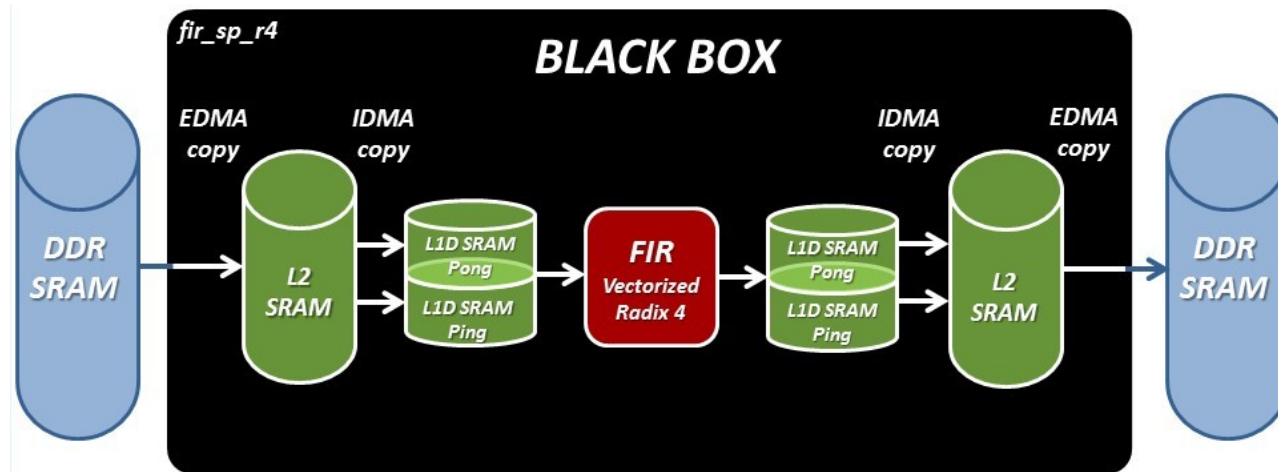
void DSP_fir_gen_cn (
    const short *restrict x,    /* Input array [nr+nh-1 elements] */
    const short *restrict h,    /* Coeff array [nh elements] */
    short *restrict r,         /* Output array [nr elements] */
    int nh,                    /* Number of coefficients */
    int nr                      /* Number of output samples */
)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Objectif du TP

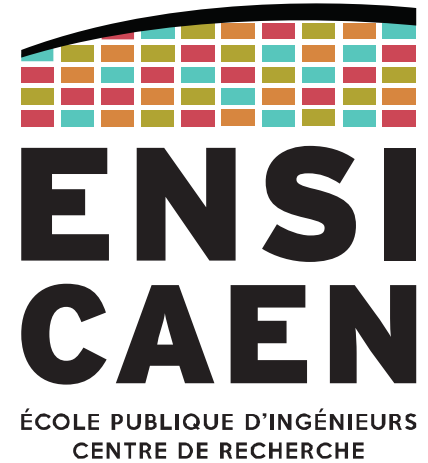
L'objectif principal du TP sera de présenter une **méthodologie d'optimisation des algorithmes pour une architecture spécifique**.

Dans notre cas, nous chercherons à optimiser un **algorithme de convolution discrète** pour un **DSP C6678 de Texas Instruments**.



JEU D'INSTRUCTIONS C6678

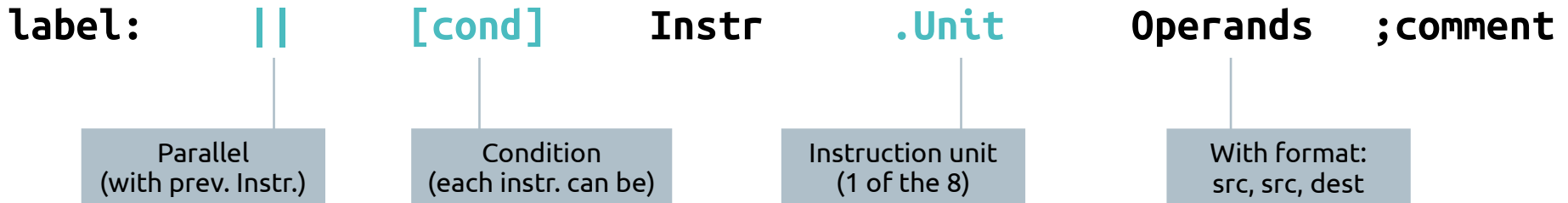
ISA – Instruction Set Architecture



Champs d'une instruction

Observons les différents champs qui constituent une instruction en assembleur pour les architectures Texas Instruments C6600.

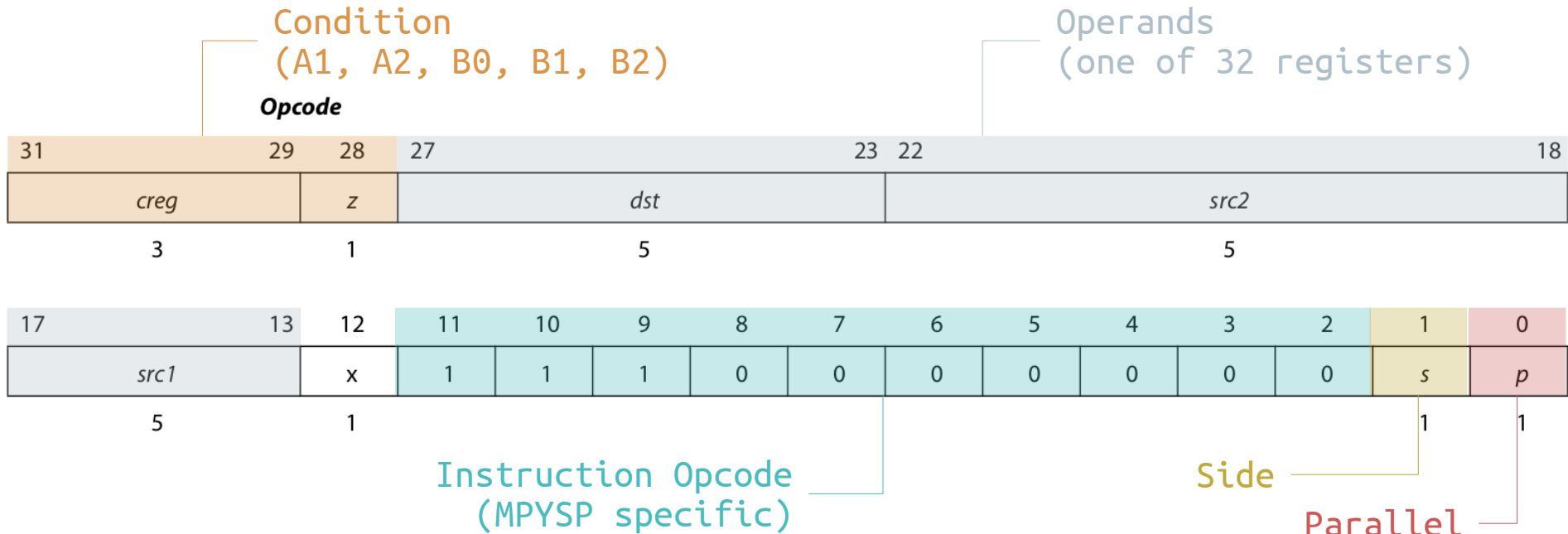
Notez que certains champs sont spécifiques aux architectures VLIW.



Code binaire opératoire

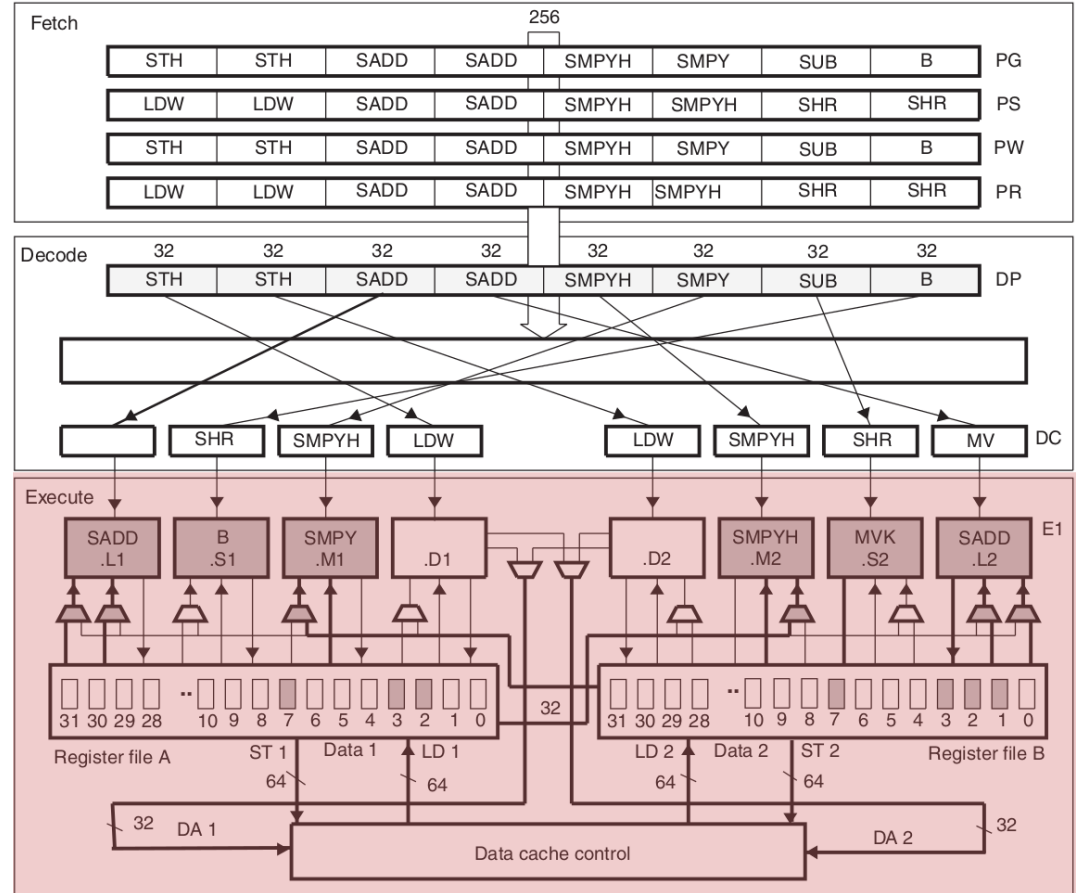
Pour rappel, à chaque champ de l'instruction assembleur correspondent un ou plusieurs bits dans le code binaire opératoire de l'instruction (sauf pour label et commentaire).

Voici par exemple l'instruction **MPYSP**.



Étage d'exécution

Afin de faciliter la compréhension du jeu d'instructions C6600 (ou **ISA** pour **Instruction Set Architecture**), nous ne regarderons que l'étage d'exécution du pipeline matériel.



Unité d'exécution – *Execution Unit (EU)*

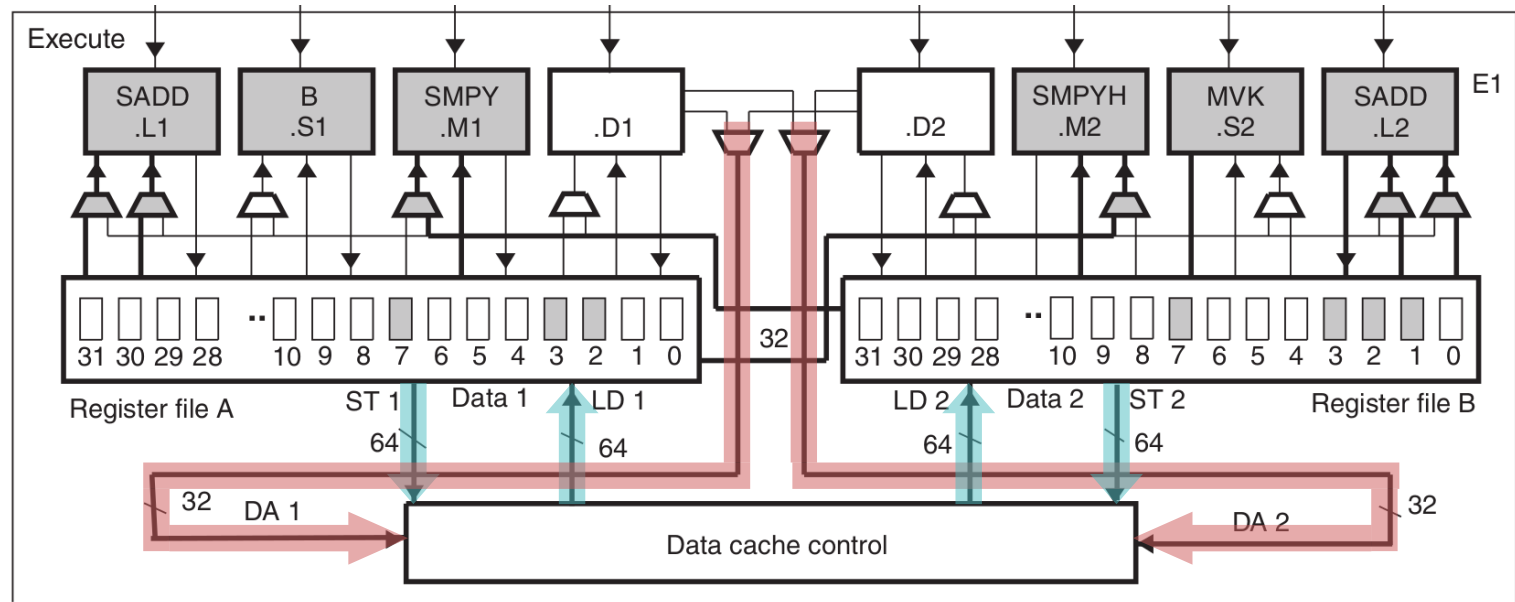
Le CPU C6600 possède une **architecture Load-Store**.

Cela signifie que deux unités d'exécution (.D1 et .D2) sont dédiées aux accès mémoire, ayant toutes les deux un accès direct vers la mémoire cache L1 (bus 64-bit).

Les autres EU sont utilisées pour le contrôle et le traitement des données.

Data bus (64-bit)
L1 cache ↔ Reg file A/B

Memory address bus (32-bit)
.D1/.D2 → L1 cache



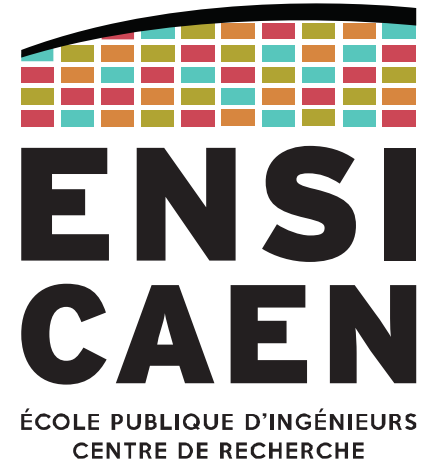
Seuls **3 modes d'adressage** sont supportés par l'ISA C6000.

Pour rappel, un mode d'adressage correspond à un mécanisme de manipulation des données par les instructions.

En tant que processeur orienté calcul, le CPU C6000 utilise massivement le mode d'adressage registre.

- **Register addressing**
 - 324 instructions (full ISA)
- **Indirect addressing**
 - 18 instructions (load/store instructions)
- **Immediate addressing**

CONVOLUTION DISCRÈTE EN ASSEMBLEUR CANONIQUE C6678



Hypothèses de départ

Nous allons maintenant traduire l'implémentation de l'algorithme de convolution discrète, passant du langage C au langage d'assemblage C6678.

La version C canonique est sur la diapo suivante.



Pour des besoins de clarté, nous décidons d'ignorer les *delay slots* associés aux instructions (temps d'exécution).

En TP, ne pas oublier d'ajouter les instructions NOP pour assurer les *delay slots* en accord avec la datasheet du DSP.

Hypothèses de départ

Voici l'algorithme de référence utilisé pendant les TP.

Il s'agit d'une implémentation en C canonique, avec *float* simple-précision IEEE-754.

```
void fir_sp (    const float * restrict xk,    \
                const float * restrict a,    \
                float * restrict yk,        \
                int na,                      \
                int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```

Hypothèses de départ

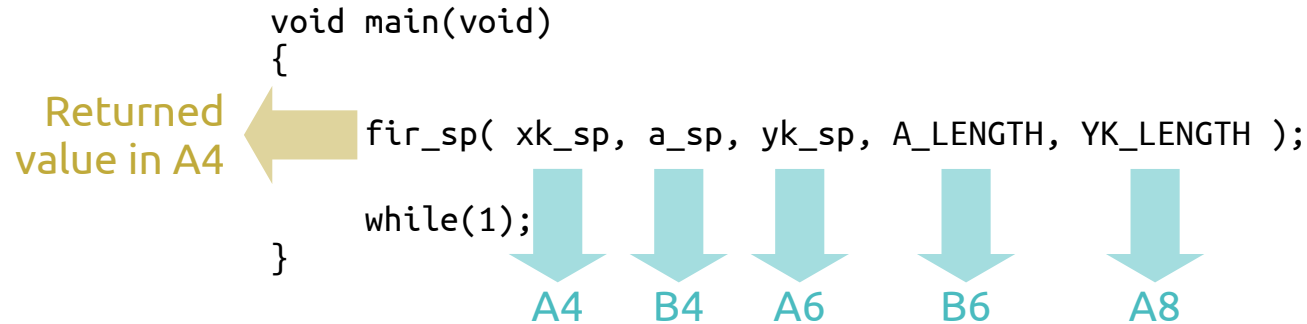
Regardons comment les registres utilisés pour passer les arguments à une fonction :

"TMS320C6000 Optimizing Compiler V7.6" User's guide, Chapter "7.3 Register conventions"

```
void main(void)
{
    fir_sp( xk_sp, a_sp, yk_sp, A_LENGTH, YK_LENGTH );
    while(1);
}
```

Returned value in A4 ←

A4 B4 A6 B6 A8



B3 register used to save return address

En plus, nous décidons d'imposer ces registres pour les variables locales :

- $i = B0$
- $j = A1$
- $yk[i] = A5$
- $xk_sp = A19$
- $a_sp = B19$
- $xk[i+j] = A9$
- $a[j] = B9$

Multiplication-Accumulation

La stratégie la plus simple pour traduire du C vers l'assembleur est de commencer par le traitement principal, dans la boucle interne.

Comme de nombreux algorithmes de traitement du signal, la convolution discrète utilise des instructions **MAC (Multiply-Accumulate)** ou **SOP (Sum Of Products)**.

Regardons d'abord l'instruction **MPYSP** :

4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

Syntax **MPYSP** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

```
fir_sp_asm:
```

```
MPYSP .M1        A9, B9, A17
```

Multiplication-Accumulation

L'exemple ci-contre présente un *cross-path*, c-à-d que les données proviennent de deux files de registres différentes : A9, A17 vs B9.

Cela apporte quelques limitations :

- Le **registre de destination (A17) et l'unité d'exécution (.M1)** doivent être du même côté
- **Seule une des deux sources (A9, B9) peut être issue de l'autre côté (ici, B9)**
- Il faut ajouter le suffixe '**x**' (*cross-path*) l'unité d'exécution indiquée (ici .M1x)

```
fir_sp_asm:
```

```
MPYSP .M1x    A9, B9, A17
```

Multiplication-Accumulation

Ajoutons maintenant l'instruction d'addition :

4.14 ADDSP

Add Two Single-Precision Floating-Point Values

Syntax **ADDSP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, .S2

Et voilà !

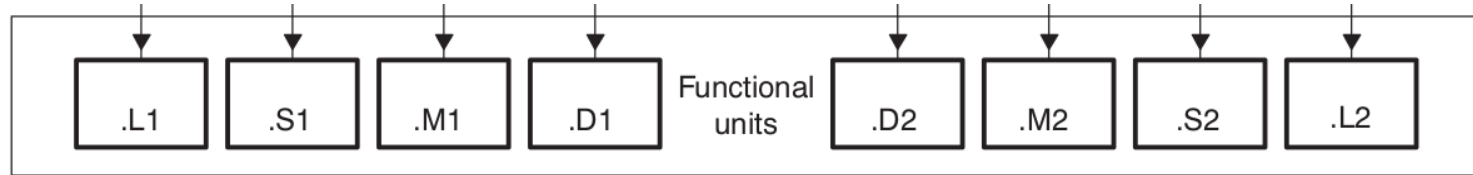
Vous venez d'implémenter
une opération MAC !



```
fir_sp_asm:
```

```
MPYSP .M1x    A9, B9, A17  
ADDSP .L1    A17, A5, A5
```

Récapitulatif des unités d'exécution :



MPYSP



ADDSP



Manipulation des données

Pour déplacer les données d'un registre du CPU vers un autre :

4.222 MV

Move From Register to Register

Syntax `MV (.unit) src2, dst`

unit = .L1, .L2, .S1, .S2, .D1, .D2

Rappel: l'instruction **MOV** fait généralement office d'opérateur d'affectation ('=' en C)

```
fir_sp_asm:
    MV      .L2      A8, B0

    MPYSP  .M1x      A9, B9, A17
    ADDSP  .L1      A17, A5, A5
```

Manipulation des données

Avant d'exécuter l'opération MAC, les données du tableau doivent être chargées depuis la mémoire cache L1 vers les registres du CPU.

On utilise alors une des instructions **LD** (*load*) :

- LDB, B = Byte = 1 byte = char
- LDH, H = Half-word = 2 bytes = short int
- LDW, W = Word = 4 bytes = int, float
- LDDW, DW = Double-Word = 8 bytes = long, double

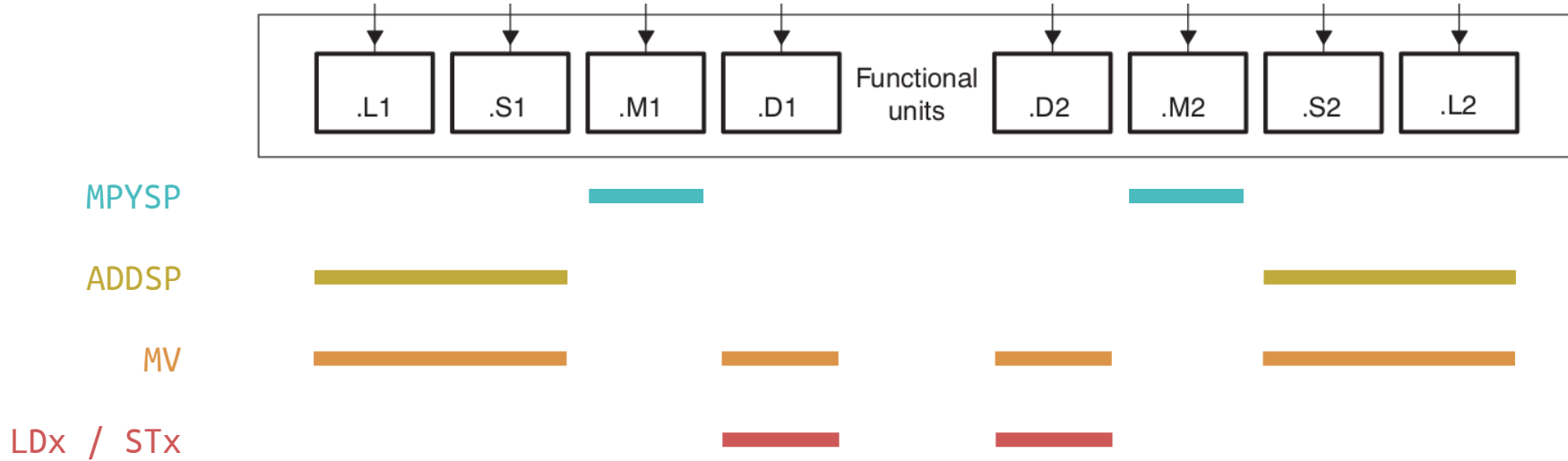
Les instructions **ST** et **LD** utilisent uniquement les unités d'exécution **.D1** et **.D2**.

```
fir_sp_asm:
    MV      .L2      A8, B0

    LDW     .D1      *A19, A9
    LDW     .D2      *B19, B9
    MPYSP   .M1x     A9, B9, A17
    ADDSP   .L1      A17, A5, A5
```

Manipulation des données

Récapitulatif des unités d'exécution :



Manipulation des données

Notons qu'une notation style-pointeur ('*') est utilisée.

Dans notre exemple, les registres **A19** et **B19** contiennent chacun une adresse.

Le caractère '*' placé devant le nom du registre indique l'utilisation du **mode d'adressage indirect**.

C'est l'équivalent des pointeurs en C.

```
fir_sp_asm:
    MV      .L2      A8, B0

    LDW     .D1      *A19, A9
    LDW     .D2      *B19, B9
    MPYSP   .M1x     A9, B9, A17
    ADDSP   .L1      A17, A5, A5
```

Manipulation des données

De manière analogue à l'utilisation de pointeurs en C, les registres utilisés en mode d'adressage indirect supportent la **pre- et post-incrémentations**.

De plus, avec les opérations *load/store*, les registres peuvent être **indexés avec la notation []**, comme les tableaux en C.

Table 3-10 Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Predecrement of Address Register	Postincrement or Postdecrement of Address Register
Register indirect	*R	*++R *- -R	*R++ *R- -
Register relative	*+R[ucst5]	*++R[ucst5]	*R++[ucst5]
	*-R[ucst5]	*- -R[ucst5]	*R- -[ucst5]
Register relative with 15-bit constant offset	*+B14/B15[ucst15]	not supported	not supported
Base + index	*+R[offsetR]	*++R[offsetR]	*R++[offsetR]
	*-R[offsetR]	*- -R[offsetR]	*R- -[offsetR]

Manipulation des données

Pour résumer sur la manipulation de données :

Les registres **A19** et **B19** contiennent l'adresse de la cellule courante des tableaux **a[]** et **xk[]**.

Les deux instructions **LDW** chargent 4 octets de la mémoire cache L1 vers les registres **A9** et **B9**.

L'adresse contenue dans les registres **A19** et **B19** sont ensuite incrémentées, les faisant ainsi pointer vers la case suivante du tableau.

```
fir_sp_asm:
    MV     .L2      A8, B0

    LDW   .D1      *A19++, A9
    LDW   .D2      *B19++, B9
    MPYSP .M1x     A9, B9, A17
    ADDSP .L1      A17, A5, A5
```


Instructions de contrôle et branchement

Historiquement, la famille C6000 ne supporte qu'une seule instruction de saut :

l'instruction **B (branch)**.

Elle permet de réaliser les opérations de contrôle élémentaires (if, for, while, ...) ainsi que les appels de fonctions.

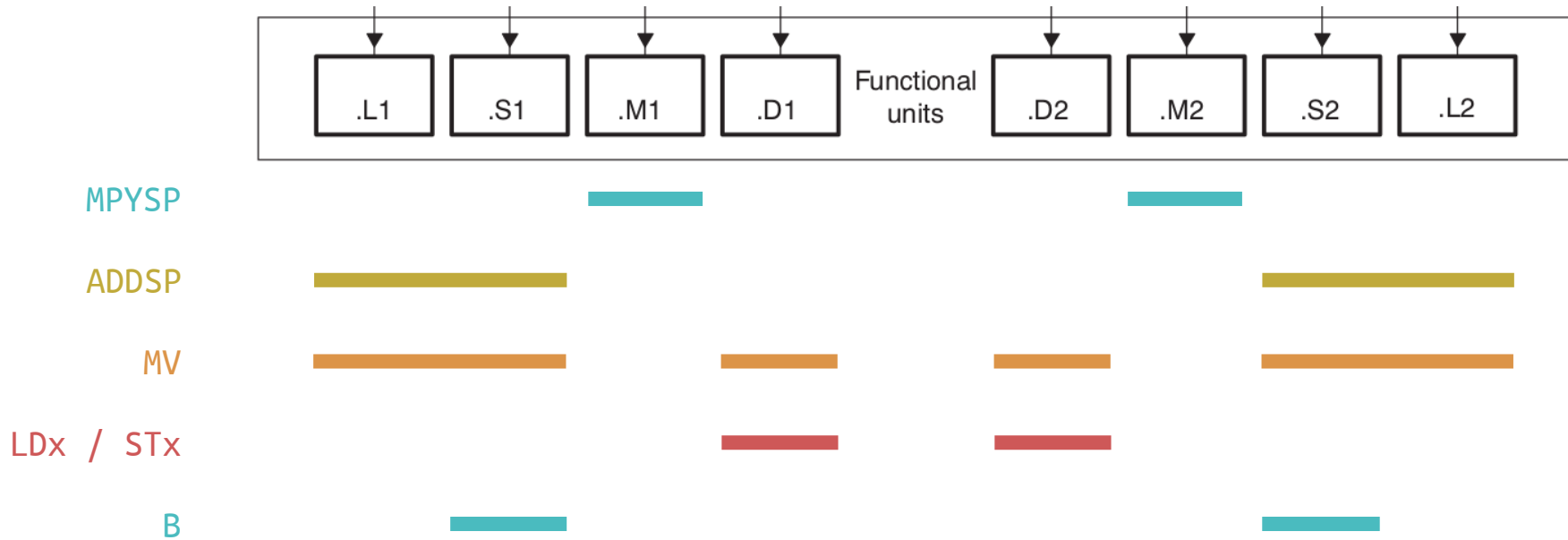
L'instruction *branch* fonctionne sur les unités d'exécution **.S1** et **.S2**.

```
fir_sp_asm:
    MV      .L2      A8, B0

fir_sp_asm_l2:
    LDW     .D1      *A19++, A9
    LDW     .D2      *B19++, B9
    MPYSP   .M1x     A9, B9, A17
    ADDSP   .L1      A17, A5, A5

    B      .S1      fir_sp_asm_l2
```

Récapitulatif des unités d'exécution :



Instructions de contrôle et branchement

Pour donner de la flexibilité malgré une seule instruction de branchement, **toute instruction peut être rendue conditionnelle.**

Cinq registres (A1, A2, B0, B1, B2) peuvent être utilisés comme conditions.

Syntaxe :

- [R] = instruction exécutée si R ≠ 0
- [!R] = instruction exécutée si R = 0

```
fir_sp_asm:
    MV      .L2      A8, B0

fir_sp_asm_l2:
    LDW     .D1      *A19++, A9
    LDW     .D2      *B19++, B9
    MPYSP   .M1x     A9, B9, A17
    ADDSP   .L1      A17, A5, A5

    [A1] B   .S1      fir_sp_asm_l2
```

Implémentation du compteur de boucle interne.

```
fir_sp_asm:
    MV     .L2     A8, B0

    MV     .L1     B6, A1

fir_sp_asm_l2:
    LDW    .D1     *A19++, A9
    LDW    .D2     *B19++, B9
    MPYSP  .M1x    A9, B9, A17
    ADDSP  .L1     A17, A5, A5
    [A1] SUB  .L1     A1, 1, A1
    [A1] B     .S1     fir_sp_asm_l2
```


Implémentation de la boucle externe.

```
fir_sp_asm:
    MV    .L2    A8, B0

fir_sp_asm_l1:

    MV    .L1    B6, A1

fir_sp_asm_l2:
    LDW   .D1    *A19++, A9
    LDW   .D2    *B19++, B9
    MPYSP .M1x   A9, B9, A17
    ADDSP .L1    A17, A5, A5
    [A1] SUB .L1  A1, 1, A1
    [A1] B    .S1  fir_sp_asm_l2

[B0] SUB .L2    B0, 1, B0
[B0] B    .S1    fir_sp_asm_l1
```

Instructions de contrôle et branchement

L'adresse de l'instruction de retour de fonction est toujours fournie à la fonction, dans le registre **B3**.

```
fir_sp_asm:
    MV    .L2    A8, B0

fir_sp_asm_l1:

    MV    .L1    B6, A1

fir_sp_asm_l2:
    LDW   .D1    *A19++, A9
    LDW   .D2    *B19++, B9
    MPYSP .M1x   A9, B9, A17
    ADDSP .L1    A17, A5, A5
    [A1] SUB .L1  A1, 1, A1
    [A1] B   .S1  fir_sp_asm_l2

    [B0] SUB .L2  B0, 1, B0
    [B0] B   .S1  fir_sp_asm_l1
```

Version finale

(sans prendre en compte les delay slots)

```
void fir_sp ( const float * restrict xk, \
             const float * restrict a,  \
             float * restrict yk,      \
             int na,                   \
             int nyk){
    int i, j;

    for (i=0; i<nyk; i++) {
        yk[i] = 0;

        /* FIR filter algorithm - dot product */
        for (j=0; j<na; j++){
            yk[i] += a[j]*xk[i+j];
        }
    }
}
```

```
fir_sp_asm:
    MV    .L2    A8, B0

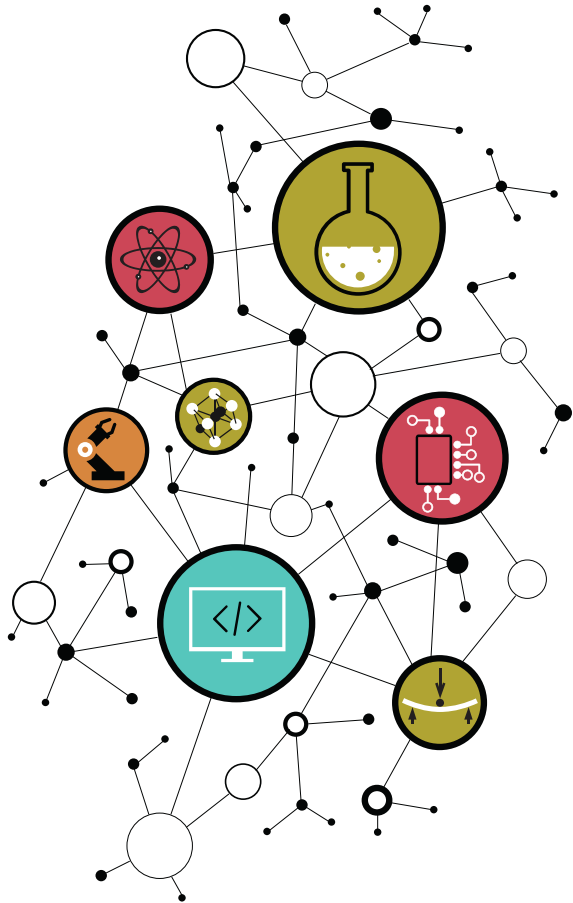
fir_sp_asm_l1:
    ZERO  .L1    A5
    MV    .L1    B6, A1
    MV    .L1    A4, A19
    MV    .L2    B4, B19

fir_sp_asm_l2:
    LDW   .D1    *A19++, A9
    LDW   .D2    *B19++, B9
    MPYSP .M1x   A9, B9, A17
    ADDSP .L1    A17, A5, A5
    [A1]  SUB    .L1    A1, 1, A1
    [A1]  B      .S1    fir_sp_asm_l2

    STW   .D1    A5, *A6++
    ADD   .L1    A4, 4, A4
    [B0]  SUB    .L2    B0, 1, B0
    [B0]  B      .S1    fir_sp_asm_l1

    B     B3
```

CONTACT



Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)



Except where otherwise noted, this work is licensed under
<https://creativecommons.org/licenses/by-nc-sa/4.0/>