

Ingénierie des Systèmes Electroniques Embarqués et Commande des systèmes

Module GE4.2 :

Processeurs de traitement du signal

Programmation bas niveau des DSPs

TMS320C6xxx

Plan

- Introduction
- Appel depuis un programme en C
- Structure minimale d'un code en assembleur
- Structure d'une ligne de code
- Description d'une instruction
- Delay Slot (DS) & FUL (Functional Unit Latency)
- Appel depuis un en langage C
- Types d'instructions
- Modes d'adressage
- Les contraintes


Introduction

- l'assembleur est plus difficile à construire qu'un code C, en compromis aux performances.
- En général on souhaite seulement optimiser des routines critiques principaux. Les autres routines n'ont pas intérêt à être optimiser tellement.
- En général, on mixe l'utilisation du C et de l'assembleur :
 - Le C pour des routines/opérations n'ayant pas besoin d'être optimisées (routines d'initialisations, déclarations des données, allocations, ...).
 - l'Assembleur pour les fonctions principaux et critiques en temps d'exécution.

Structure minimale d'un code en assembleur

- L'appel d'une routine se fait depuis le code en C de la même manière que l'appel d'une fonction en C.

```
Int main ()  
{Déclarations; Initialisations ;  
...  
...  
x = Fonction_Asm(a,b);  
...  
...  
return 0; }
```



```
.global _Fonction_Asm  
_Fonction_Asm  
    ADD ....  
    MPY ....    ; Commentaire  
label  SUB ....  
    ...  
    || ...  
    B .S2 B3  
    NOP 5
```

Structure minimale d'un code en assembleur

- ***_Fonction_Asm*** : étiquette indiquant le début de la définition de la routine ASM.
- ***.global*** : directive affecte une visibilité globale à l'entité "***_Fonction_Asm***", de telle façon que ça soit possible de l'appeler à partir d'un code C
- l'instruction **B** (branch), à la fin, s'exécutera sur l'unité **.S2** avec l'opérande dans le registre **B3**. Cet opérande ni que

l'adresse de la suite du programme de la fonction appelante. Cet adresse a été sauvegardée dans le registre B3 lors de l'appel de **_Fonction_Asm**.

```
.global _Fonction_Asm
_Fonction_Asm
    ADD ....
    MPY ....    ; Commentaire
label    SUB ....
    ...
    || ...
    B .S2 B3
    NOP 5
```

Appel d'une fonction assembleur

Les arguments fournis à un code assembleur sont automatiquement placés dans des registres bien spécifiques, ainsi :

Int _somme_asm(int *a, int *b, int *s, int n)

le premier argument (a) est placé dans A4, le 2ème (b) en B4, le 3ème (s) en A6, le 4ème (n) en B6 et ainsi de suite. Notez bien que le registre B3 est pour l'adresse de retour vers la fonction mère, le registre A4 servira également comme valeur de retour de la fonction assembleur (la somme).

Appel d'une fonction assembleur

A		B
	0	
	1	
	2	
	3	ret addr
arg1/r_val	4	arg2
	5	
arg3	6	arg4
	7	
arg5	8	arg6
	9	
arg7	10	arg8
	11	
arg9	12	arg10
	13	
	14	
	15	

Format d'une instruction en assembleur

Label: parallel bars **[condition]** instruction **unit** operands **;comment**

Label:	Parallel bars ()	[Condition]	Instruction	Unit	Operands	;Comments
--------	--------------------	-------------	-------------	------	----------	-----------

- Labels identify a line of code or a variable and represent a memory address that contains either an instruction or data. The first character of a label must be in the first column and must be a letter or an underscore (_) followed by a letter. Labels can include up to 32 alphanumeric characters.

Parallel bars - Condition

- **||** : indicates that current instruction executes in parallel with previous instruction, otherwise left blank
- **Condition** : All assembly instructions are conditional
 - If no condition is specified, the instruction executes always
 - If a condition is specified, the instruction executes only if the condition is valid
 - Registers used in conditions are A0 A1, A2, B0, B1, and B2
 - Examples:
 - [A] ;executes if $A \neq 0$
 - [!A] ;executes if $A = 0$
 - [B0] ADD .L1 A1,A2,A3

Types d'instruction

- Loading constants to registers
- Loading from memory to registers.
- Register moves, zeroing
- Storing data to memory
- Addition, Subtraction and Multiplication
- Branching and conditional operations
- Logical operations and bit manipulation

Modes d'adressage

- Un mode d'adressage correspond à une stratégie de manipulation d'une donnée par une instruction
- Les processeurs C66x sont dédiés au calcul, Seuls 3 modes d'adressage sont supportés par le jeu d'instruction (ISA) :
- ***Adressage registre*** : instructions arithmétiques, ...
- ***Adressage indirect*** : instructions de management de données de et vers la mémoire (LDx et STx)
- ***Adressage immédiat*** : move, zeroing, ...

Instructions de management de données : Move, Load & Store

- Le CPU est à modèle LOAD/STORE : des unités d'exécution dédiées aux accès mémoire (D1 et D2).
- Chacune de ces deux unités possède un chemin de 64bits (8 octets de données) vers le niveau physique L1D (Level 1, SRAM ou Cache) de la hiérarchie mémoire.

Instructions de management de données

Les instructions des familles LDx (Load) et STx (Store) sont suffixées par la taille des données à transférer :

- xxB (Byte) : 1 octet / 8 bits (char)
- xxH (Half Word) : 2 octets / 16 bits (short)
- xxW (Word) : 4 octets / 32 bits (int, float)
- xxDW (Double Word) : 8 octets / 64 bits (long, double)

Exemple : LDW .D1 *A6, A3

Remarque : le registres contenant un pointeur (adressage indirect) sont préfixés par le symbole *, comme en langage C.

Instructions arithmétiques

- Add/subtract/multiply:

ADD .L1 A3,A2,A1 ; $A1 \leftarrow A2 + A3$

SUB .S1 A1,1,A1 ; decrement A1

MPY .M2 A7,B7,B6 ; multiply LSBs

|| MPYH .M1 A7,B7,A6 ; multiply MSBs

Instructions pour branchement et boucle

- exemple de boucle :

```

                                MVK .S1 count, A1      ;loop   counter
                                ||
                                MVKH .S2 count, A1
Loop                            MVK .S1 val1, A4        ;loop
                                MVKH .S1 val1, A4      ;body

                                SUB .S1 A1,1,A1 ;decrement counter
[A1]                            B .S2 Loop            ;branch if A1 ≠ 0
                                NOP 5                  ;5 NOPs for branch
```

Jeu d'instruction par unité

.S Unit	
ADD	MVKLH
ADDK	NEG
ADD2	NOT
AND	OR
B	SET
CLR	SHL
EXT	SHR
MV	SSHL
MVC	SUB
MVK	SUB2
MVKL	XOR
MVKH	ZERO

.M Unit	
MPY	SMPY
MPYH	SMPYH

Other	
NOP	IDLE

.L Unit	
ABS	NOT
ADD	OR
AND	SADD
CMPEQ	SAT
CMPGT	SSUB
CMPLT	SUB
LMBD	SUBC
MV	XOR
NEG	ZERO
NORM	

.D Unit	
ADD	STB/H/W
ADDA	SUB
LDB/H/W	SUBA
MV	ZERO
NEG	

Note: Refer to the 'C6000 CPU Reference Guide for more details.

Delay Slots & Functionnal Unit Latency

- La plupart des instructions assembleur , nécessitent une latence, appelée "Delay Slots" [**DS**], avant de pouvoir retourner un résultat/exécution. Par exemple pour le cas d'un branchement on trouve un "delay slots"=5, pour une multiplication flottante à simple précision la latence DS=3.
- Une autre caractéristique des instructions assembleurs C6000 est la latence sur les unités d'exécution "Functional Unit Latency = Latence de l'unité fonctionnelle " [**FUL**], qui indique le nombre des cycles que l'instruction assembleur occupe une unité d'exécution. Presque seules les instructions à double précision (Ex : MPYDP) nécessitent $FUL > 1$, par contre, les autres instructions ont quasiment toutes $FUL = 1$.

L'étage d'exécution

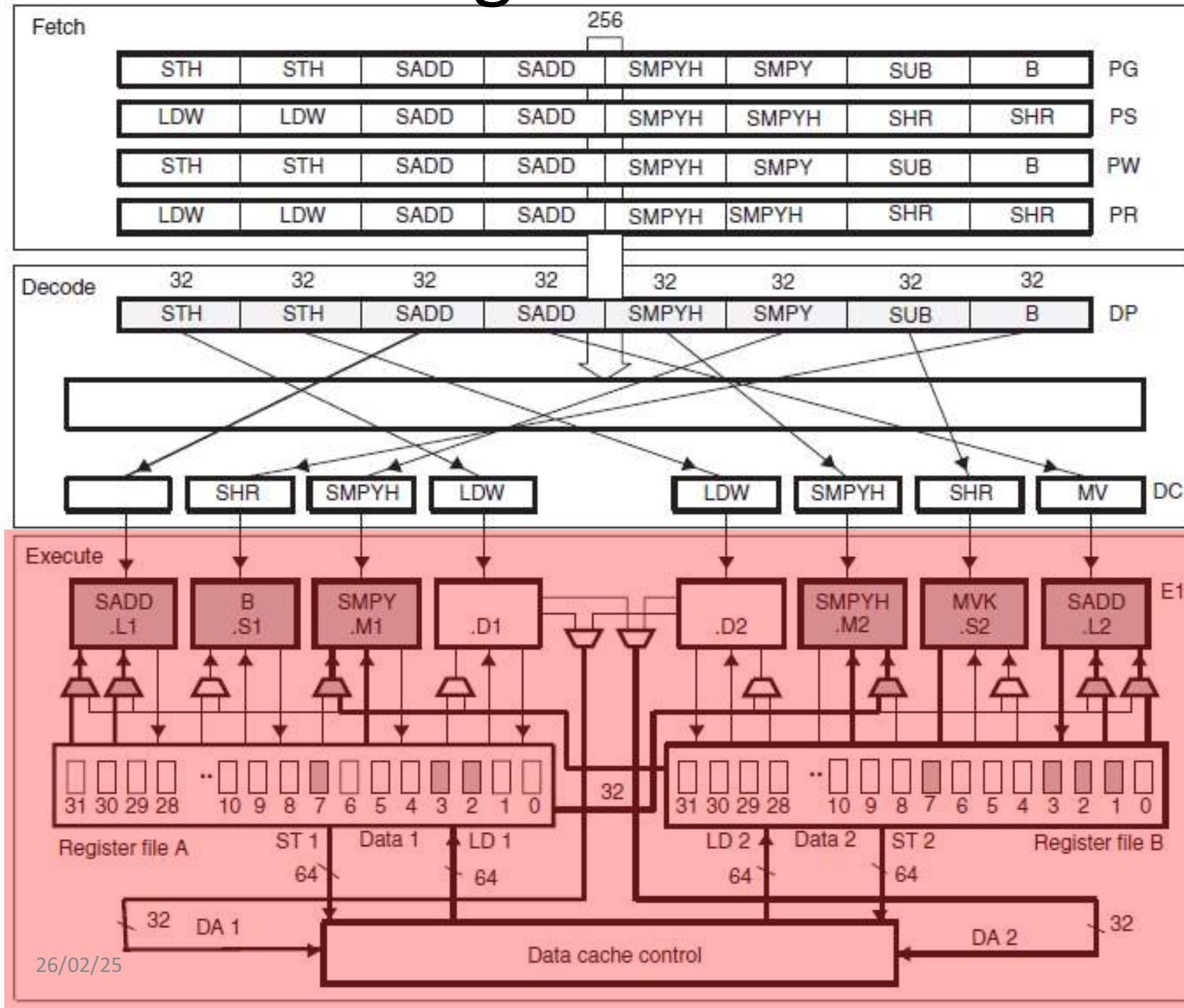


Table 4-2. Execution Stage Length Description for Each Instruction Type

Execution phases	Instruction Type				
	Single Cycle	16 × 16 Single Multiply	Store	Load	Branch
E1	Compute result and write to register	Read operands and start computations	Compute address	Compute address	Target code in PG [‡]
E2		Compute result and write to register	Send address and data to memory	Send address to memory	
E3			Access memory	Access memory	
E4				Send data back to CPU	
E5				Write data into register	
Delay slots	0	1	0 [†]	4 [†]	5 [‡]

[†] See sections 4.2.3 and 4.2.4 for more information on execution and delay slots for stores and loads.

[‡] See section 4.2.5 for more information on branches.

Notes: 1) This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

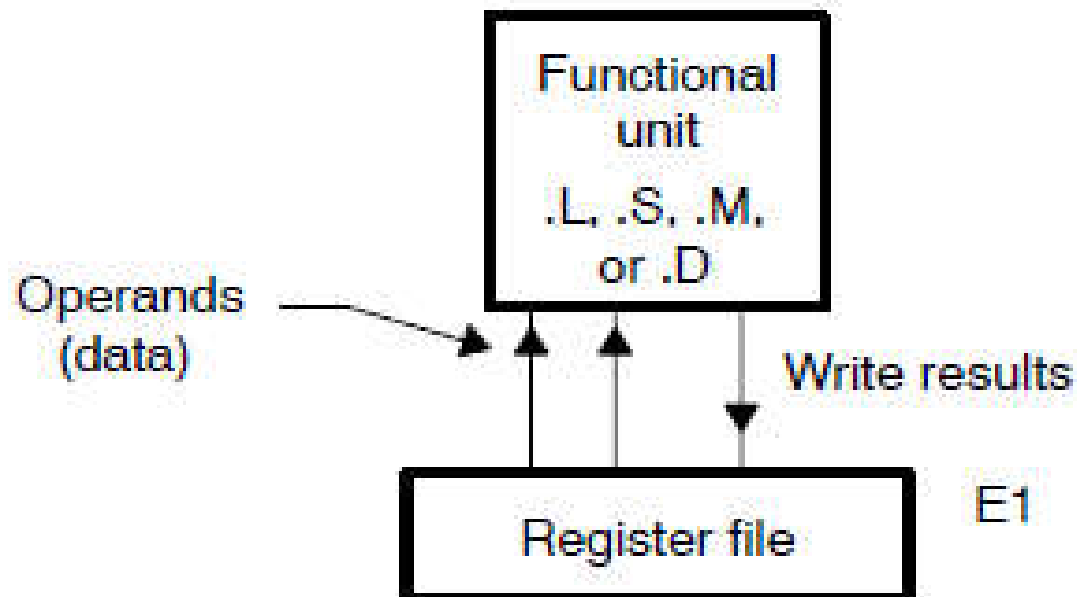
2) **NOP** is not shown and has no operation in any of the execution phases.

Delay Slots

Single-Cycle Instruction Phases



Single-Cycle Instruction Execution Block Diagram

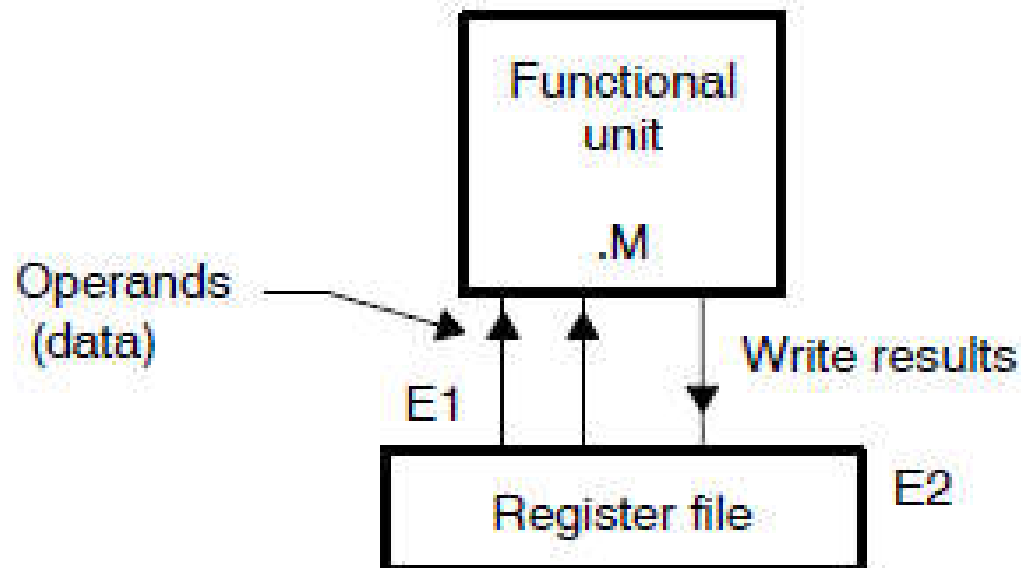


Delay Slots

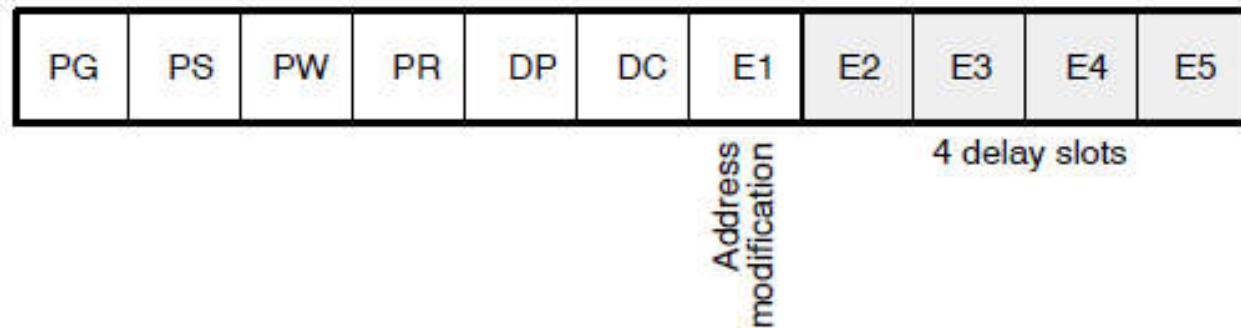
Two-Cycle Instruction Phases



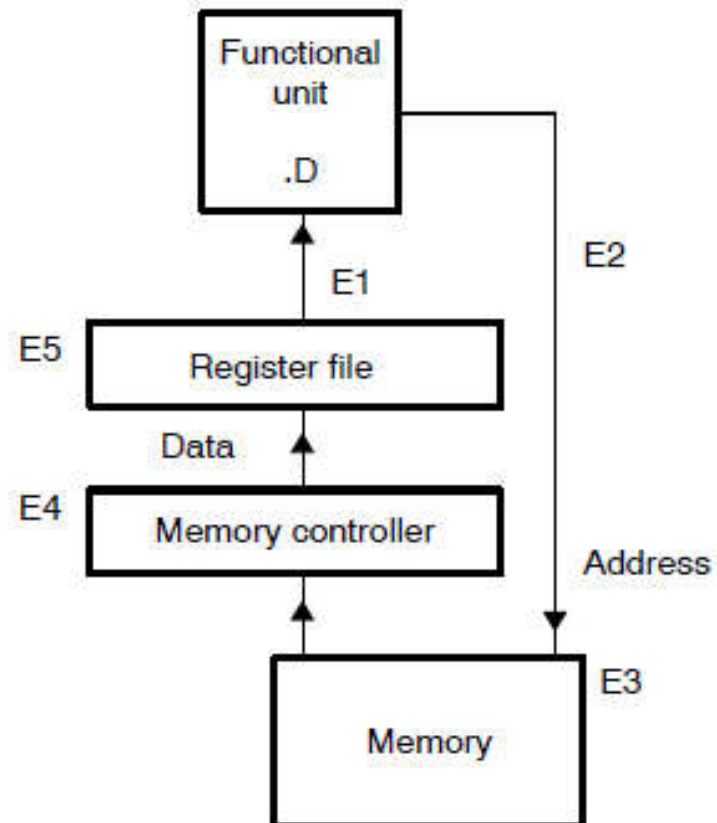
Single 16 x 16 Multiply Instruction Execution Block Diagram



Load Instruction Phases



Load Instruction Execution Block Diagram



Instruction Delays

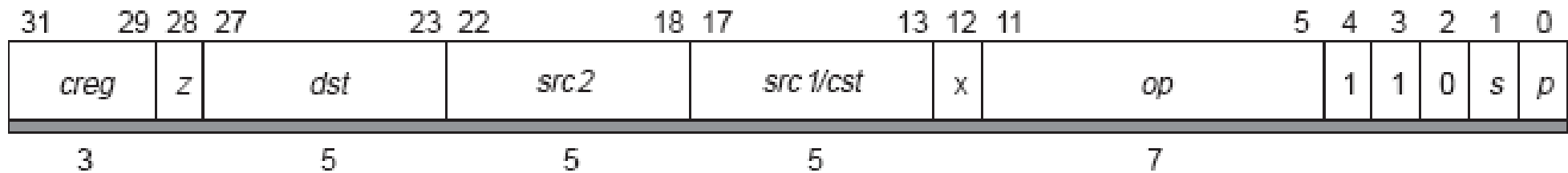
All C66x instructions require only one cycle to execute, but some results are delayed.

Description	Instruction Example	Delay
Single Cycle	All instructions except	0
Integer multiplication and new floating point	MPY, FMPYSP	1,2
Legacy floating point multiplication	MPYSP	3
Load	LDW	4
Branch	B	5

FUL

- The functional unit latency is equivalent to the number of cycles that must pass before the functional unit can start executing the next instruction.
- All new floating point instructions have a functional unit latency of one cycle (can be fully pipelined). Improvements of the existing floating have also been realized, but in order to maintain a fully backward binary compatibility, new instructions opcode have been created.
- **FMPYDP** is the optimized version of **MPYDP**
- **FADDSP/FSUBSP** are the optimized version of **ADDSP/SUBSP**
- **FADDDP/FADDDP** are the optimized version of **ADDDP/SUBDP**

Opcode map



- **creg**: pour specifier le registre à tester quand l'instruction est à traiter conditionnellement.
- **Z**: pour le test de la condition.

Si Z est mis à zéro alors le registre spécifié dans creg est testé pour voir s'il est mis à zéro. Si la condition est vraie, alors l'instruction conditionnelle sera traitée.

- **dst**: destination -- **src1/src2**: sources -- **op**: opcode
- **S**: sélectionne le côté A ou B pour la destination
- **x**: utiliser le chemin croisé pour src2
- **P**: détermine le statut de parallélisme de l'instruction

Accès aux variables globales depuis l'Asm

```
Int somme(int, int);  
int x = 7, y, w = 3;  
  
void main (void)  
{  
    y = somme(x, 5);  
}
```

```
; somme.asm  
  
        .global  _somme  
        .global  _w  
  
_somme:  
        mvkl     _w, A1  
        mvkh     _w, A1  
        ldw      *A1, A0
```

- Déclarer les variables globaux
- Utilisez un trait de soulignement pour accéder aux variables C
- Avantages de déclarer des variables en C :
Déclarer en C est plus facile et le compilateur initialise la variable (int w = 3)

General-Purpose Register Files

- C66x CPU contains two general purpose register files (A and B). Each of these files contains thirty-two 32-bit registers (A0-A31 for file A and B0-B31 for file B).
- Support data ranging in size from packed 8-bit through 128-bit fixed point data.
- Values larger than 32 bits (such as 40-bit values and 64-bit quantities) are stored in register pairs. Values larger than 64-bits (such as 128-bit quantities) are stored in a pair of register pair (a quadruplet of registers).

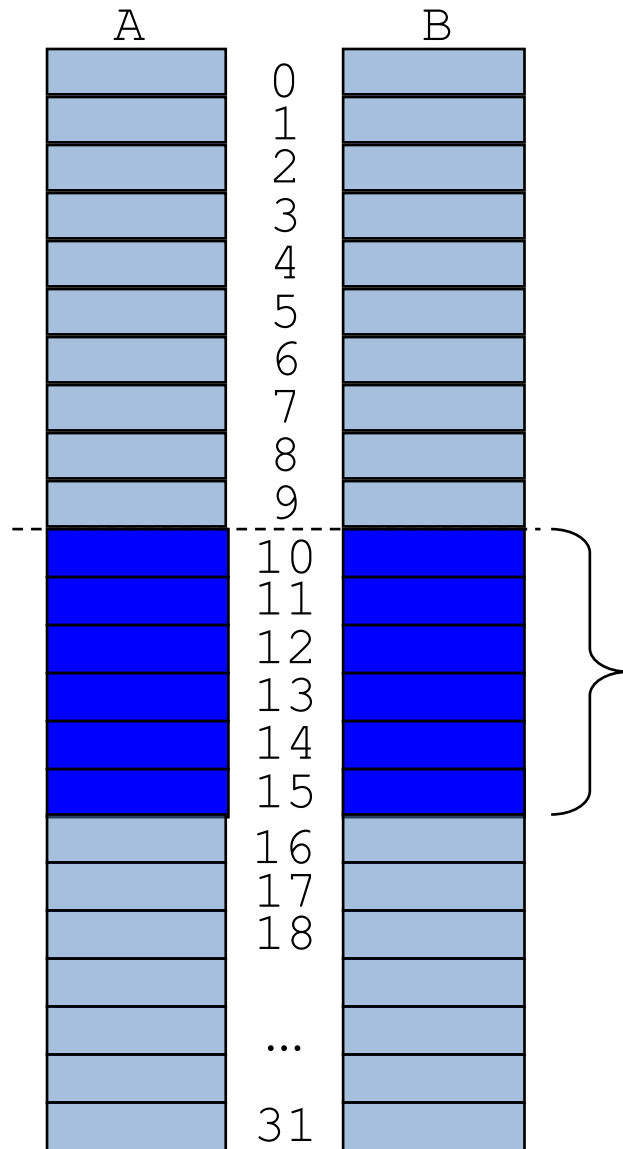
64-bit Register Pairs

Register File	
A	B
A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14
A17:A16	B17:B16
A19:A18	B19:B18
A21:A20	B21:B20
A23:A22	B23:B22
A25:A24	B25:B24
A27:A26	B27:B26
A29:A28	B29:B28
A31:A30	B31:B30

128-bit Register Quadruplets

Register File	
A	B
A3:A2:A1:A0	B3:B2:B1:B0
A7:A6:A5:A4	B7:B6:B5:B4
A11:A10:A9:A8	B11:B10:B9:B8
A15:A14:A13:A12	B15:B14:B13:B12
A19:A18:A17:A16	B19:B18:B17:B16
A23:A22:A21:A20	B23:B22:B21:B20
A27:A26:A25:A24	B27:B26:B25:B24
A31:A30:A29:A28	B31:B30:B29:B28

Registres utilisés par le code en C



A sauvegarder puis à restaurer si on les utilise en assembleur

Assembleur linéaire

- enables writing assembly-like programs without worrying about register usage, pipelining, delay slots, etc.
- The assembler optimizer program reads the linear assembly code to figure out the algorithm, and then it produces an optimized list of assembly code to perform the operations.
- Source file extension is .sa
- The linear assembly programming lets you:
 - use symbolic names
 - forget pipeline issues
 - ignore putting NOPs, parallel bars, functional units, register names
 - more efficiently use CPU resources than C.

Example : Assembleur Linéaire vs Assembleur standard

```

_sumfunc:      .cproc      np      ;.cproc directive starts a C callable
procedure
    .reg      y      ;.reg directive use descriptive names for values that will be stored in
registers

    MVK      np,cnt
loop:          .trip      6      ; trip count indicates how many times a loop will
iterate
    SUB      cnt,1,cnt
    ADD      y,cnt,y
    [cnt]    B      loop

    .return   y
    .endproc      ; .endproc to end a C procedure
  
```

-----Equivalent assembly function-----

```

.def _sumfunc
_sumfunc:      MV      .L1 A4,A1 ;n is loop counter
LOOP:          SUB      .S1 A1,1,A1 ;decrement n

              ADD      .L1 A4,A1,A4 ;A4 is accumulator
              [A1] B    .S2 LOOP ;branch if A1 ≠ 0
              NOP      5      ;branch delay nops
              B      .S2      B3      ;return from calling
              NOP      5      ;five NOPS for delay
              .end
  
```


Tradeoffs

	C/C++	Assembly	Linear Assembly
Instruction Selection	Automatic	Manual	Manual
Partitioning	Automatic	Manual	Optional
Functional Unit Allocation	Automatic	Manual	Automatic
Instruction Scheduling	Automatic	Manual	Automatic
Register Allocation	Automatic	Manual	Optional
Software Pipelining	Automatic	Manual	Automatic
Ease of Use	Easy	Hard	In-between
Performance	Good	Best	Ok
Portability Across Compiler Releases			

Chaine de compilation

