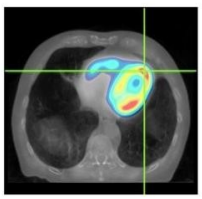
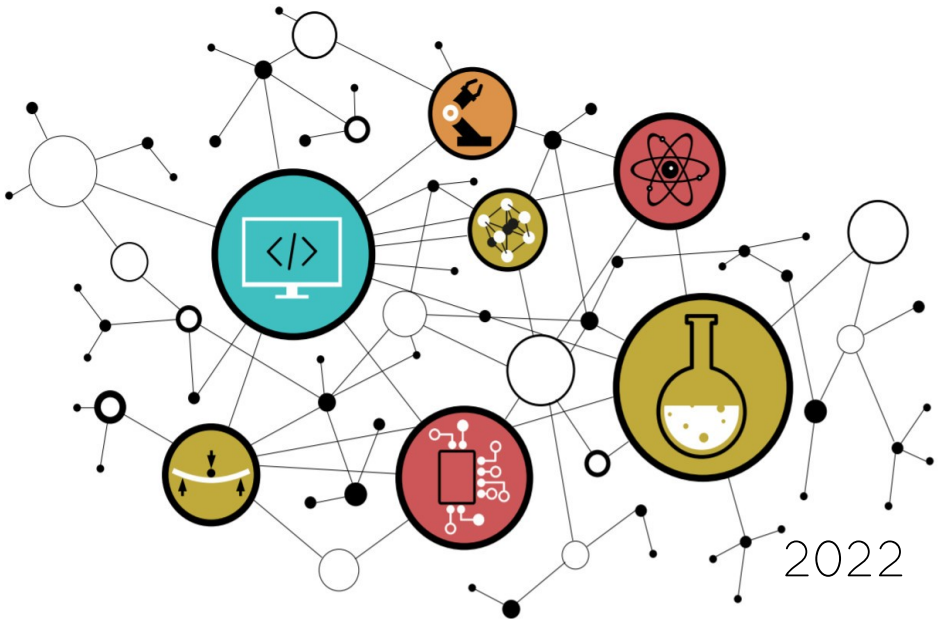
ARCHITECTURES POUR LE CALCUL

TRAVAUX PRATIQUES



2022

CONTACTS

hugo descoubes
hugo.descoubes@ensicaen.fr
+33 (0)2 31 45 27 61

Isabelle Lartigau
isabelle.lartigau@ensicaen.fr

Emmanuel Cagniot
emmanuel.cagniot@ensicaen.fr

ENSICAEN
6 boulevard Maréchal Juin
CS 45053
14050 CAEN cedex 04

RESSOURCES

Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSICAEN. Télécharger l'archive complète de travail dsp.zip

https://foad.ensicaen.fr/course/view.php?id=117

ÉVALUATION

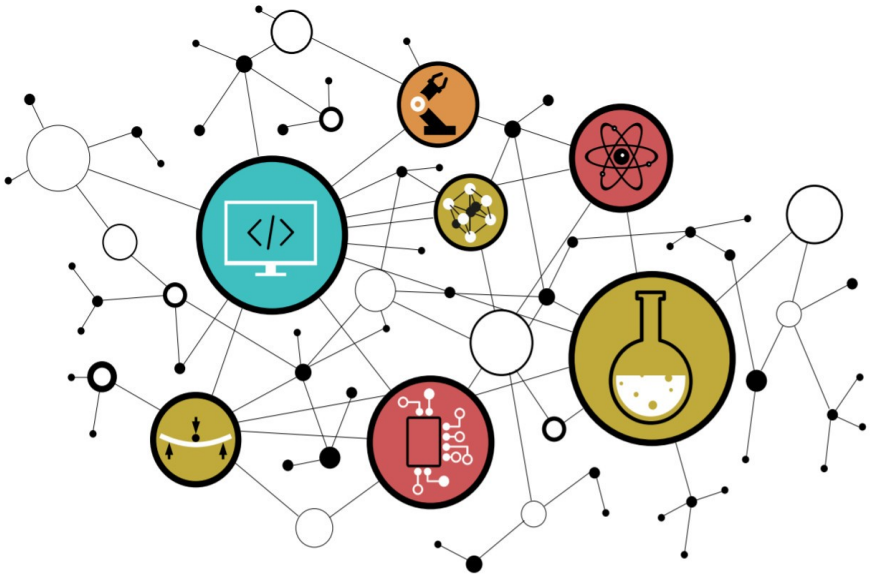
- Examen de pratique sur ordinateur (1h30)

L'évaluation de la compétence se fera sur machine personnelle ou machine école et portera sur les points suivants :

- Création d'un projet sous IDE CCS. A l'image du projet présent dans cm/eval/examen_nom
- Optimisation d'une fonction algorithmique élémentaire (cf. trame de TP)
  - écriture en C canonique
  - écriture ASM C6000 canonique
  - écriture ASM VLIW
  - écriture de l'algorithme optimisé avec l'une des techniques avancée suivante :
    - Vectorisation en langage C par programmation intrinsèque
    - Pipelining software en ASM C6000
    - Vectorisation en base 2 ou 4 en ASM C6000

TRAVAUX PRATIQUES

PRÉLUDE



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

La trame de TP minimale que nous considérons comme être les compétences minimales à acquérir afin d'accéder aux métiers de base du domaine suit le séquencement suivant : chapitres 1, 2 et 3. Le reste de la trame ne sera pas évalué et représente une extension au jeu de compétences minimales relatif au domaine en cours d'étude (* devant chapitres facultatifs voire complémentaires). Libre à vous d'aller plus loin selon votre temps disponible et votre volonté de mieux comprendre et maîtriser ce domaine !

1. PRÉLUDE

- 1.1. Optimisation algorithmique
- 1.2. Convolution discrète ou filtre FIR
- 1.3. Exemple des applications Radar
- 1.4. Architecture DSP VLIW C6600 de TI
- 1.5. Outils de développement
- 1.6. Objectifs pédagogiques
- 1.7. Benchmarking

2. PROGRAMMATION VECTORIELLE SUR DSP VLIW C6600

- 2.1. Création du projet de test
- 2.2. Analyse du programme de test
- 2.3. Assembleur canonique C6600
- 2.4. Assembleur VLIW C6600
- 2.5. Pipelining software en assembleur C6600
- 2.6. Vectorisation d'algorithme en assembleur C6600
- 2.7. Déroulement de boucle en C canonique
- 2.8. Vectorisation d'algorithme en C intrinsèque

3. PROGRAMMATION VECTORIELLE SUR GPP INTEL x86_64

- 3.1. Analyse du programme de test
- 3.2. Vectorisation avec ISA extension SSE4.1
- 3.3. Synthèse

*4. MÉMOIRE CACHE ET MÉMOIRE SRAM ADRESSABLE

- 4.1. Mémoire locale SRAM adressable
- 4.2. Préchargement des données de DDR DRAM vers L2 SRAM
- 4.3. Préchargement des données de L2 SRAM vers L1D SRAM

*5. PÉRIPHÉRIQUES DE COPIE MÉMOIRE DMA

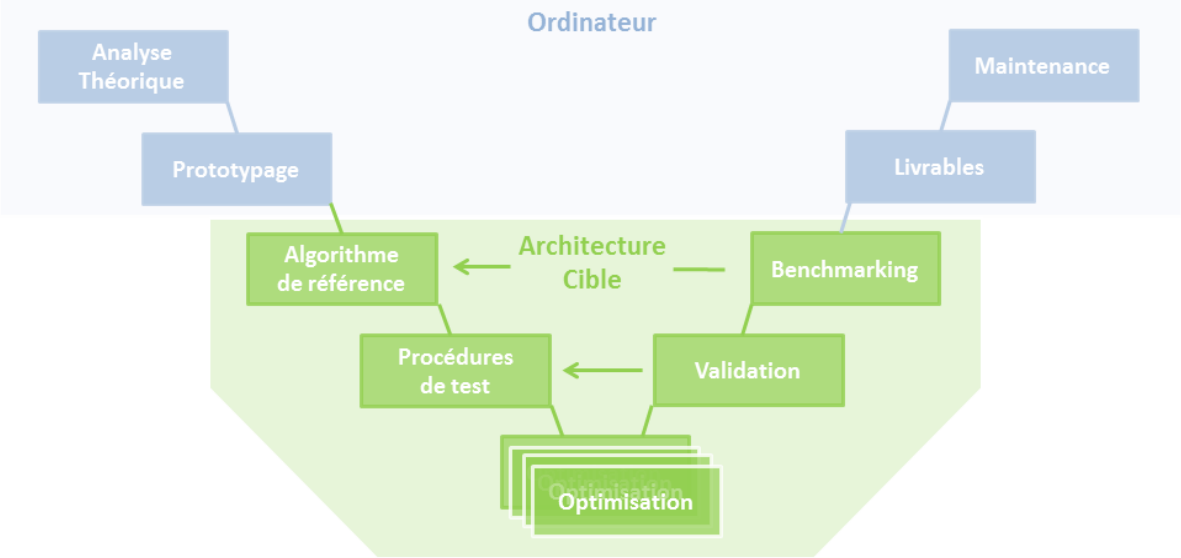
- 5.1. Transferts par IDMA
- 5.2. Stratégie Ping Pong
- 5.3. Transferts par EDMA

SEQUENCEMENT

Le séquencement de la trame de Travaux Pratiques correspond au chemin proposé afin d'atteindre les objectifs pédagogiques fixés. Ces objectifs ont été choisis au regard des attentes et exigences demandées par les marchés de l'industrie du logiciel et des couches basses des systèmes : systèmes embarqués, systèmes temps réel, développement de systèmes d'exploitation, développement de bibliothèques spécialisées, développement de chaînes de compilation, attaque et sécurité des systèmes, etc, tous des métiers dans divers domaines actuellement exercés par certains de nos anciens élèves. Il s'agit d'un séquencement conseillé qui n'a en aucune façon volonté à être imposé (étudiant comme enseignant encadrant). Pour se dérouler sous les meilleurs hospices, il serait cependant préférable dès le début de l'enseignement de rythmer 1 à 2 heures par semaine de travail personnel à la maison en dehors des séances en présentiel avec enseignant. Voici une proposition de séquencement (2h par créneau de TP) :

- Avant le début des TP : Lire le document de prélude, installer les outils de développement (IDE CCS 5.5, bibliothèques CSL et DSPLIB), valider l'exercice 2.1 (projet hello seulement) et faire le travail préparatoire n°1 (dsp-tp-preparations.pdf)
- Séance n°1 : Exercices 2.1 et 2.2
- Séance n°2 : Exercice 2.3 et 2.4 *(avant la séance, faire le travail préparatoire n°2)*
- Séance n°3 : Exercice 2.5
- Séance n°4 : Exercice 2.6
- Séance n°5 : Exercice 2.7
- Séance n°6 : Exercice 2.8
- Séance n°7 : Exercices 3.1 et 3.2
- Séance n°8 : Exercice 4.1 et 4.2 *(avant la séance, faire le travail préparatoire n°4)*
- Séance n°9 : Exercice 4.3
- Séance n°10 : Exercice 5.1, etc *(avant la séance, faire le travail préparatoire n°5)*

1.1. Optimisation algorithmique



Durant cette trame de travaux pratiques, nous allons nous intéresser au workflow typiquement rencontré en milieu industriel dans le cadre d'optimisations logicielles d'algorithmes embarqués sur une cible matérielle spécialisée. Ce processus de développement peut par exemple être rencontré chez les acteurs des grands domaines du traitement du signal (traitement d'antenne, traitement d'image, traitement du son, cloud computing, etc). Nous avons travaillé par le passé et travaillons encore à l'ENSICAEN avec plusieurs partenaires industriels ayant ces types de contraintes (THALES, SAFRAN, CANON, diverses entreprises en traitement d'image, etc).

Nous nous intéresserons à l'implémentation d'un algorithme simple et standard du domaine du traitement du signal, un filtre FIR ou produit scalaire (convolution discrète). Nous nous attarderons sur les stratégies d'optimisation pour une architecture matérielle spécialisée et non à la théorie mathématique associée. Tous nos développements seront guidés par le test, étape pouvant tenir une place très importante dans le temps de développement global d'une application et le Benchmarking (analyse comparative). Observons le workflow de la trame de travaux pratiques :

- Analyse mathématique théorique de l'algorithme sur ordinateur. Optimisation mathématique à cette étape afin de diminuer la complexité algorithmique (notation "grand O" ou "Omicron" de Landau) notamment en MAC (Multiply-Accumulate) pour un algorithme du TNS (Traitement Numérique du Signal)
- Modélisation, conception et validation durant les phases de prototypage sur ordinateur. L'outil logiciel de prototypage rapide le plus rencontré en milieu industriel à notre époque dans le domaine du traitement du signal est Matlab/Simulink (Scilab en version libre). Cette étape est essentielle afin de valider la structure en pseudo code des algorithmes, les procédures de test ainsi que les vecteurs d'entrée et de sortie
- Développement et validation sur cible des procédures de test. Dans le cadre de nos développements, nous nous intéresserons aux tests de conformité/validité (cohérence des valeurs de sortie par rapport à l'algorithme de référence) et de performance (mesures temporelles)
- Développement sur cible d'un algorithme de référence en C canonique ou C naturel
- Développements, tests et validations successives sur cible des stratégies d'optimisation sur architecture processeur spécialisée (vectorisation monocœur, parallélisation multicœur, gestion optimale de la hiérarchie mémoire L1/L2/L3, périphériques d'accélération, périphériques DMA, etc)

1.2. Convolution discrète ou filtre FIR

Cette trame de TP consiste à implémenter un algorithme optimisé de filtrage FIR (ou produit scalaire ou convolution discrète) sur une architecture DSP spécialisée pour du calcul numérique. Effectuons quelques rappels sur cet algorithme. Une implémentation courante de ce filtre consiste à appeler périodiquement (période d'échantillonnage) l'algorithme dans une optique de calcul en temps réel. A chaque appel, seul un échantillon de sortie est calculé puis traité :

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

x() = vecteur d'échantillons d'entrée (taille égale à N)
a() = vecteur de coefficients
y(k) = échantillon courant de sortie
k = indice courant
N = ordre du filtre
N+1 = nombre de coefficients

L'implémentation vue en travaux pratiques se fera en temps différé. Nous calculerons un vecteur de sortie complet en traitant l'information depuis un vecteur d'entrée pouvant être très long (plusieurs Mo) et dans tous les cas de figure de taille supérieure ou égale au nombre de coefficients (ordre du filtre FIR + 1) :

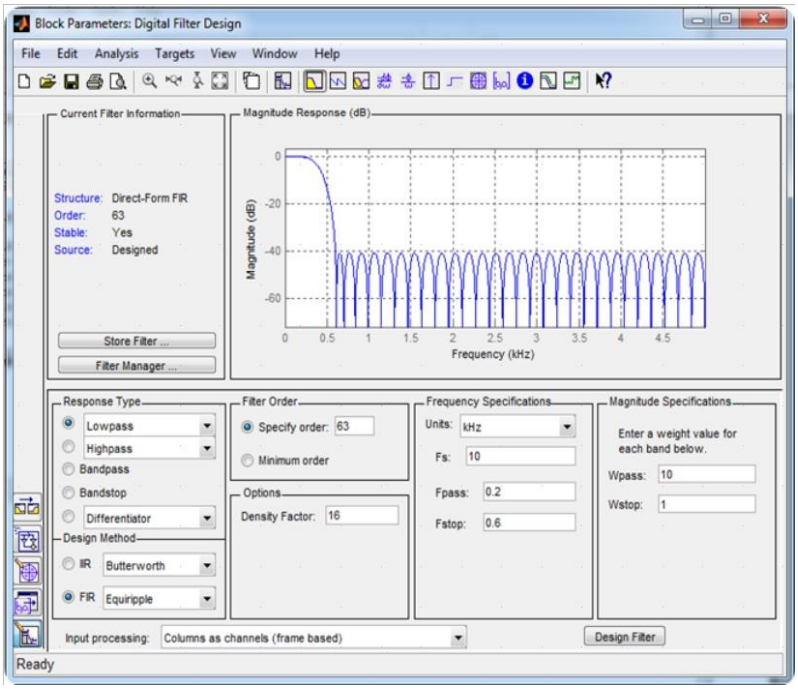
$$y(k)=\sum_{k=0}^{Y}\sum_{j=0}^{N} a(j).x(k-j)$$

x() = vecteur d'échantillons d'entrée (taille supérieure ou égale à N)
a() = vecteur de coefficients
y() = vecteur d'échantillons de sortie
k = indice courant
N = ordre du filtre
N+1 = nombre de coefficients
Y = taille du vecteur de sortie
Y+N-1 = taille du vecteur d'entrée

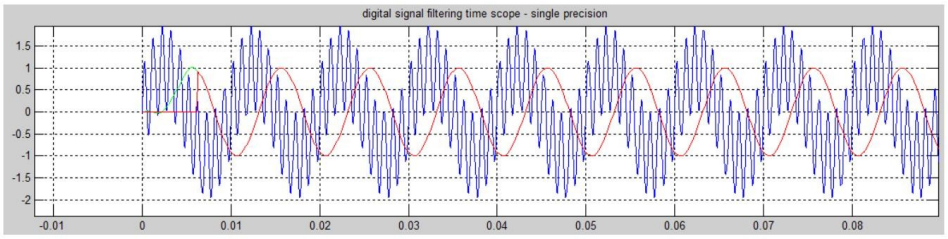
La phase de prototypage rapide sous environnement Matlab/Simulink ne sera pas à faire et vous est donnée dans le répertoire de projet /disco/matlab/. Elle est entièrement recouverte par le programme de 1ière année en Traitement Numérique du Signal. Nous nous attarderons uniquement sur les problématiques liées à l'intégration et l'optimisation sur DSP VLIW TMS320C6678. Observons l'implémentation en pseudo code Matlab au format flottant simple précision IEEE754 de l'algorithme de filtrage en temps différé précédemment présenté. Comprendre cet algorithme ...

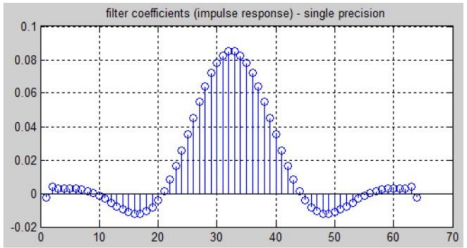
```
function yk = fir_sp(xk, coeff, coeffLength, ykLength)
    yk = single(zeros(1,ykLength));    % output array preallocation

    % output array loop
    for i=1:ykLength
        yk(i) = single(0);

        for j=1:coeffLength
            yk(i) = single(yk(i)) + single(coeff(j)) * single(xk(i+j-1));
        end
    end
end
```

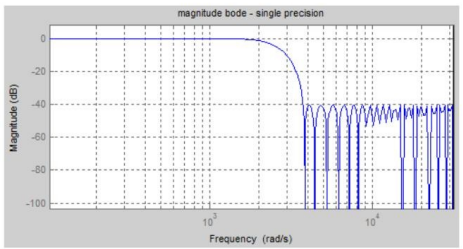
Le filtre à intégrer sur cible ainsi que les vecteurs de test d'entrée ont été générés et validés durant les phases de prototypage sous Matlab/Simulink en utilisant l'outil FDATool (cf. capture ci-dessous):



Le filtre FIR à intégrer est un filtre Passe Bas coupant à 200Hz pour une fréquence d'échantillonnage à 10KHz. Il s'agit d'un filtre ordre 63 comprenant donc 64 coefficients de symétrie paire en flottant simple précision IEEE-754 (cf. ci-dessous). Ce filtre offre un gain unitaire dans la bande passante et une atténuation de -40dB dans la bande coupée. La bande d'atténuation est comprise entre 200Hz et 600Hz. Ce filtre ne vise aucune application ou domaine spécifique et ne nous servira qu'à illustrer les concepts et stratégies d'optimisation sur processeur spécialisé. Le vecteur de test d'entrée est une simple somme de deux sinusoïdes à 100Hz et 1KHz (bleu ci-dessous). Après filtrage, seule l'harmonique à 100Hz (rouge ci-dessous) est identifiable :

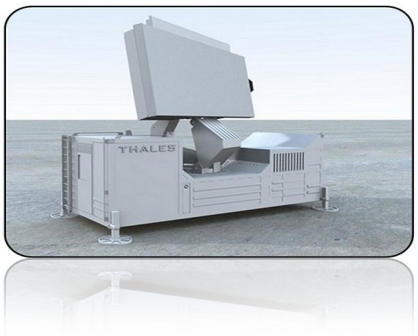






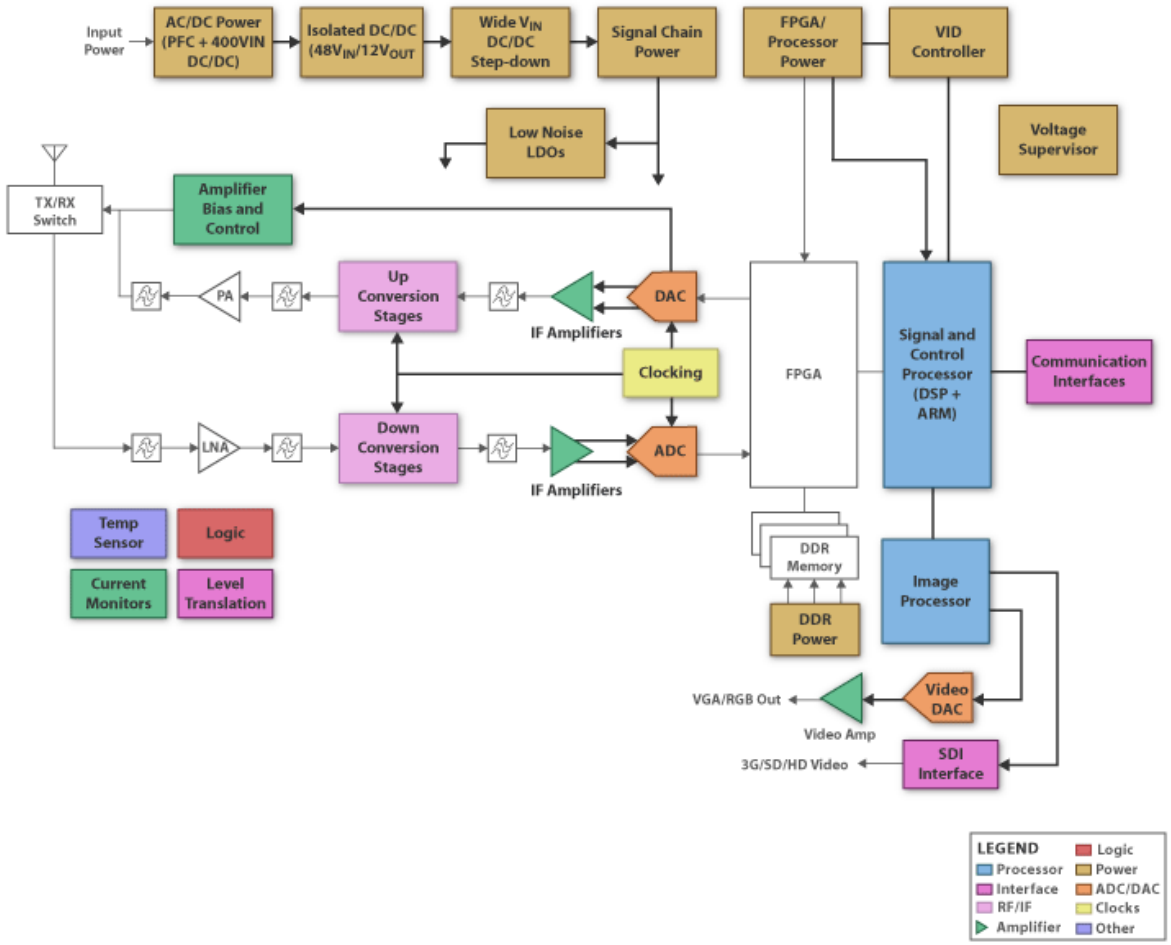

```
Fs = 10000;                     % sample frequency
Ts = 1/Fs;                      % period frequency
F1 = 100;                       % harmonic n°1 of input vector
F2 = 1000;                      % harmonic n°2 of input vector
XK_LENGTH = 2048;               % 2K/8Kb samples : length of input vector
t=0 : Ts : (XK_LENGTH-1)*Ts;    % time vector


xk = single(sin(2*pi*F1*t) + sin(2*pi*F2*t));
```

1.3. Exemple des applications Radar



Radar de défense aérienne GM 400 - THALES AIR SYSTEMS

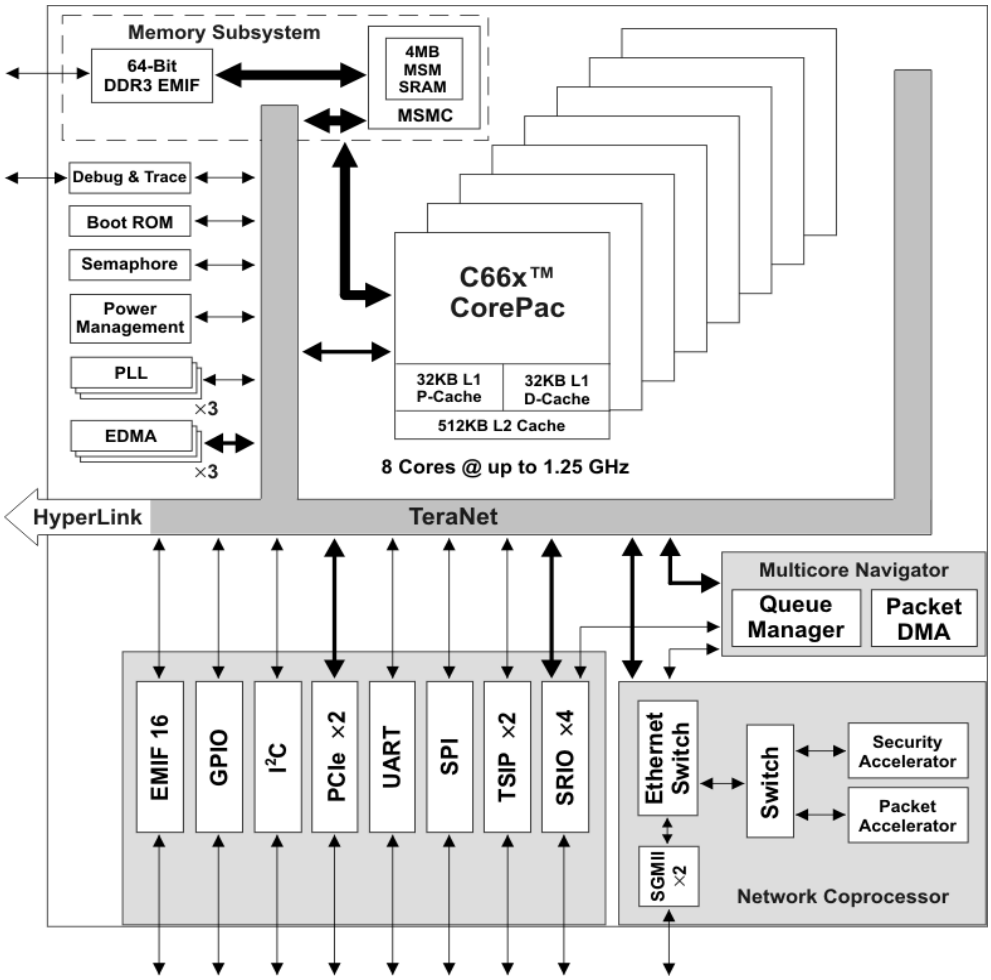
Afin de bien prendre conscience de l'intérêt de tel développement et phases d'optimisation, prenons l'exemple d'une application radar. Nous pouvons observer, ci-dessous, l'architecture matérielle typique d'un radar. Observons l'emplacement du processeur de traitement numérique du signal, dans le cas présent un DSP. Il faut savoir que d'autres familles de processeurs orientés calcul peuvent également remplir cette tâche (FPGA, MPPA, GPP voire GPU), chaque famille offrant son lot d'avantages et d'inconvénients.



Une chaîne numérique de traitement radar implémente également un produit scalaire comme celui étudié dans cette trame de travaux pratiques . A titre indicatif, sans optimisation l'algorithme du produit scalaire représente environ près de 40% du temps de traitement d'une chaîne Radar complète. Compte-tenu des contraintes temps réel excessivement lourdes imposées par ce type d'application, soit quelques Mo de données à traiter en quelques ms, nous pouvons pressentir tout l'intérêt de développer des bibliothèques de fonctions de calcul spécialisées pour l'architecture cible. Ce sera l'objectif premier de cet enseignement tout en respectant des méthodologies assurant un développement optimal.

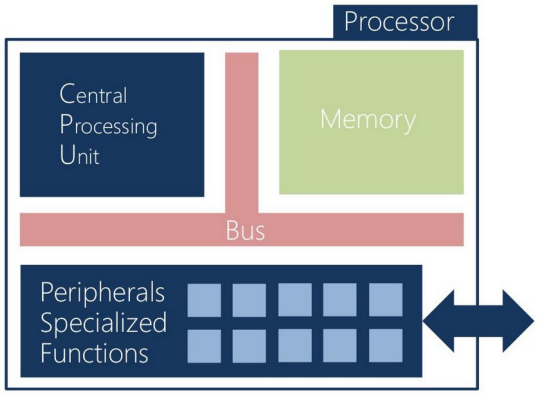
1.4. Architecture DSP VLIW C6600 de TI

Nos développements seront réalisés sur l'architecture DSP VLIW C6600 (Very Long Instruction Word) proposée par TI ou Texas Instruments (processeur utilisé par certains partenaires). Pour information, à notre époque la famille CPU C6000 de TI est l'architecture CPU leader sur le marché des processeurs DSP (Digital Signal Processor). Le processeur C6678 étudié en TP étant l'un des composants haut de gamme de la famille avec ses 8 cœurs vectoriels VLIW. Cette architecture propose quelques atouts assurant une grande flexibilité et permettant une bonne compréhension des architectures processeurs actuelles. Le processeur DSP TMS320C6678 offre les services matériels illustrés ci-dessous :



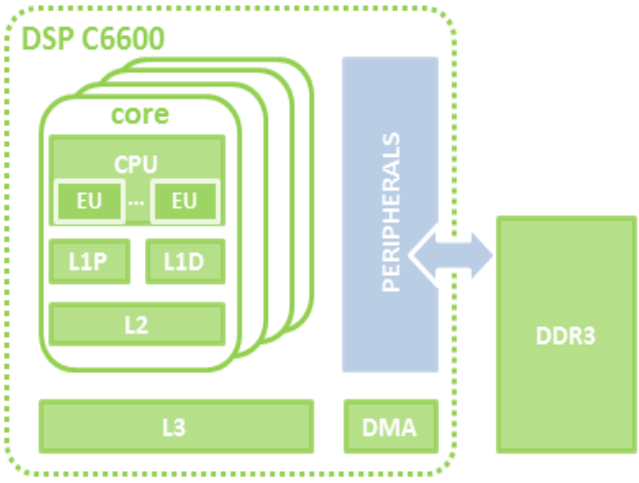
- 8 cœurs avec CPU vectoriels VLIW possédant chacun 32Ko caches L1P (L1-Program) et L1D (L1-Data), ainsi que 512Ko de cache L2 unifié (Program/Data)
- Chaque cœur peut être cadencé jusqu'à 1,4GHz
- Le niveau mémoire L3 unifié de 4Mo nommé MSM (Multi-core Shared Memory) est partagé entre cœurs
- Chaque niveau mémoire peut être configurable en cache ou en mémoire adressable SRAM permettant une architecture matérielle configurable en modèle mémoire uniforme UMA ou non-uniforme NUMA. Chose impossible sur processeur généraliste GPP Inte/AMD x86_64 par exemple.
- De la mémoire SDRAM DDR3 externe peut également être ajoutée sur circuit imprimé et est alors interconnectée via le périphérique d'interface EMIF (External Memory Interface).

En première année, nous avons découvert les bases du développement sur processeur numérique MCU (Micro Controller Unit) ou microcontrôleur. Ces processeurs étant généralistes et non spécialisés pour du calcul numérique, nous nous sommes donc assez longuement attardés sur les périphériques de communication (UART, I2C, etc) et d'interface (GPIO, ADC, etc) :



Cet enseignement est différent et doit être perçu comme une extension des compétences de première année. Cette année nous allons travailler sur machine fortement parallèle et nous nous efforcerons de comprendre puis d'exploiter au mieux les ressources matérielles proposées par notre processeur. Les périphériques d'interface ne seront donc pas vus, seule la partie traitement nous intéressera. Nous nous focaliserons sur :

- CPU vectoriel VLIW (pipelining software d'instructions et vectorisation des données)
- Mémoires L1/L2/L3 configurables en SRAM adressable ou en Cache et mémoire DDR
- Périphériques DMA (Direct Memory Access) de copie mémoire (copies sans passer par le CPU)



Même si l'exemple qui suit n'a que peu de sens, afin de bien comprendre les différences majeures entre les architectures de première année et de cette année, comparons les performances théoriques maximales des deux processeurs. Avec ces quelques chiffres, nous pouvons commencer à pressentir le potentiel pour du calcul numérique flottant de cette famille de processeur.

- Un MCU 8bits entier PIC18F27K40 de Microchip peut exécuter jusqu'à 16MIPS (soit 16 millions instructions entières 8bits par seconde)
- Un DSP VLIW 32bits flottant TMS320C6678 de Texas Instruments peut exécuter jusqu'à 22,4GFLOP/core (soit 22,4 Giga instructions flottantes 32bits en simple précision IEEE-754 par seconde et par cœur). Soit jusqu'à 179,2GFLOP pour le processeur complet grâce à ses 8 cœurs, le tout avec une horloge à 1,4GHz.

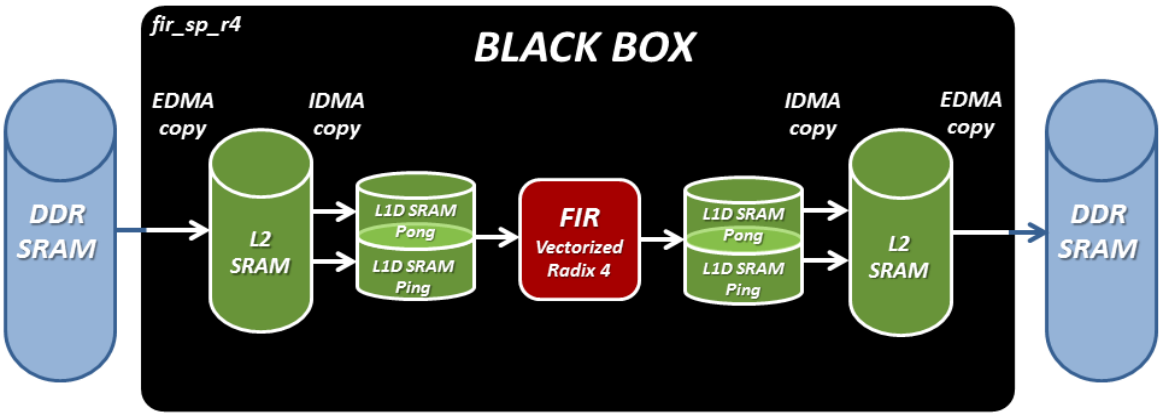
1.5. Objectifs pédagogiques

Nous venons de présenter un rapide tour d'horizon du potentiel de notre architecture processeur. Nous comprenons mieux le sens du nom de l'enseignement ''Architectures pour le calcul''. Beaucoup des **mécanismes d'accélération déterministes** (temps réel) présentés sont impossibles ou en tous cas moins performants ou moins déterministes sur architectures généralistes (GPP ou AP avec MMU/Cache). Néanmoins, une bonne compréhension des stratégies présentées précédemment vous assurera une adaptabilité forte pour des problématiques d'optimisation sur la plupart des architectures processeurs parallèles du marché (GPP, DSP, MPPA, GPU et MCU vectoriel). En milieu industriel, les ingénieurs bas niveau chargés du développement des bibliothèques spécialisées et système possèdent des compétences différentes des ingénieurs haut niveau assurant l'intégration logicielle et le développement des applications :

- **Développeur système temps réel bas niveau (système, driver ou algorithmique)** : ingénieur spécialisé dans les architectures matérielles processeurs et le développement de bibliothèques spécialisées. Maîtrise forte de l'architecture, des outils de développement, des langages C/ASM et du fonctionnement de la machine.

- **Développeur logiciel applicatif haut niveau** : concepteur et intégrateur applicatif haut niveau travaillant dans les couches hautes de l'application et utilisant les API et bibliothèques en mode boîtes noires (faible voire aucune idée de l'implémentation réelle bas niveau). Maîtrise faible de l'architecture et du fonctionnement de la machine.

```
#define XK_LENGTH 2048        /* 8kb vector*/
#define A_LENGTH 64
#define YK_LENGTH XK_LENGTH - A_LENGTH + 1

float32_t xk_sp[XK_LENGTH];
float32_t a_sp[A_LENGTH]={-0.00244444,0.00380928,...};
float32_t yk_sp[YK_LENGTH];

fir_sp_r4 (xk_sp, a_sp, yk_sp, A_LENGTH, YK_LENGTH);
```



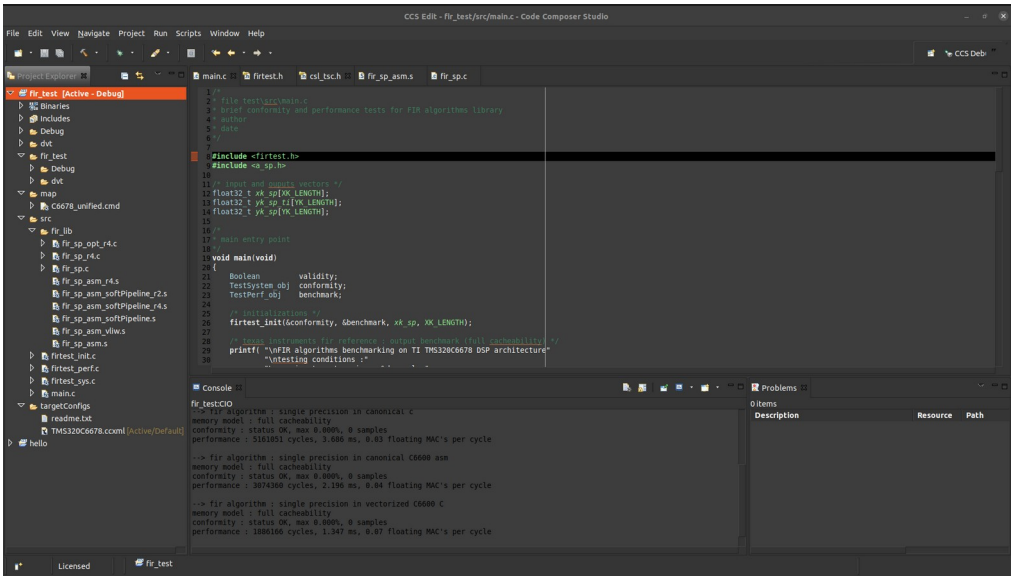
Cette trame d'enseignement vise donc à ouvrir les portes des stages et des métiers dédiés à l'optimisation et l'accélération algorithmique sur processeur spécialisé. Ces compétences, rares sur le marché, sont demandées par des entreprises ayant ce type de besoin spécifique (THALES, SAFRAN, ARIANE GROUP, sociétés en traitement d'image, cloud computing, etc). Par exemple, **un élève ingénieur réalisant la trame complète, aura développé en fin d'enseignement un seul et unique algorithme fir_sp_r4** (~30-40h de développement équivalent à une voire deux semaines entreprise). Cependant cet algorithme implémente certaines subtilités d'intégration :

- Copies montantes et descendantes des données entre DDR et L2 SRAM réalisées par DMA

- Copies montantes et descendantes des données entre L2 SRAM et L1D SRAM avec alternance des buffers de stockage (ping-pong) par IDMA interne à chaque Cœur

- Vectorisation C intrinsèque avec déroulement de boucle en base 4 sur CPU DSP VLIW

1.6. Outils de développement

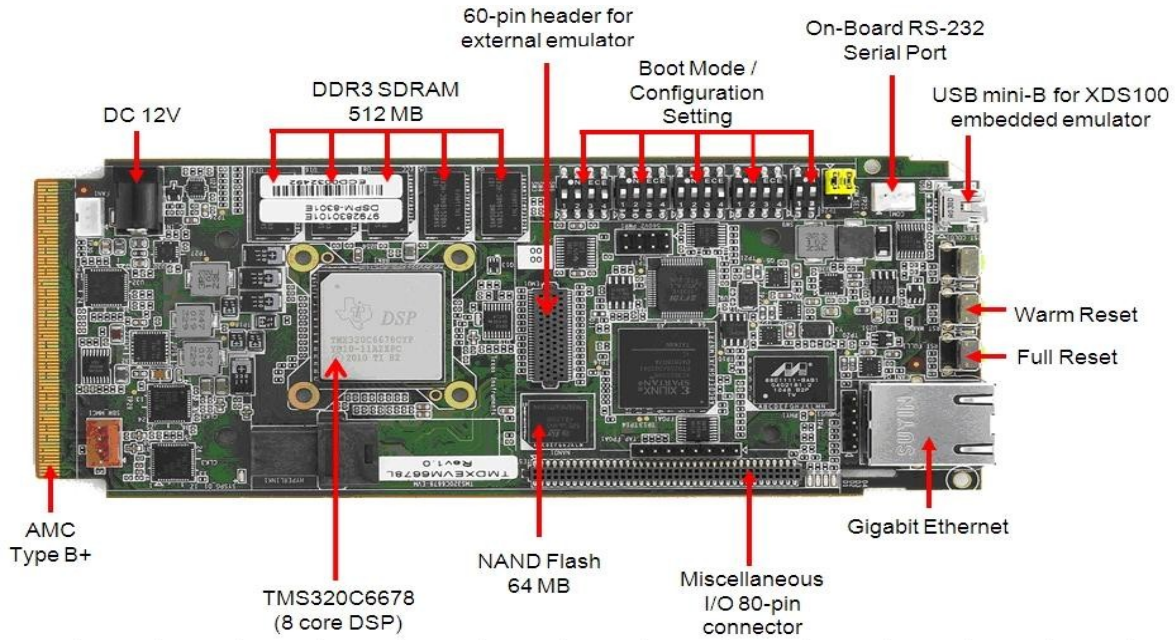


Nous développerons sous IDE CCS5.5 (Code Composer Studio) développé par Texas Instruments et basé sur un framework libre Eclipse. Les outils de développement et bibliothèques sont librement téléchargeables et installables depuis internet. Se référer à la page moodle de l'enseignement pour l'installation des outils de développement (section – OUTILS DE DEVELOPPEMENT):

https://foad.ensicaen.fr/course/view.php?id=117

La trame de travaux pratiques sera réalisée sur la plateforme de développement TMDXEVM6678L EVM (Evaluation Module) proposée par la société Advantec. Cette maquette d'évaluation embarque notamment une sonde de programmation USB XDS100 assurant la programmation et le débogage des applicatifs. Pour information, les outils de développement logiciel deviennent payants si nous souhaitons travailler avec des sondes d'émulation évoluées, telle que la sonde XDS560 également présente à l'école pour des phases de projet industriel. :

https://www.ti.com/tool/TMDSEVM6678



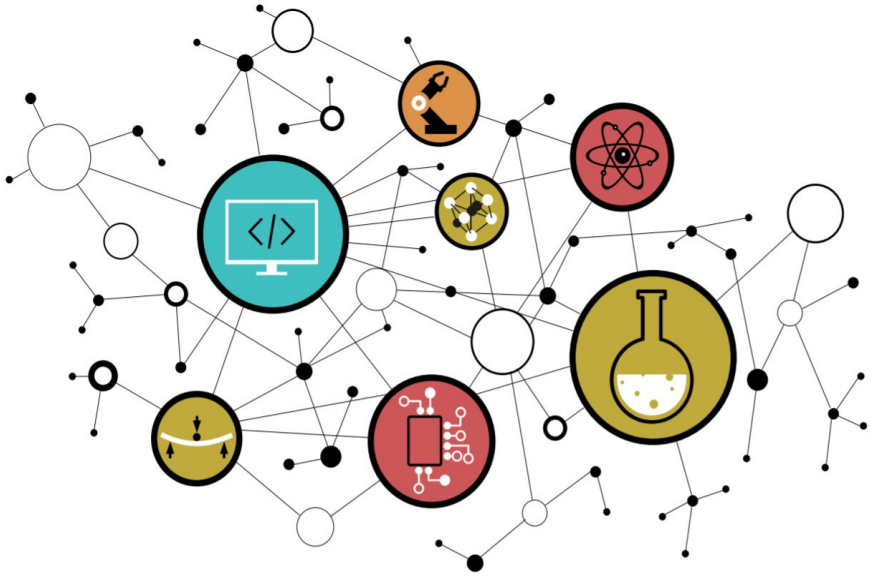
1.7. Benchmarking

ANALYSE COMPARATIVE

Algorithme vecteur d'entré 2K samples ou 8Ko	Architecture	Temps d'exécution ms / Cycles	Performances < 8 MACS max.	Temps de Développement heures	Observations et Limitations
DSPF_sp_fir_gen	Texas Instr. DSP VLIW C6600	0.031ms 43695cy	3.0	0	cf. documentation DSPLIB TI *nr and nh are a multiples of 4 and greaters than or equals to 4. x, h and r are double-word aligned. Interruptible code.*
	Intel GPP superscalar corei7 Haswell IA-64 A203/A201				

TRAVAUX PRATIQUES

PROGRAMMATION VECTORIELLE SUR DSP VLIW C6600



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

2. PROGRAMMATION VECTORIELLE SUR DSP VLIW C6600

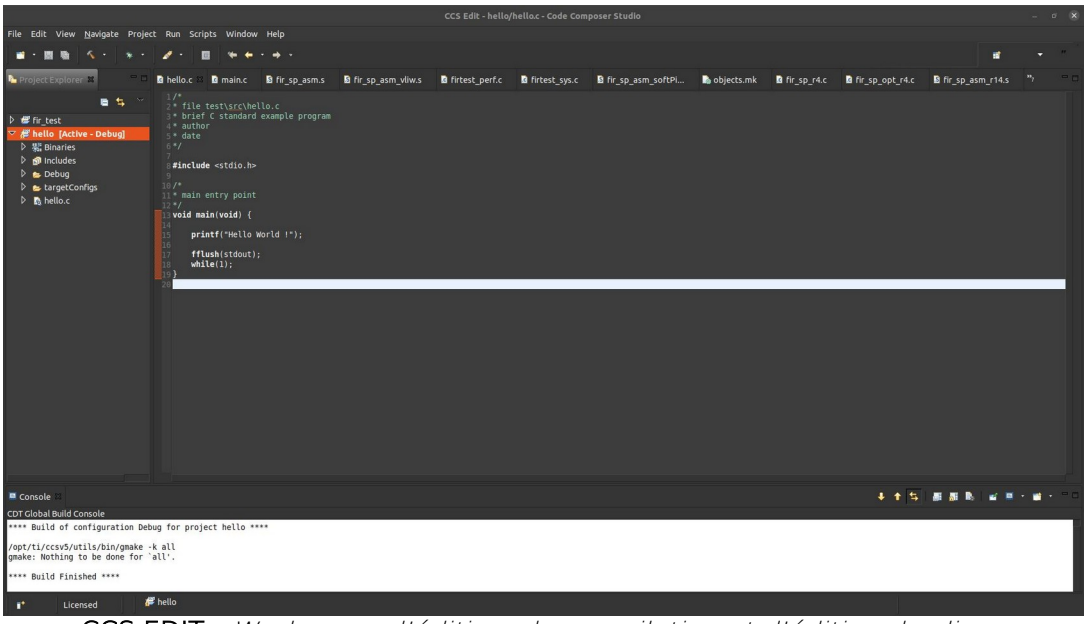
2.1. Création du projet de test
2.2. Analyse du programme de test
2.3. Assembleur canonique C6600
2.4. Assembleur VLIW C6600
2.5. Pipelining software en assembleur C6600
2.6. Vectorisation en assembleur C6600
2.7. Déroulement de boucle en C canonique
2.8. Vectorisation en C intrinsèque

2.1. Création du projet de test

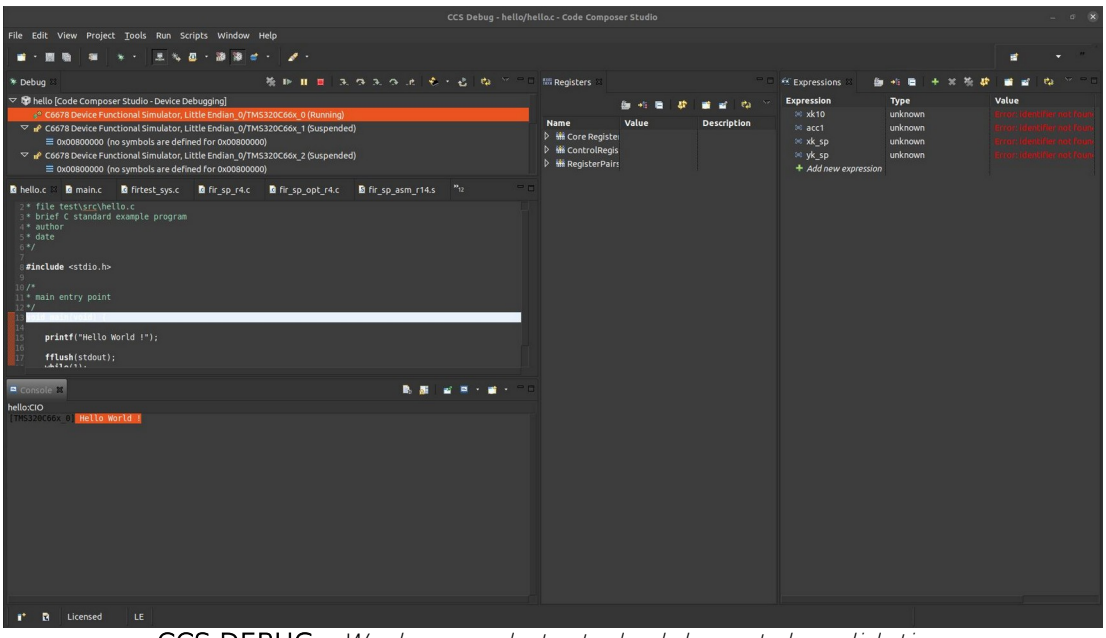
Parcourir puis s'aider des ressources en annexes durant les travaux pratiques (création de projet CCS, extraits de datasheet TMS320C6678, jeu d'instructions, etc). Toutes les documentations techniques utiles se trouvent dans le répertoire **tp/doc/datasheet**. Pour information, Texas instruments propose un cours en ligne gratuit sur sa famille CPU C6000 :

https://training.ti.com/c6000-embedded-design-workshop

- Créer un projet CCS nommé **hello** dans le répertoire **/disco/c6678/hello/** incluant le fichier source **/disco/c6678/hello/hello.c** (cf. image CCS EDIT ci-dessous). S'assurer de la bonne compilation et exécution du projet (5-10mn). Observer la sortie dans la fenêtre de Console (cf. image CCS DEBUG ci-dessous). Ce projet ne réalise qu'un printf et n'a pour objectif que de valider ''rapidement'' le bon fonctionnement de l'IDE CCS 5.5 (maintenant d'une génération et version ancienne). **Lire les 3 pages suivantes pour vous aiguiller**. Une fois fonctionnel et validé sur simulateur voire sur cible, ce projet peut être fermé et ne sera plus utilisé durant la totalité de la trame !

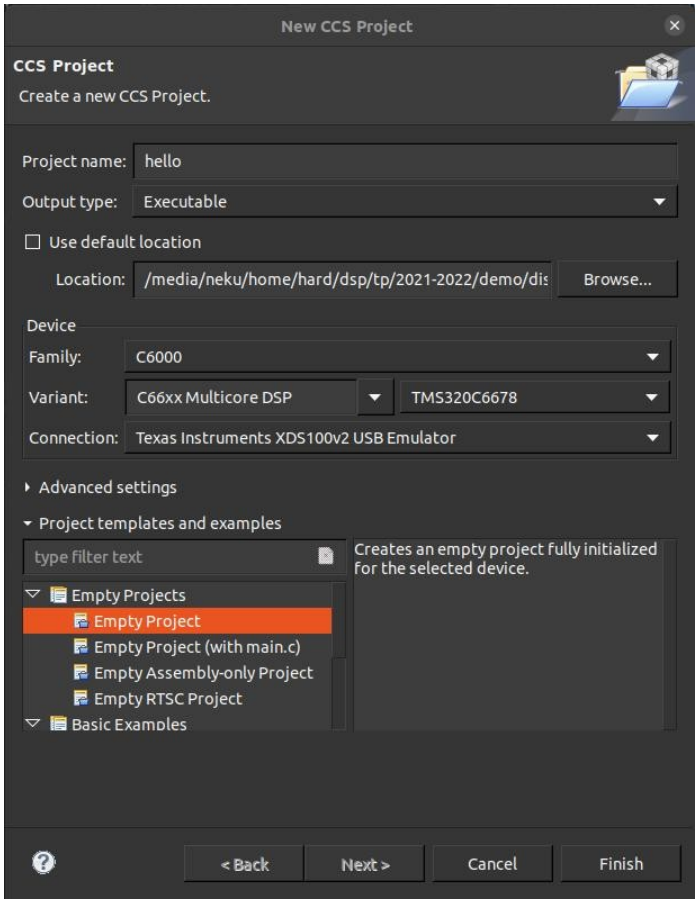


CCS EDIT - *Workspace d'édition, de compilation et d'édition des liens*

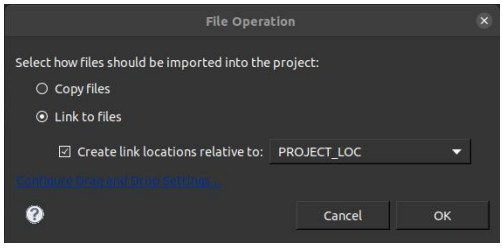


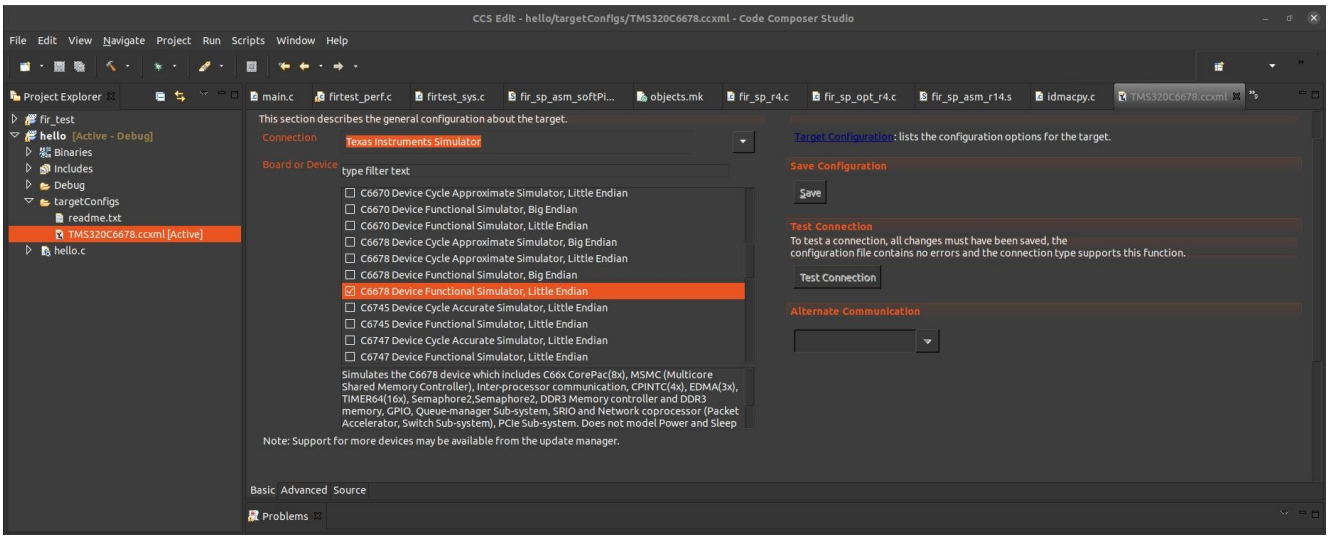
CCS DEBUG - *Workspace de test, de debug et de validation*

- File > New > CCS Project pour créer le projet CCS :
  - Project Name : hello
  - Output : Executable
  - Location : <your_path>/disco/c6678/hello
  - Family : C6000
  - Variant : C66xx Multicore DSP > TMS320C6678
  - Connection : Texas Instruments XDS100 v2 USB Emulator
  - Project templates and examples : Empy Projects > Empty Project
- Finish



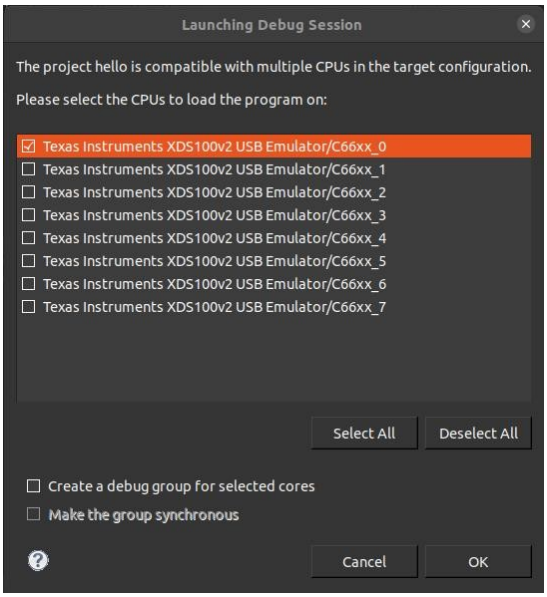
- [Clic droit] sur le nom du projet dans Project Explorer pour ajouter un fichier :
- Add Files...
- Ajouter hello.c et spécifier Link to File



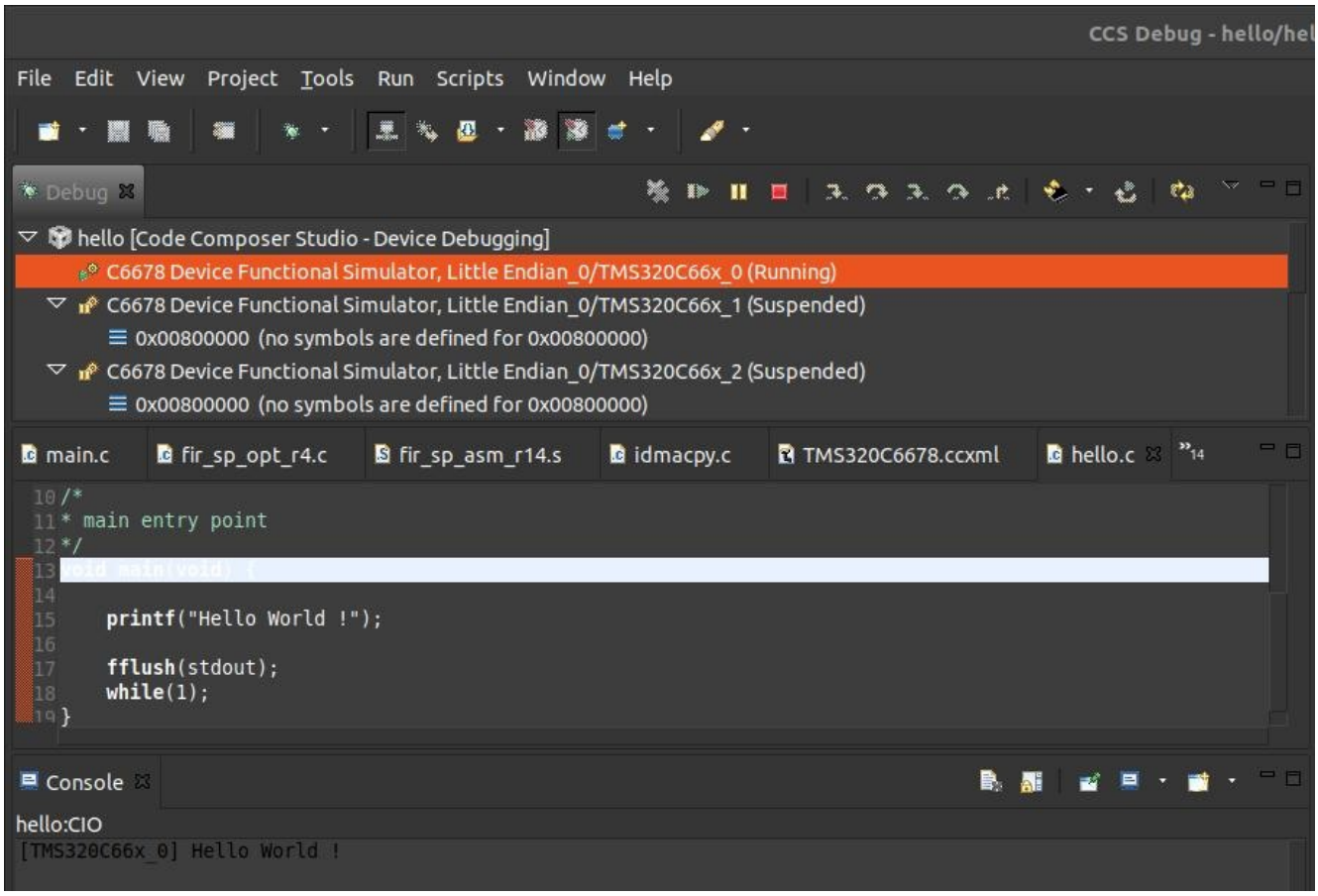


- Project Explorer > hello > targetConfigs > TMS320C6678.ccxml pour configurer le mode simulateur de CCS :
  - Connection : Texas Instruments Simulator
  - Board or Device : C6678 Device Functional Simulator, Little Endian
- Save

- icône Marteau pour compiler le projet (workspace CCS EDIT) :
- icône scarabée pour charger et tester l'exécutable dans le simulateur (workspace CCS DEBUG)
- Charger le programme sur le cœur n°0 seulement (et non sur les 8 cœurs du DSP):
  - Deselect All
  - [x] …./C66xx_0 pour sélectionner le cœur n°0
  - OK



- Tester le projet (icône flèche verte / play – workspace de debug)

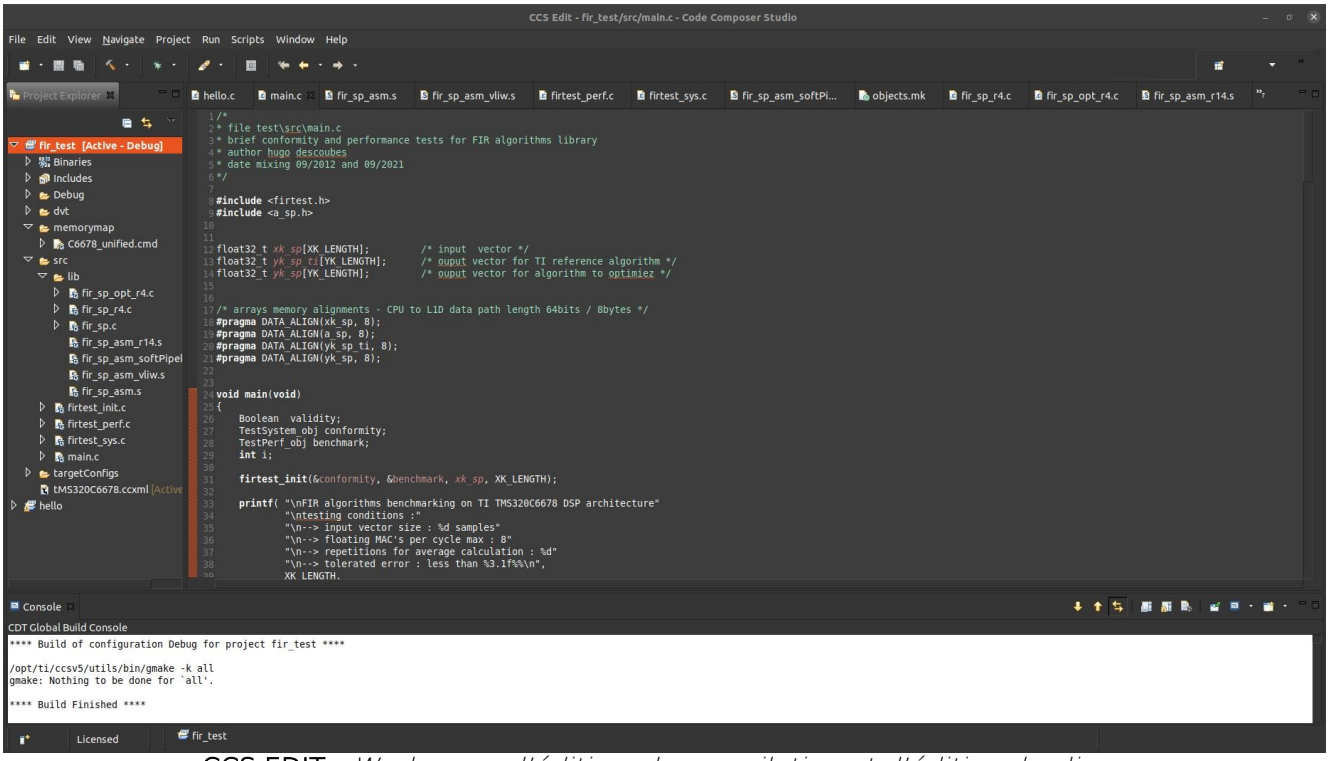


- Observer la sortie (fenêtre Console – workspace de debug)

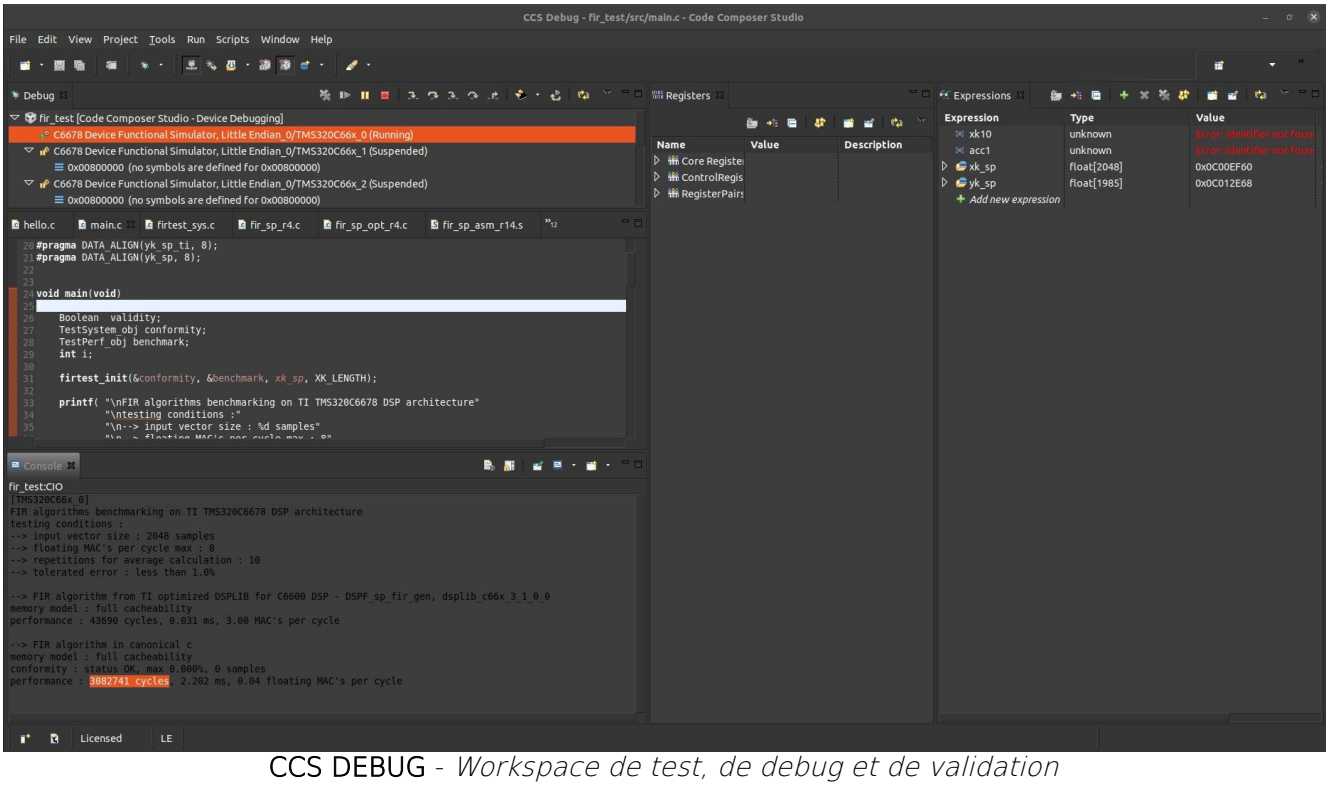
[TMS320C66x_0] Hello World Bro's!

- Arrêter la session de debug (icône carré rouge / stop – workspace de debug)
- Passer à la suite !

- Créer le projet de test nommé **fir_test** dans le répertoire **/disco/c6678/test/pjct/** . S'assurer de la bonne compilation (cf. image CCS EDIT ci-dessous) et exécution du projet. Observer la sortie dans la fenêtre de Console (cf. image CCS DEBUG ci-dessous). Inclure les fichiers sources suivants et **s'aider des 3 pages suivantes comme des 3 précédentes** :
  - *~/disco/c6678/test/src/*.c (à ajouter sous CCS dans src)*
  - *~/disco/c6678/firlib/src/*.c (à ajouter sous CCS dans un répertoire logique src/lib)*
  - *~/disco/c6678/firlib/src/*.s (à ajouter sous CCS dans un répertoire logique src/lib)*
  - *~/disco/c6678/test/map/C6678_unified.cmd (à ajouter sous CCS dans memorymap)*

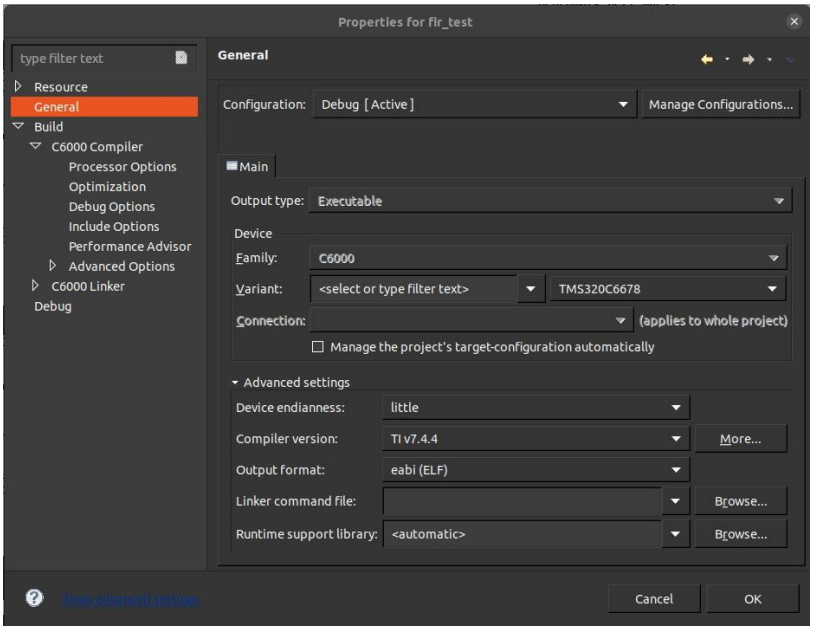


CCS EDIT - *Workspace d'édition, de compilation et d'édition des liens*

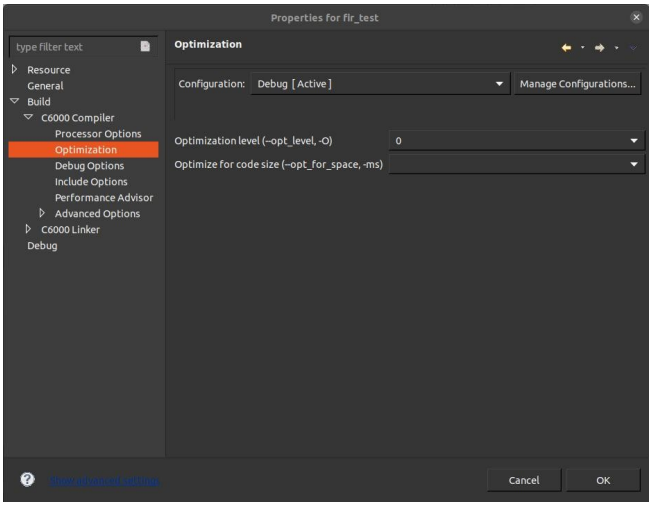


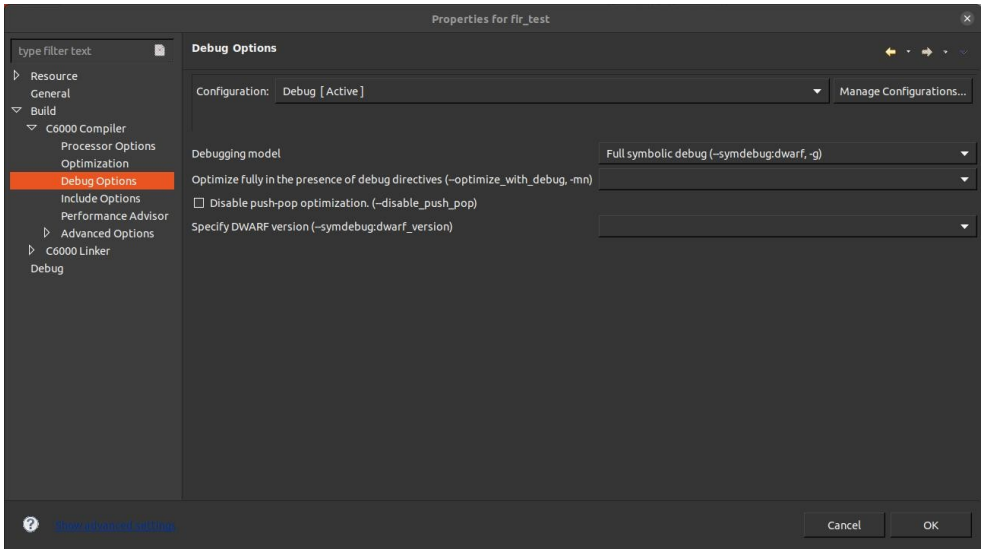
CCS DEBUG - *Workspace de test, de debug et de validation*

- Project Explorer > votre_projet_fir_test [Active – Debug] > clic droit > Properties
- Vérifier la configuration générale du projet



- Vérifier la configuration du projet en mode debug





- Vérifier l'inclusion des chemins pour permettre au compilateur de trouver les fichiers d'en-tête nécessaires au projet (C6000 compiler) :
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans /opt/ti sous GNU\Linux
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans C:\ti sous Windows

    - Fichiers d'en-tête applicatifs pour le projet de test

      <your_project_path>/disco/c6678/test/h

    - Fichiers d'en-tête applicatifs pour la bibliothèque projet FIRLIB

      <your_project_path>/disco/c6678/firlib/h

    - Fichiers d'en-tête de la bibliothèque CSL (Chip Support Library) de TI

      <depends_on_your_system>/ti/pdk_C6678_1_1_2_6/packages

    - Fichiers d'en-tête de la bibliothèque CSL (Chip Support Library) de TI

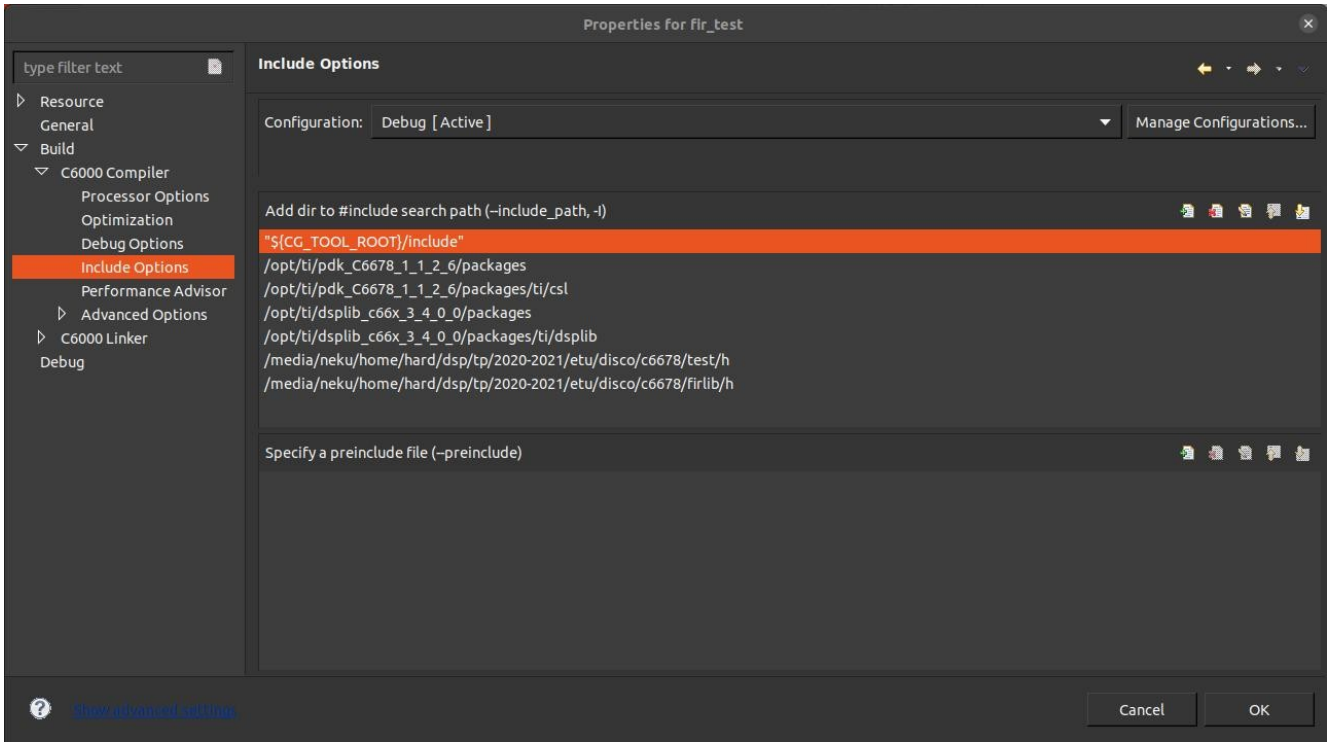
      <depends_on_your_system>/ti/pdk_C6678_1_1_2_6/packages/ti/csl

    - Fichiers d'en-tête de la bibliothèque DSPLIB (Digital Signal Processing Library) de TI

      <depends_on_your_system>/ti/dsplib_c66x_3_4_0_0/packages

    - Fichiers d'en-tête de la bibliothèque DSPLIB (Digital Signal Processing Library) de TI

      <depends_on_your_system>/ti/dsplib_c66x_3_4_0_0/packages/ti/dsplib



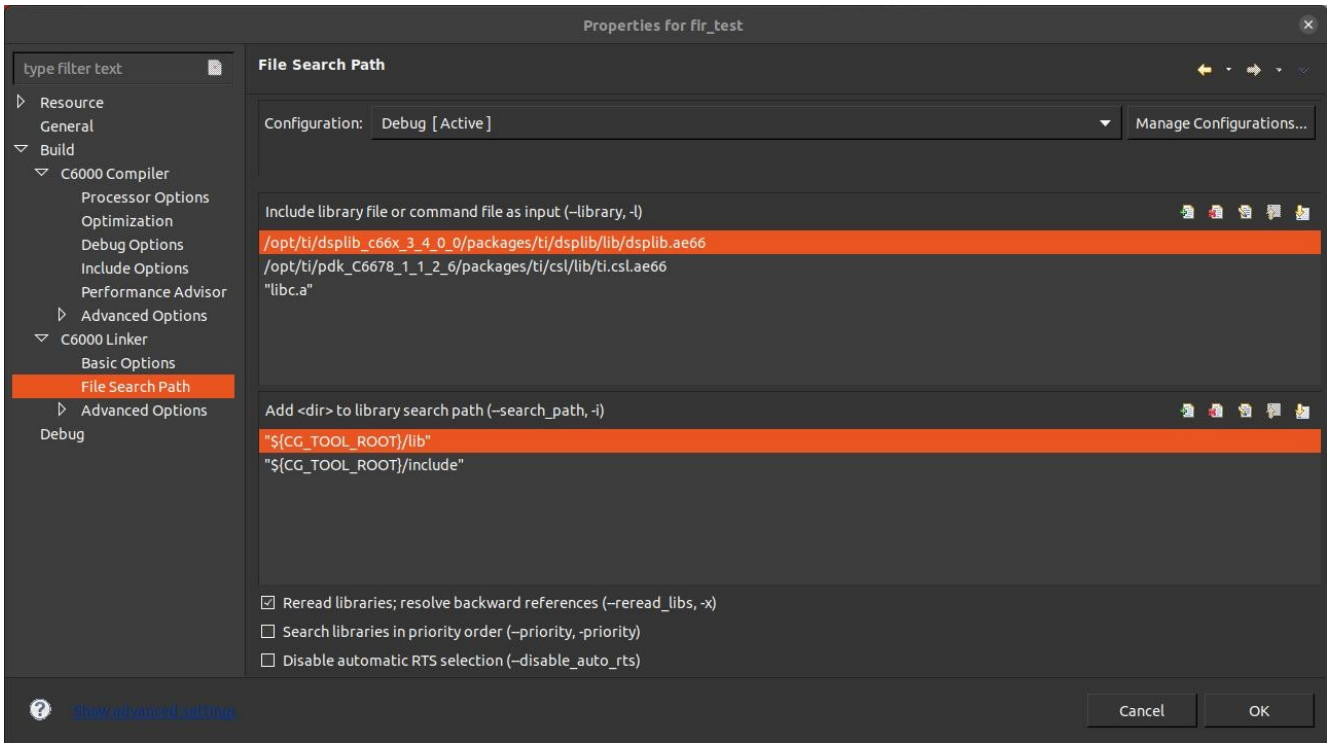
- Vérifier l'ajout des bibliothèques nécessaires au projet pour l'édition de liens (C6000 linker):
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans /opt/ti sous GNU\Linux
    - Attention, fichiers d'en-tête et bibliothèques CSL (Chip Support Library) et DSPLIB (DSP Library) de TI installées dans C:\ti sous Windows

    - Bibliothèque CSL (Chip Support Library) pour DSP TMS320C6678

<depends_on_your_system>/ti/pdk_C6678_1_1_2_6/packages/ti/csl/lib/ti.csl.ae66

    - Bibliothèque DSPLIB (Digital Signal Processing Library) pour DSP TMS320 famille C6600

<depends_on_your_system>/ti/dsplib_c66x_3_4_0_0/packages/ti/dsplib/lib/dsplib.ae66



- Cliquer sur OK pour quitter les propriétés du projet
- Compiler votre projet (icône scarabée – workspace d'édition)
- Tester le projet (icône flèche verte / play – workspace de debug)
- Observer la sortie (fenêtre Console – workspace de debug) ainsi que les mesures de performances :
    - Algorithme de référence TI présent dans la DSPLIB : ~43690 cy (soit ~0.031ms)
    - Algorithme de filtrage FIR en C canonique : ~3082742 cy (soit ~2.202ms)
- Arrêter la session de debug (icône carré rouge / stop – workspace de debug)
- Passer à la suite !

2.2. Analyse du programme de test

- Analyser le projet de test complet (10-15mn) puis répondre aux questions suivantes. Fichiers main.c, firtest.h, firtest_init.c, firtest_sys.c et firtest_perf.c ?

- Dans le fichier firtest.h, quel est le rôle des macros ci-dessous ?

```
#define TEST_FIR_SP             1
#define TEST_FIR_SP_R4          0
#define TEST_FIR_SP_OPT_R4      0
#define TEST_FIR_ASM            0
#define TEST_FIR_ASM_VLIW       0
#define TEST_FIR_ASM_PIPE       0
#define TEST_FIR_ASM_R4         0
```

- Dans le fichier firtest_init.c, que réalise la ligne ci-dessous ?

```
CSL_tscEnable();
```

- Dans le fichier firtest_perf.c, que réalisent les lignes ci-dessous ?

```
start = CSL_tscRead ();

…

end = CSL_tscRead ();
```

- Dans le fichier main.c (test de la fonction fir_sp), à quoi correspond le champ &fir_sp passé comme argument à la fonction firtest_perf ?

```
firtest_perf (&benchmark, UMA_L2CACHE_L1DCACHE, yk_sp, &fir_sp);
```

- Dans le fichier firtest_perf.c, que réalise la ligne ci-dessous ?

```
(*fir_fct) (xk_sp, a_sp, output, A_LENGTH, YK_LENGTH);
```

- Dans le fichier main.c, quels champs contiennent les variables structurées suivantes ? Quel est le rôle de chaque champ ?

```
TestSystem_obj conformity;
TestPerf_obj benchmark;
```

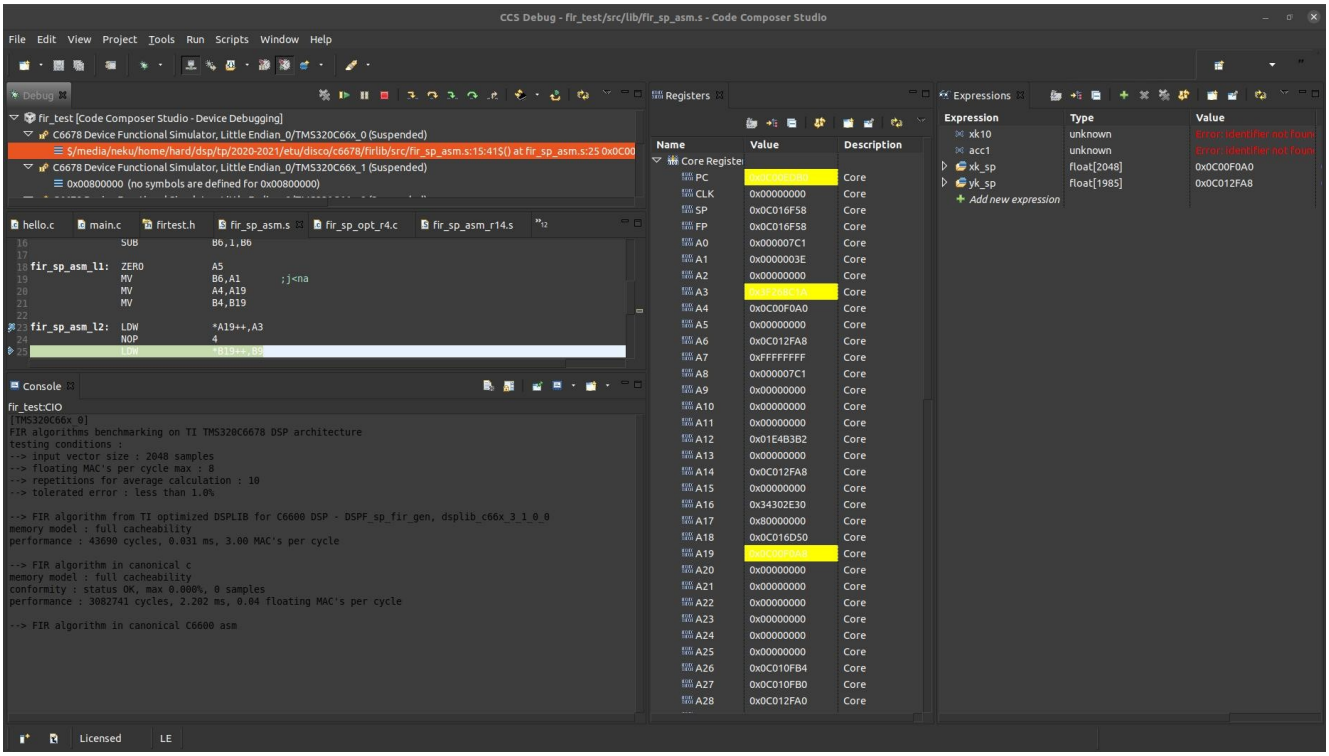
- En vous aidant du fichier firtest_sys.c voire d'autres fichiers, quelle est la marge d'erreur tolérée par la procédure de test de la conformité/validité de l'algorithme par défaut ?

2.3. Assembleur canonique C6600

- Dans le fichier **firtest.h**, mettre la macro TEST_FIR_ASM à 1

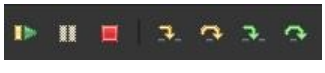
```
#define TEST_FIR_ASM    1
```

- En vous aidant du cours, implémenter le code de la fonction **fir_sp_asm** puis valider son fonctionnement. Cette fonction implémente l'**algorithme de filtrage FIR en assembleur canonique C6600 sans optimisation**. Ne pas oublier les delay slot (NOP) dans votre implémentation.



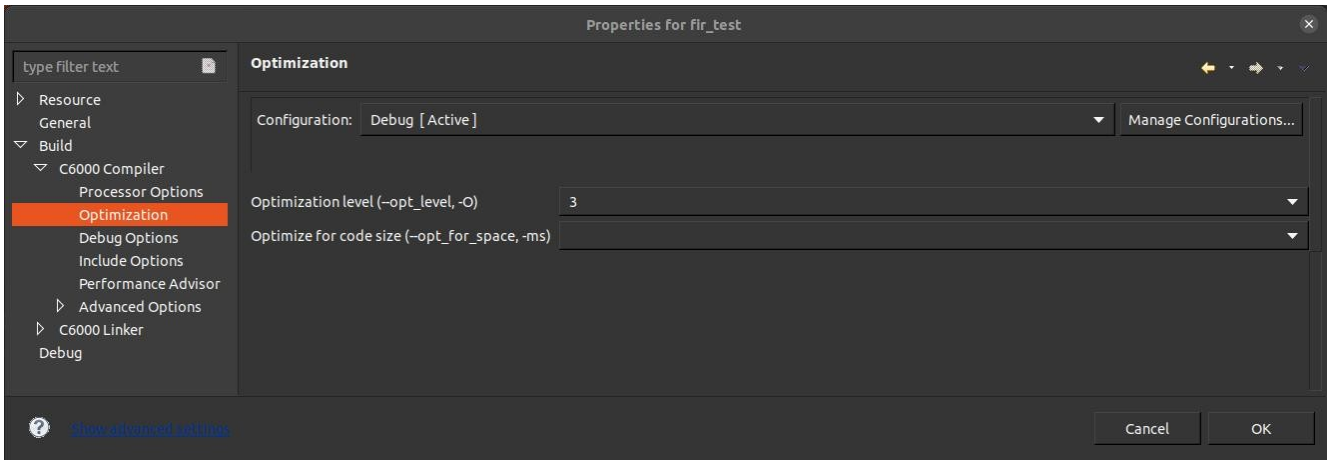
CCS DEBUG - *Workspace de test, de debug et de validation*

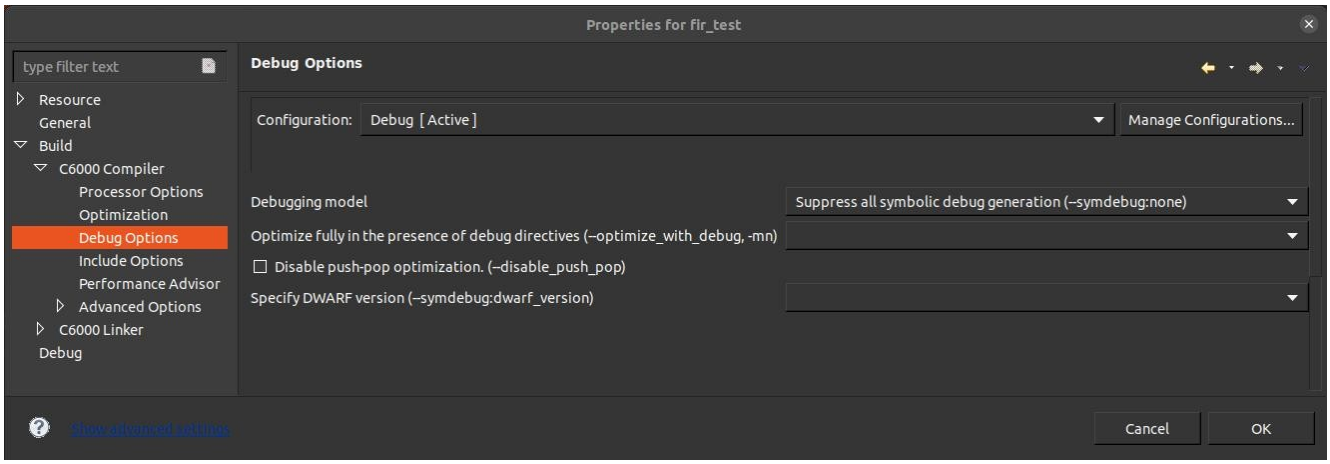
- Conseils pour Debugger le programme :
  - Travailler en mode **Debug**
  - Compiler et se placer dans l'espace de travail **CCS Debug** (Workspace de test, de debug et de validation)
  - Ouvrir la fenêtre **Registers > Core Registers** (cf. ci-dessus) afin d'observer le contenu des registres CPU (fenêtre accessible par Window > Show View). La fenêtre **Expressions** vous permet de voir le contenu des tableaux sous tests.
  - Placer des **points d'arrêts** dans votre programme sous test en réalisant un double clic dans la fenêtre d'édition sur le numéro d'une ligne (cf.ci-dessus – point bleu). La flèche représente l'adresse actuelle pointée par le PC (Program Counter – future instruction à exécuter) dans l'étage de FETCH du CPU (cf. ci-dessus).
  - Exécuter pas à pas le programme **CCS Debug > Run > Assembly Step** ou **F5** ou en utilisant le bandeau de contrôle du programme (cf. ci-dessous)



- Conseils d'implémentation :
  - Étape n°1 : s'assurer que l'appel et le retour de la procédure se déroulent bien puis vérifier les valeurs des paramètres d'entrée de la fonction.
  - Étape n°2 : implémenter la boucle vide avec la condition de sortie associée. S'assurer du bon nombre d'itérations et du fait de quitter celle-ci.
  - Étape n°3: dans la boucle, vérifier les premiers chargements de données de la mémoire vers le cœur ainsi que la validité des données pré-chargées.
  - Étape n°4 : écrire le code correspondant au produit scalaire dans le cœur de la boucle. Cette partie est plus complexe à tester et à valider.
  - Remarques : de façon générale, toujours tester les valeurs limites (condition de sortie de boucle, débordement de tableau, valeurs des pointeurs en mémoire …)

- Après validation en mode Debug, lancer une exécution en mode Release (cf. ci-dessous), compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.
  - Projet Explorer > votre_projet_fir_test [Active – Debug] > clic droit > Properties





- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

2.4. Assembleur VLIW C6600

Contrairement aux processeurs superscalaires, les architectures CPU VLIW (Very Long Instruction Word) et EPIC (Explicitly Parallel Instruction Computing) ont pour point commun d'avoir un code Out-Of-Order en mémoire (OOO ou dans le désordre) mais offrant une exécution in-order (exécution et sortie du pipeline dans l'ordre). Dans cet exercice, nous allons jouer sur ce point sans pour autant utiliser d'instructions vectorielles ni tenter d'avoir une utilisation optimale du pipeline logiciel d'instructions du CPU. En effet, nous n'utiliserons que des instructions scalaires (MPYSP, ADDSP, etc).

Rappelons le principe de ce type d'optimisation, uniquement applicable sur architecture CPU VLIW et EPIC (MPPA Kalray, DSP TI C6000, NXP TriMedia, DSP SHARC Analog Device, ST200 STMicroelectronics, Intel Itanium, etc). L'exemple qui suit présente un avancement de branche d'exécution sur architecture C6600. Le code est alors dans le désordre en mémoire et pourtant les deux files d'instructions ci-dessous réalisent le même traitement seulement l'une sortira le résultat plus rapidement :

Canonical C6600 assembly instructions in-order in memory
(processing time : 14 CPU cycles)

```
      MPYSP       A3,B9,A17
      NOP         3
      FADDSP      A17,A5,A5
      NOP         2
[A1]  SUB         A1,1,A1
[A1]  B           L2
      NOP         5
```

VLIW C6600 assembly instructions out-of-order in memory
(processing time : 7 CPU cycles)

```
      MPYSP     A3,B9,A17
||[A1] SUB      A1,1,A1
  [A1] B        L2
      NOP       2
      FADDSP    A17,A5,A5
      NOP       2
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_ASM_VLIW à 1

```
#define TEST_FIR_ASM_VLIW     1
```

- En vous aidant du cours, implémenter le code de la fonction fir_sp_asm_vliw puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en assembleur C6600 avec optimisation propre aux architectures VLIW (avancement de branches, exécutions d'instructions en parallèle, remplacement de NOP, etc). Ne pas oublier les delay slot (NOP) dans votre implémentation.

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

2.5. Pipelining software en assembleur C6600

Dans cet exercice, nous allons nous efforcer d'obtenir une utilisation optimale du pipeline logiciel d'instruction pour notre algorithme écrit en assembleur canonique. Cet exercice consiste à jouer sur le nombre d'unités d'exécution en essayant d'utiliser le CPU au maximum de son potentiel théorique. Dans notre cas, chaque cœur possédant 8 unités d'exécution (.M1, .M2, .L1, .L2, .S1, .S2, .D1 et .D2), toutes capables de travailler en parallèle. Nous chercherons à obtenir un maximum de 8 instructions exécutées par cycle CPU (notamment pour le code du cœur de l'algorithme). Nous allons donc nous intéresser au parallélisme d'instructions. Nous verrons le parallélisme des données à l'exécution dans les prochains exercices sur la vectorisation.

Pour notre algorithme, sans déroulement de boucle (vectorisation des données), le facteur optimal d'optimisation en terme d'accélération sera obtenu à travers cet exercice. Observons de façon graphique dans un tableau l'architecture du code à implémenter. Dans la table de programmation ci-dessous, vous trouverez une implémentation de la boucle interne, la boucle externe restant inchangée. Le prolog est exécuté une seule fois, la boucle kernel autant de fois qu'il y a d'itérations de boucles en retranchant la profondeur du prolog et l'epilog sera également exécuté qu'une seule fois. L'allocation et le choix des registres utilisés reste libre dans le cadre de cet exercice.

	Unités d'Exécution (occupation des pipelines hardware et software)							
	.D1	.M1	.S1	.L1	.L2	.S2	.M2	.D2
PROLOG	LDW							LDW
	NOP							NOP
	NOP							NOP
	NOP							NOP
	NOP							NOP
	LDW	MPYSP						LDW
	NOP	NOP						NOP
	NOP	NOP						NOP
	NOP	NOP						NOP
	NOP	NOP						NOP
KERNEL	LDW	MPYSP	BDEC	FADDSP				LDW
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
	NOP	NOP	NOP	NOP				NOP
EPILOG		MPYSP		FADDSP				
		NOP		NOP				
		NOP		NOP				
		NOP		NOP				
				FADDSP				
				NOP				
				NOP				

- Dans le fichier firtest.h, mettre la macro TEST_FIR_ASM_PIPE à 1

```
#define TEST_FIR_ASM_PIPE      1
```

- Implémenter le code de la fonction fir_sp_asm_softPipeline puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en assembleur C6600 avec optimisation propre aux architectures VLIW et usage optimal du pipeline software d'instructions du CPU (avancement de branche, exécution d'instructions en parallèle, remplacement de NOP, etc). Ne pas oublier les delay slot (NOP) dans votre implémentation.
    - Conseils d'implémentation :
        - Étape n°1 : la profondeur de la boucle kernel interne doit posséder une taille supérieure ou égale au nombre de cycles CPU nécessaires à l'exécution de l'instruction B (branch).
        - Étape n°2 : tester la condition de sortie de la boucle ainsi que le bon nombre d'itérations.
        - Étape n°3 : vérifier les données préchargées depuis la mémoire par les instructions de chargement sur plusieurs itérations.

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

- Sur CPU VLIW ou EPIC, les NOP's (delay slot) peuvent être vus comme des espaces libres dans lesquels peuvent être insérées des instructions à exécuter en parallèle. Dans une optique de déroulement de boucle, nous pourrons alors être amenés à charger de façon conséquente la boucle kernel actuellement sous exploitée.

- Sans nous en rendre compte, au fil des exercices en assembleur précédents et actuel nous nous efforçons d'entrer dans la tête de notre compilateur C. En effet, lorsque les options d'optimisation sont levées, il tente d'appliquer un maximum des techniques d'accélération que nous sommes en train de découvrir. Sans pour autant obtenir une implémentation optimale. Nous constaterons par la suite que si nous souhaitons obtenir des facteurs d'accélération optimaux après compilation en restant à l'étage du langage C, il nous faudra effectuer du refactoring de code ainsi qu'aiguiller au maximum la chaîne de compilation. Nous perdrons alors en portabilité de code.

2.6. Vectorisation en assembleur C6600

Nous allons nous intéresser au parallélisme des données en utilisant un maximum d'instructions vectorielles SIMD (Single Instruction Multiple Data) permettant un traitement des opérandes à l'étage assembleur par vecteur de données (2, 4, 8, etc selon la technologie du CPU) et non plus par données scalaires (opérandes traitées une à une). De plus, ceci nous permettra de minimiser l'usage du pipeline logiciel d'instructions et donc le nombre d'unités d'exécution utilisées en parallèle. Nous laisserons ainsi la place potentielle à d'autres instructions. Prenons l'exemple d'une même section de code avec des instructions scalaires et vectorielles. Dans les deux cas, les résultats calculés sont les mêmes et seront stockés dans les mêmes registres CPU. L'implémentation vectorielle est seulement plus rapide.

C6600 scalars assembly instructions
(processing time: 18 CPU cycles)

```
LDW         *A19++, A25
NOP         4
LDW         *A19++, A24
NOP         4
MPYSP       A25, B23, B5
NOP         3
MPYSP       A24, B22, B4
NOP         3
```

C6600 vectorials assembly instructions
(processing time : 9 CPU cycles)

```
LDDW        *A19++, A25:A24
NOP         4
DMPYSP      A25:A24, B23:B22, B5:B4
NOP         3
```

Observons ci-dessous un déroulement d'un facteur 4 (radix 4) de la boucle interne de l'algorithme. La boucle externe reste inchangée et calcule les échantillons de sortie un à un. Dans cet exercice, nous allons implémenter cet algorithme en assembleur C6600 (partie à vectoriser en *gras et italique*). Nous utiliserons les instructions vectorielles suivantes mais sans software pipelining ni optimisation VLIW (ASM C6600 canonique) : LDDW, LDNDW, DMPYSP et DADDSP

```
void  fir_sp_r14 ( const float * restrict xk,
                   const float * restrict a,
                   float * restrict yk,
                   int na,
                   int nyk)
{
      int i, j;
      float xk0, xk1, xk2, xk3;
      float a0, a1, a2, a3;
      float acc0, acc1, acc2, acc3;

      for (i=0; i<nyk; i++) {
            acc0 = 0.0;
            acc1 = 0.0;
            acc2 = 0.0;
            acc3 = 0.0;

            for (j=0; j<na; j+=4){
                  a0 = a[j];
                  a1 = a[j+1];
                  a2 = a[j+2];
                  a3 = a[j+3];

                  xk0 = xk[j+i];
                  xk1 = xk[j+i+1];
                  xk2 = xk[j+i+2];
                  xk3 = xk[j+i+3];

                  acc0 += a0*xk0;
                  acc1 += a1*xk1;
                  acc2 += a2*xk2;
                  acc3 += a3*xk3;
            }
            yk[i] = acc0 + acc1 + acc2 + acc3;
      }
}
```

Avant de développer cet algorithme, nous allons analyser le code de base fourni dans la trame pour la fonction fir_sp_asm_r14. Nous le constaterons par nous même par la suite, mais la programmation vectorielle peut être très gourmande en ressources de stockage interne au CPU. Nous allons donc potentiellement avoir besoin de ''beaucoup'' de registres CPU.

```
save_context      .macro      rsp
; save core working registers context on the top of stack
                  MV                B15,rsp     ; save Stack Pointer
                  STDW              B15:B14,*rsp--[1]
                  STDW              B13:B12,*rsp--[1]
                  STDW              B11:B10,*rsp--[1]
                  STDW              A15:A14,*rsp--[1]
                  STDW              A13:A12,*rsp--[1]
                  STDW              A11:A10,*rsp--[1]
                  MVC               ILC,B15
                  MVC               RILC,B14
                  STDW              B15:B14,*rsp--[1]
                  .endm

restore_context   .macro      rsp
; restore core working registers context from the top of stack
                  LDDW              *++rsp[1],B15:B14
                  MVC               B14,RILC
                  MVC               B15,ILC
                  LDDW              *++rsp[1],A11:A10
                  LDDW              *++rsp[1],A13:A12
                  LDDW              *++rsp[1],A15:A14
                  LDDW              *++rsp[1],B11:B10
                  LDDW              *++rsp[1],B13:B12
                  LDDW              *++rsp[1],B15:B14
                  MV                rsp,B15     ; restore Stack Pointer
                  NOP               3
                  .endm

fir_sp_asm_r14:   save_context      A3

                  ; user code

                  restore_context   A3
                  B           B3
                  NOP         5
```

- Le compilateur C6600 utilise certains registres CPU pour des usages spécifiques liés au langage C (arguments de fonction, adresse et valeur de retour d'une fonction, etc). En vous aidant de la documentation technique dédiée aux mécanismes d'optimisation sur architectures C6000, préciser les registres utilisés par défaut pour les usages suivants (s'aider de la documentation mentionnée ci-dessous) :
    - Arguments de fonction ?
    - Adresse et valeur de retour d'une fonction ?
    - Pointeur SP (Stack Pointer – cf. cours Archi. Ordi. 1A) ?
    - Pointeur FP (Frame Pointer appelé BP ou Base Pointer) ?
- Documentation technique support :
    - opt/tp/doc/datasheet
    - datasheet - optimizing compiler – spru187v.pdf
    - Chapter 7.3 Register Conventions
    - Table 7-2. Register Usage

- La directive préprocesseur assembleur .macro s'utilise comme une macro fonction en langage C (macro avec paramètres). Expliquer le rôle de ces macros dans le programme (s'aider de la documentation mentionnée ci-dessous) ?
    - Documentation technique support :
        - opt/tp/doc/datasheet
        - datasheet - assembly tools - spru186x.pdf
        - Chapter 6 Macro Language Description
        - 6-2. Defining Macros

- Durant l'implémentation de notre algorithme assembleur vectorisé en base 4 (radix 4), quels sont les 5 registres de travail du CPU que nous ne devons en aucun cas utiliser durant la totalité du traitement de la fonction et pourquoi ? Préciser leurs usages spécifiques

- En quoi diffère l'instruction LDDW de l'instruction LDNDW (s'aider de la documentation mentionnée ci-dessous) ?
    - Documentation technique support :
        - opt/tp/doc/datasheet
        - datasheet - instruction set and cpu - sprugh7
        - Chapter 4 Instructions Descriptions
        - *Chercher les documentations techniques des 2 instructions …*
        - *La page suivante présente succinctement ce qu'est un alignement mémoire !*

Pour information, le concept d'alignement mémoire des données n'existe que sur processeur à CPU vectoriel et donc des processeurs capables de manipuler les données (lecture/écriture depuis le cache L1D) par paquet (2, 4, 8, etc octets). Prenons ci-dessous un exemple de deux déclarations de deux variables, les premières non-alignées en mémoire et les secondes explicitement alignées modulo 8 octets. Comme pour notre CPU C6600, nous supposerons que les bus entre CPU DSP C6xx (étage d'exécution unités .D1 et .D2) et le niveau mémoire L1D possèdent une taille de 64bits ou 8o (cf. image ci-dessous) :

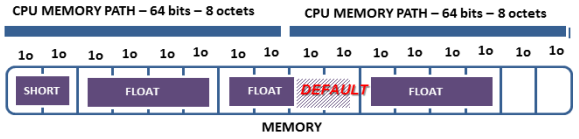
Non Aligned pointers NOK

```
short foo ;
float bar[3] ;
```

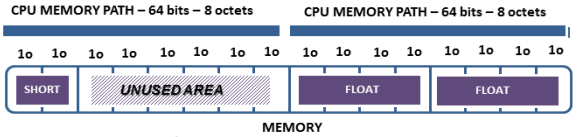
Aligned pointers OK

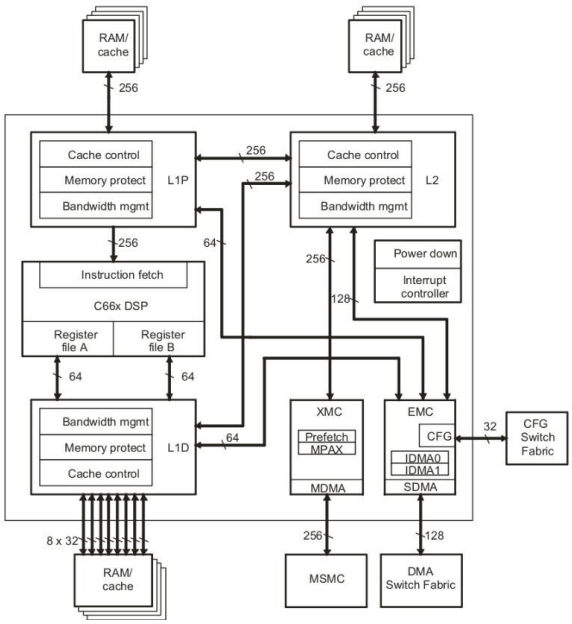
```
short foo ;
float bar[3] ;

#pragma DATA_ALIGN(foo, 8);
#pragma DATA_ALIGN(bar, 8);
```

NOK



OK





Vous constaterez que la chaîne de compilation effectue à votre place des alignements mémoire. Vous ne rencontrerez ce type d'optimisation potentielle que sur processeur à CPU vectoriel. Ceci se manifeste par des espaces mémoires vides de quelques octets présents entre vos différentes allocations de structures de données en mémoire. Tout pointeur aligné, possède une valeur multiple de l'alignement réalisé et peut ouvrir l'accès au compilateur à l'utilisation d'instructions de chargement et de sauvegarde mémoire plus rapide en temps d'exécution.

Il s'agit de mécanismes d'optimisation matérielles permettant au CPU de minimiser les accès mémoire aux données. Dans une optique d'optimisation, ceci peut éviter des défauts d'alignement mémoire (programmation vectorielle) et l'utilisation d'instructions dédiées à la lecture/écriture de données alignées (souvent plus lentes). La taille de ces alignements mémoire est le plus souvent liée à la taille d'un mot ''vectoriel'' CPU. D'une architecture CPU à une autre, cette taille diffère. Sur architecture vectorielle x86/x64 récente, un mot CPU fait 128bits (16o) depuis l'arrivée des extensions SIMD SSE (chemins/bus larges vers la mémoire donnée L1D). Les alignements mémoire de structures de données se feront donc le plus souvent via des adresses de valeurs multiples de 16 (modulo 16o). En résumé, la taille d'un mot vectoriel CPU dépend :

- Taille des path vers la mémoire donnée de niveau 1 ou L1D (le plus souvent un cache) : Exemple de la famille coreiX sandyBridge, 128bits vers le cache programme L1P, 2x128bits load et 1x128bits store vers le cache donnée L1D
- Taille des lignes de cache (tailles multiples d'un mot vectoriel CPU)

- Dans le fichier firtest.h, mettre la macro TEST_FIR_ASM_R4 à 1

```
#define TEST_FIR_ASM_R4 1
```

- Implémenter le code de la fonction fir_sp_asm_r14 puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en assembleur C6600 sans optimisation propre aux architectures VLIW mais avec usage d'instructions vectorielles C6600. Ne pas oublier les delay slot (NOP) dans votre implémentation. Utiliser les instructions suivantes :
    - LDDW
    - LDNDW
    - DMPYSP
    - DADDSP
- Quelle limitation d'usage amène l'implémentation de notre algorithme de filtrage ?

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

Voilà, à ce stade là de la formation en systèmes embarqués, vous devez être apte à porter un esprit critique sur les performances annoncées d'un processeur et ne plus regarder que sa seule fréquence de travail. Un processeur, même mono-cœur, cadencé à 500MHz peut très bien offrir des performances supérieures à un processeur multi-coeurs cadencé à 2GHz. Pour une architecture mono-cœur voire multi-coeurs. Il vous faut notamment tenir compte :

- de sa faculté à traiter des instructions en parallèle (nombre d'unités d'exécution pouvant travailler simultanément, largeur des bus entre cache L1P et étage de FETCH du CPU, etc)
- de sa faculté à traiter les données en parallèle (largeur des bus cache L1D et étage d'exécution du CPU, largeur des registres de travail vectoriels du CPU, jeu d'instructions vectorielles, etc)
- de la technologie de son pipeline matériel (superscalaire, VLIW ou EPIC), voire pour les plus curieux, des stratégies d'accélération interne au pipeline (le plus souvent aux étages de décodage et d'exécution)

2.7. Déroulement de boucle en C canonique

A partir de maintenant, la totalité de nos développements se feront en langage C. Nous découvrirons dans un premier temps des stratégies simples d'optimisation portables et applicables à beaucoup de processeurs vectoriels. Nous conclurons par une stratégie d'accélération optimale pour notre architecture mais amenant son lot d'inconvénients.

Le principe du déroulement de boucle consiste, sans mécanisme d'optimisation particulier, de ré-implémenter en C canonique notre algorithme en doublant/quadruplant/etc le nombre de chargement mémoire, d'opérations arithmétiques, etc dans nos boucles. Prenons un exemple ci-dessous de déroulement de boucle d'un facteur 2, souvent nommé unrolling radix 2 ou déroulement en base 2 :

Canonical C

```
void fir_sp ( const float* restrict xk,
              const float * restrict a,
              float * restrict pYk,
              int nbCoeff)
{
      int i;
      float yk_tmp = 0.0f;


      for (i=0; i<nbCoeff; i++){

            yk_tmp += a[i]*xk[i];

      }
      *pYk = yk_tmp;
}
```

Canonical C with unrolling radix 2

```
void fir_sp_r2 ( const float* restrict xk,
                 const float* restrict a,
                 float * restrict pYk,
                 int nbCoeff)
{
      int i;
      float xk_tmp1, xk_tmp2;
      float a_tmp1, a_tmp2;
      float mul1, mul2;
      float add1 = 0.0f, add2 = 0.0f;

      for (i=0; i<nbCoeff; i+=2){

            xk_tmp1 = xk[i];
            xk_tmp2 = xk[i+1];
            a_tmp1 = a[i];
            a_tmp2 = a[i+1];

            mul1 = a_tmp1 * xk_tmp1;
            mul2 = a_tmp2 * xk_tmp2;

            add1 += mul1;
            add2 += mul2;
      }
      *pYk = add1 + add2;
}
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_SP_R4 à 1

```
#define TEST_FIR_SP_R4  1
```

- Implémenter le code de la fonction fir_sp_r4 puis valider son fonctionnement. Cette fonction implémente l'algorithme de filtrage FIR en C canonique avec déroulement des 2 boucles interne et externe d'un facteur 4 (radix 4).
- Quelle limitation d'usage amène l'implémentation de notre algorithme de filtrage ?
- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.
- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

2.8. Vectorisation en C intrinsèque

Durant cet exercice, nous allons réutiliser la totalité de nos anciens acquis afin d'obtenir une implémentation optimale de notre algorithme de filtrage. Nous allons comprendre que les étapes précédentes, même si contre-performantes au seul regard des Benchmarking, étaient nécessaires à une bonne compréhension des stratégies présentées ci-dessous. Afin, d'exploiter au mieux l'architecture matérielle parallèle, vectorielle, VLIW de chaque cœur, sans pour autant descendre à l'étage assembleur qui peut être très chronophage en temps de développement, nous allons devoir aiguiller au maximum notre chaîne de compilation en effectuant du refactoring de code.

Découvrons quelques une des techniques les plus efficaces pour maîtriser la totalité du potentiel d'optimisation de notre chaîne de compilation (s'aider de la documentation mentionnée ci-dessous). Lire également la page suivante :

- opt/tp/doc/datasheet
- datasheet - optimizing compiler – spru187v.pdf
- Chapter 7.5.6 Using intrinsics to Access Assembly Langage Statements

Examples of C intrinsics functions for C6600 compiler and architecture	**C6600 assembly equivalence (automatic register allocation by compiler)**

// __float2_t container type for 2 floats
 	
`__float2_t data10;`
`const float ldData[2] = {0.0f,1.0f};`
 	
`data10 = _amemd8_const(&ldData[0]);`
	
	`; data10 = A1:A0`
	`; &ldData[0] = ldData = A2`
	
	`LDDW` `*A2, A1:A0`

`__float2_t data10, data32;`
`__float2_t result31_20;`
 	
`result31_20 = _daddsp(data10, data32);`
	`; data10 = A1:A0, data32 = A3:A2,`
	`; result31_20 = A11:A10`
	
	`DADDSP` `A1:A0, A3:A2, A11:A10`

`__float2_t data10, data32;`
`__float2_t result31_20;`
 	
`result31_20 = _dmpysp(data10, data32);`
	`; data10 = A1:A0, data32 = A3:A2,`
	`; result31_20 = A11:A10`
	
	`DMPYSP` `A1:A0, A3:A2, A11:A10`

`__float2_t data10;`
`__float2_t data32;`
`__float2_t data21;`
 	
`data21 = _ftod(_lof(data32), _hif(data10));`
	`; data32 = A3:A2`
	`; data10 = A1:A0`
	`; data21= A11:A10`
	
	`MV` `A2, A11`
	`MV` `A1, A10`

`__float2_t data10;`
`float stData[2];`
 	
`_amemd8(&stData[0]) = data10;`
	`; data10 = A1:A0`
	`; &stData[0] = stData = A2`
	
	`STDW` `A1:A0, *A2`

Les fonctions intrinsèques ou intrinsics sont dépendantes de l'architecture CPU cible. Elles existent principalement sur processeur vectoriel. D'une architecture CPU et donc d'une chaîne de compilation à une autre, la prise en main de la nouvelle API (Application Programming Interface) intrinsics est nécessaire.

A l'image d'une fonction inline, une fonction intrinsèque force le compilateur à générer automatiquement une séquence d'instructions. Cependant, contrairement aux fonctions inlines, le compilateur possède une connaissance poussée de la fonction intrinsèque qui lui assure une insertion optimale pour un contexte donné.

Observons un exemple de comparaison entre une implémentation C canonique et une solution avec fonctions intrinsèques. __float2_t est un type conteneur (container type) sur 64bits et pouvant contenir 2 flottants en simple précision sur 32 bits :

Canonical C example	Example of C intrinsic function

`float data[2] = {0.0f, 1.0f};`
`float data0, data1;`
 	
`// 2 x LDW assembly instructions`
`data0 = data[0];`
`data1 = data[1];`
	`const float data[2] = {0.0f, 1.0f};`
	`__float2_t data10;`
	
	`// 1 x LDDW assembly instruction`
	`data10 = _amemd8_const(&data[0]);`

- Assertion : l'objectif de cette technique est de donner un maximum d'informations (écrites et garanties par le développeur) sur une variable pour la chaîne de compilation afin de l'aiguiller dans ses phases d'optimisation futures. Par exemple, nous pouvons spécifier à la toolchain sa valeur minimale, si la valeur d'une variable est toujours multiple d'une autre valeur, l'alignement mémoire de données, etc

Example of program with assertions	Description

`short i, size=4;`
`const float data[4] = {0.0f, 1.0f, 2.0f, 3.0f};`
`__float2_t acc = 0.0f;`
 	
`#pragma DATA_ALIGN(data, 8);`
 	
`_nassert(size >= 2);`
`_nassert(size % 2 == 0);`
`_nassert((int) data % 8 == 0);`
 	
`for (i=0; i<size; i+=2) {`
 	
`acc =_daddsp (acc, _amemd8_const(&data[i]);`
 	
`}`
	• Force l'édition des liens (linking) à aligner en mémoire les données du tableau data[] modulo 8 octets
	• size est forcément supérieur ou égal à 2
	• size est forcément un multiple de 2
	• L'adresse de base du tableau data possède une valeur multiple de 8 (pointeur modulo 8 octets)

- Alignement mémoire : dans l'exemple ci-dessus, la directive de compilation #pragma DATA_ALIGN force l'édition des liens ou linking à garantir un alignement mémoire modulo 8 octets des données visées par le pointeur passé en argument. Prenons un exemple d'alignement mémoire de données modulo 8 octets sur chaîne de compilation:

```
/* arrays alignments - CPU data path length 64bits */
#pragma DATA_ALIGN(xk_sp, 8);
```

- Dans l'exemple de code avec assertions et alignement mémoire ci-dessus, que vaut la variable de type conteneur acc après exécution de la boucle ?

- Implémenter le code de la fonction **fir_sp_opt_r4** puis valider son fonctionnement. Cette fonction implémente l'**algorithme de filtrage FIR en C intrinsèque C6600 avec déroulement des 2 boucles interne et externe d'un facteur 4 (radix 4)**.

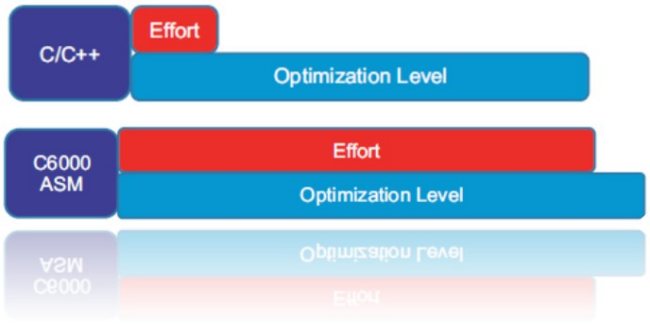
- Quelle limitation d'usage amène l'implémentation de notre algorithme de filtrage ?

- Après validation en mode Debug, lancer une exécution en **mode Release**, compiler, tester puis reporter les résultats des tests dans le tableau d'**analyse comparative** ou **Benchmarking** présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en **mode Debug**. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

- En vous aidant de l'annexe 1, générer une bibliothèque statique nommée **firlib.a** et la placer dans le répertoire **/disco/c6678/firlib/lib/**. Tester votre bibliothèque statique binaire avec votre projet de test.

- Voilà, à ce stade nous venons d'effectuer une implémentation optimale de notre algorithme pour notre DSP VLIW C6600. Porter un regard critique sur nos développements et citer les avantages et inconvénients de l'algorithme final



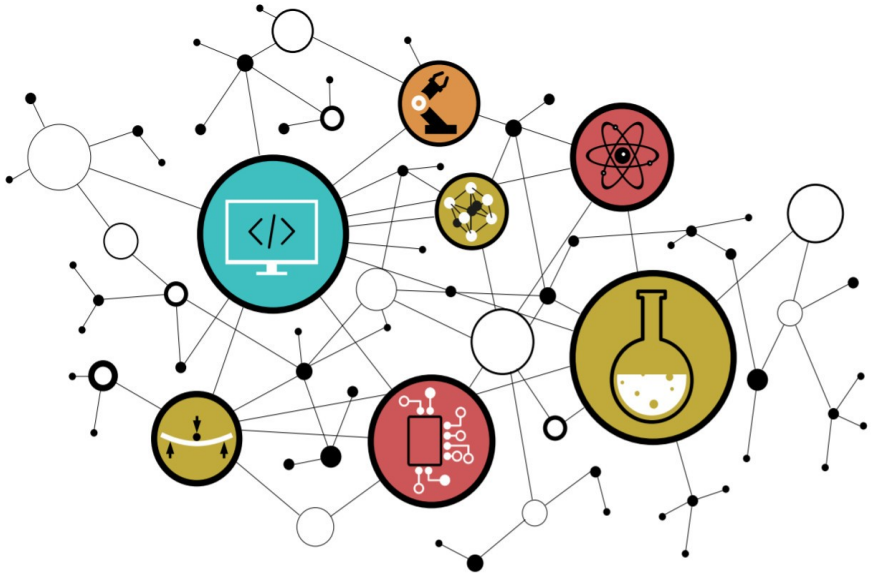
- L'exécution de l'algorithme tel qu'il a été écrit mathématiquement dans sa forme initiale ne peut pas être plus accélérée dans une optique de vectorisation en C sur notre architecture. Si nous souhaitons améliorer le temps de calcul de notre produit scalaire, il nous faudrait alors réduire sa complexité Mathématique en nombre de MACS. Les transformations à apporter ne seraient donc plus d'ordre technologique dans un premier temps, mais d'ordres mathématique et théorique. Une fois ces transformations appliquées, nous pourrions alors repasser sur des phases d'optimisation architectures dépendantes telles que celles présentées dans ce chapitre.

- De plus, nous venons de le constater, du moment que nous avons une connaissance poussée de l'architecture, du jeu d'instructions et de notre chaîne de compilation, développer des bibliothèques spécialisées en assembleur peut s'avérer contre-productif au regard du ratio performance/effort. Ceci sera vrai sur toute architecture VLIW, EPIC et superscalaire, du moment que l'étage d'optimisation de la toolchain reste un minimum performant.


notes

TRAVAUX PRATIQUES

PROGRAMMATION VECTORIELLE SUR GPP INTEL x86_64



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

3. PROGRAMMATION VECTORIELLE SUR GPP INTEL x86_64

3.1. Analyse du programme de test
3.2. Vectorisation avec ISA extension SSE4.1
3.3. Synthèse

3.1. Analyse du programme de test

Nous allons effectuer un Benchmarking entre architectures processeurs en mettant en confrontation une architecture superscalaire GPP (General Purpose Processor) Intel corei7 en micro-architecture Haswell cadencée à 3,6GHz (machines de TP en salles A203/A201) et notre architecture DSP VLIW C6600 cadencée à 1,4GHz proposée par Texas Instruments. Notre première comparaison se fera sur une implémentation en C canonique.

Intel corei7 – 4790 Haswell 4th gen	**Texas Instruments TMS320C6678**
3,6GHz, 105W en charge prix unitaire : 325€ (en 2015)	1,4GHz, 10W en charge prix unitaire : 240€ (160€ pour 1Ku)

- Ouvrir un shell et se déplacer dans le répertoire de travail disco/ia64. Afficher et analyser le contenu du fichier README.md à la racine. Compiler le projet et analyser les fichiers de test et résultats.

```
<your_computer_path>/disco/ia64$ make
<your_computer_path>/disco/ia64$ ./build/bin/firtest
```

- Exécuter plusieurs fois d'affilée le programme de test et observer les mesures de l'algorithme. Que constatez-vous ?

Nous pouvons constater qu'une implémentation en C canonique, même avec déroulement de boucle, garantit une portabilité du code, même sur des architectures matérielles différentes. Nous constatons également, que contrairement au CPU VLIW d'un DSP C6600, les CPU superscalaires des GPP x86_64 ne sont pas déterministes et donc ''potentiellement'' moins adaptés aux applications temps réel imposant des contraintes dures. Ceci est principalement lié aux étages suivants :

- Technologie du pipeline superscalaire (exécution Out-Of-Order, prédiction, spéculation, etc)
- Utilisation d'un modèle mémoire pleinement cachable pour les niveaux L1/L2/L3
- Utilisation d'une MMU (Memory Management Unit) pour la translation d'adresses virtuelles en adresses physiques (Table de translation, TLB, etc)
- Au côté multi-applicatif du système de la machine de test. De plus, Linux comme le noyau de Windows ne sont pas des kernels temps réel (scheduler non déterministe). En effet, bien d'autres programmes de même privilège sont en cours d'exécution sur l'ordinateur de développement (exécuter l'utilitaire htop pour observer en temps réel les processus et threads en cours d'exécution)

- Dans le Makefile, nous pouvons observer que des options de compilation spécifiques sont passées à GCC. A quoi correspondent les 3 options suivantes (s'aider de *man gcc* et d'internet):
    - -03
    - -std=c99
    - -march=native

```
CFLAGS = -Wall -march=native -std=c99 -O3
```

- Dans le fichier disco/ia64/test/src/main.c, nous pouvons observer la définition d'une fonction inline, dans notre cas implémentée en C avec insertion d'une séquence en assembleur 64bits x86_64. Cette fonction force le CPU courant à vider son pipeline matériel puis réalise une lecture de la valeur courante du core timer TSC (timer présent dans chaque CPU), comme précédemment sur architecture C6600. Sur architectures compatibles x86-x64, ce timer 64bits est démarré à la mise sous tension de la machine et compte jusqu'à débordement. En vous aidant d'internet, rappeler le rôle du spécificateur inline, préciser l'intérêt et donner des exemples d'utilisation ?

```
inline unsigned long long __attribute__((always_inline)) rdtsc_inline()
{
      unsigned int hi, lo;
      __asm__ __volatile__(
            // flush core pipeline
            "xorl %%eax, %%eax\n\t"
            "cpuid\n\t"
            // read current TSC value
            "rdtsc"
                  : "=a"(lo), "=d"(hi)
                  : // no parameters
                  : "rbx", "rcx");
      return ((unsigned long long)hi << 32ull) | (unsigned long long)lo;
}

...


start = rdtsc_inline();
fir_sp (xk_sp, a_sp, yk_sp_cn, A_LENGTH, YK_LENGTH);
stop = rdtsc_inline();
duration = stop - start;

...
```

- Quelles sont les principales différences entre une fonction inline et une fonction intrinsèque (s'aider d'internet)?

- Reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude. Prendre une moyenne de 10 mesures. Vous pouvez programmer la boucle de test si vous le souhaitez.

3.2. Vectorisation avec ISA extension SSE4.1

Nous allons maintenant comparer ce qui peut être comparable, à savoir les performances de notre code vectorisé sur architecture C6600 à du code vectorisé sur architecture IA-64. Nous nous intéresserons notamment à l'extension vectorielle SIMD SSE4.1 proposée par Intel. Pour information, courant 2014, Intel proposa sur sa micro-architecture Haswell une extension DSP (Digital Signal Processing). Cette extension, nommée FMA (Fused Multiply-Add), est donc dédiée aux applications de traitement numérique du signal mais ne sera néanmoins pas abordée en travaux pratiques. Pour les plus curieux, ne pas hésiter à aller voir sur MSDN (MicroSoft Developer Network) les quelques fonctions intrinsèques proposées.

Cette partie n'a pas vocation à permettre de découvrir en profondeur l'architecture interne des processeurs compatibles x86_64 (Intel ou AMD), notamment les architectures Intel. Néanmoins, nous allons pouvoir constater que nos précédents acquis nous permettent maintenant d'effectuer de la vectorisation de code sur toute architecture vectorielle processeur. Les concepts resteront le plus souvent les mêmes. Avant tout, nous avons à savoir que les architectures x86_64 actuelles possèdent plusieurs banques de registres vectoriels :

- Registres MMX 64bits : peuvent contenir jusqu'à 2 flottants 32bits
- Registres XMM 128bits : peuvent contenir jusqu'à 4 flottants 32bits
- Registres YMM 256bits : peuvent contenir jusqu'à 8 flottants 32bits

Dans notre cas, les instructions de l'extension SSE4.1 travaillent avec les registres XMM 128bits et sont aptes à manipuler les flottants en simple précision (IEEE-754) par paquets de 4. Le type conteneur __m128 permet donc de stocker jusqu'à 4 flottants en simple précision :

```
Container type (4 x 32bits floating point) : __m128
```

https://msdn.microsoft.com/fr-fr/library/y0dh78ez(v=vs.90).aspx

- Observons les préfixations et suffixations des fonctions intrinsèques présentées dans la suite de cet exercice :

```
Intel x86_64 Intrinsic syntax: _mm_<intrinsic_name>_<data_type>
```

- Mise à zéro des éléments d'un vecteur de flottants. Pour information, ps signifie Packet Single, soit paquet de flottants en simple précision 32bits IEEE754 :

```
__m128  float_vector;

 float_vector = _mm_setzero_ps ();
```

https://msdn.microsoft.com/fr-fr/library/tk1t2tbz(v=VS.90).aspx

- Initialise les éléments d'un vecteur de flottants :

```
__m128  float_vector;

 float_vector = _mm_set_ps (1.0, 2.2, 3.0, 7.0);
```

https://msdn.microsoft.com/fr-fr/library/afh0zf75(v=vs.90).aspx

- Chargement d'un vecteur de 4 flottants alignés depuis la mémoire vers un registre XMM de destination :

```
float  tab[4] = {1.0, 3.4, 5.0, 6.0};

__m128  float_vector;

 float_vector = _mm_load_ps (&tab[0]);
```

https://msdn.microsoft.com/fr-fr/library/zzd50xxt(v=vs.90).aspx

- Chargement d'un vecteur de 4 flottants non-alignés depuis la mémoire vers un registre XMM de destination :

```
float  tab[4] = {1.0, 3.4, 5.0, 6.0};

__m128  float_vector;

 float_vector = _mm_loadu_ps (&tab[0]);
```

https://msdn.microsoft.com/fr-fr/library/x1b16s7z(v=vs.90).aspx

- Produit scalaire entre deux vecteurs de 4 flottants (result = a0.x0 + a1.x1 + a2.x2 + a3.x3). Le résultat du produit scalaire est un nombre scalaire, le résultat est par défaut sauvé dans les 32bits de poids faible du vecteur 128bits de destination:

```
__m128  float_vector_src1;
__m128  float_vector_src2;
__m128  float_vector_dst;

float_vector_dst = _mm_dp_ps (float_vector_src1, float_vector_src2, 0xff);
```

https://msdn.microsoft.com/fr-fr/library/bb514054(v=vs.90).aspx

- Addition de deux vecteurs contenant 4 flottants chacun :

```
__m128  float_vector_src1;
__m128  float_vector_src2;
__m128  float_vector_dst;

 float_vector_dst = _mm_add_ps (float_vector_src1, float_vector_src2);
```

https://msdn.microsoft.com/fr-fr/library/c9848chc(v=vs.90).aspx

- Sauvegarde en mémoire d'un vecteur de 4 flottants alignés :

```
float  tab[4];
__m128  float_vector;

float_vector = _mm_set_ps (1.0, 2.2, 3.0, 7.0);

_mm_store_ps (&tab[0], float_vector);
```

https://msdn.microsoft.com/fr-fr/library/s3h4ay6y(v=vs.90).aspx

- Ouvrir le fichier /disco/ia64/firlib/h/firlib.h et observer les déclarations de type et d'union présentées ci-dessous. Cette union, greffée à une déclaration de type, permet, après déclaration d'une variable conteneur, d'accéder à un vecteur XMM soit élément par élément, soit directement au vecteur 128bits complet. En vous aidant d'internet, rappeler ce qu'est une union et préciser la différence avec une structure ?

```
union xmm_t {
      __m128   m128_vec;      // full access to XMM vector
      float    m128_f32[4];   // access to XMM vector elements
} align16_xmm;

typedef union xmm_t xmm_t;
```

- Dans l'exemple ci-dessous, comment accéder au 3ième élément de la variable vectorielle data_vec de type conteneur XMM 128bits.

```
xmm_t   data_vec;
```

- Ouvrir le fichier /disco/ia64/firlib/src/fir_sp_sse_r4.c puis implémenter le code vectorisé pour architecture x64 correspondant à la fonction fir_sp_r4. Valider son bon fonctionnement sans oublier de vérifier l'alignement mémoire des différents vecteurs de données traités par l'algorithme.

- Reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude. Prendre une moyenne de 10 mesures. Vous pouvez programmer la boucle de test si vous le souhaitez.

3.3. Synthèse

Rappelons les familles d'architectures CPU rencontrées sur le marché, et le plus souvent les familles de processeurs associées :

- CPU à pipeline classique (code In-Order en mémoire et exécution In-Order):
  - Rencontré par exemple sur MCU et certains DSP (processeur faible coût, consommation et encombrement réduits). Dans l'exemple des MCU (RISC-V, ARM Cortex-M, STM32 STMicroelectronics, PIC Microchip, Intel 8051, etc), le processeur embarque une application pouvant être sans OS (Baremetal) ou avec un système d'exploitation temps réel léger ou RTOS (FreeRTOS, Zephyr, etc).
  - Ces CPU peuvent être sinon intégrés en grand nombre sur certains processeurs, prenons l'exemple des GPU. Dans le cas des GPU, applications de calcul massivement parallèle (traitement d'image, vidéo, graphique 3D, finances, etc)
- CPU à pipeline superscalaire (code In-Order en mémoire et exécution Out-Of-Order) : grande polyvalence (processeur généraliste Système/Application/Calcul), le plus souvent grande puissance de calcul, pipeline complexe (consommation), donc forte intelligence déportée dans chaque cœur avec un faible niveau de parallélisme (typiquement 2, 4, 8, 16 cœurs souvent vectoriels). Rapport performance/consommation faiblement intéressant (par rapport aux MPPA, DSP, FPGA, etc). Chaque cœur (ensemble CPU et caches locaux) implémente le plus souvent les mécanismes suivants :
  - Prédiction au branchement
  - Étage d'exécution Out-Of-Order (exécution spéculative, étage de retirement avec étage de renommage des registres, etc)
  - Plusieurs unités d'exécution vectorielles par CPU aptes à travailler en parallèle
  - Mécanismes d'accélération aux étages de décodage, capture de boucles, etc
  - Cependant, dû à la complexité du pipeline, ces architectures peuvent offrir quelques irrégularités au regard du déterminisme à l'exécution (x86, x64, ARM cortex-A, IBM/APPLE/Freescale PowerPC, MIPS Aptiv, etc).
- CPU à pipeline VLIW ou EPIC (code Out-Of-Order en mémoire et exécution In-Order): forte puissance de calcul, pipeline relativement simple en opposition aux architectures superscalaires, donc rapport performance/consommation intéressant. Intelligence et compétences déportées vers le développeur et la toolchain. Grand déterminisme à l'exécution. Néanmoins, développements dépendants de l'architecture nécessaires, problèmes de portabilité de code. Rétrocompatibilité au niveau binaire difficile à suivre pour les fondeurs (MPPA Kalray, DSP TI C6000, NXP TriMedia, DSP SHARC Analog Device, ST200 STMicroelectronics, Intel Itanium, etc).

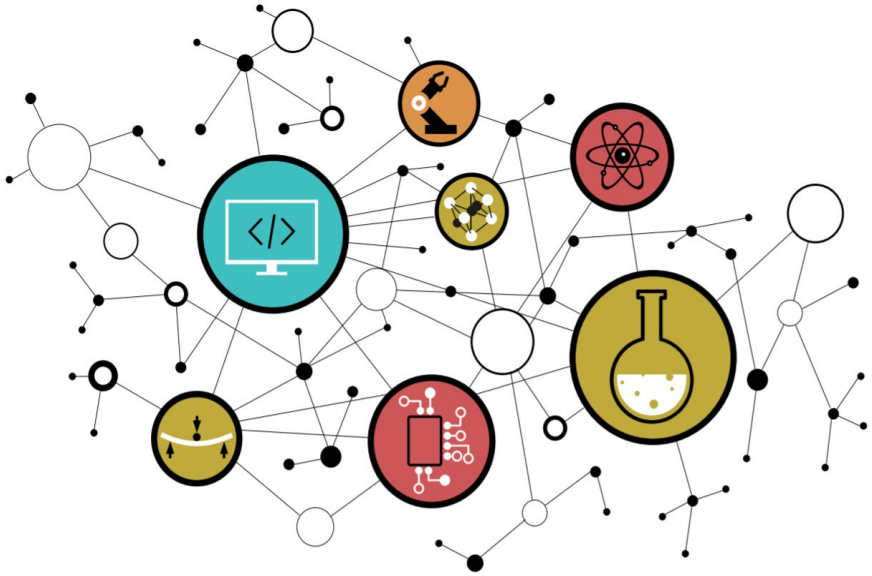
- Langage C vs Assembleur : dans une optique d'optimisation, un développement en langage C avec une bonne connaissance de l'architecture matérielle, du jeu d'instructions et des mécanismes d'optimisation de la toolchain peut éviter un passage à l'étage assembleur à notre époque (intrinsics, directives de compilation, déroulement de boucle, etc) et donc accélérer le TTM du produit (Time To Market). De même, nous avons effectué nos compilations sous gcc. Si nous souhaitons gagner en performance, il serait alors intéressant de travailler directement avec les outils fondeurs. Par exemple, si nous souhaitons garantir des performances optimales sur architecture Intel, il nous faudrait alors utiliser icc (Intel C++ Compiler).

- Tests : ne pas négliger les procédures de test (conformité et performance), notamment dans une optique de Benchmarking et d'optimisation d'algorithmes.


notes

TRAVAUX PRATIQUES

MEMOIRE CACHE
ET MEMOIRE SRAM ADRESSABLE



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

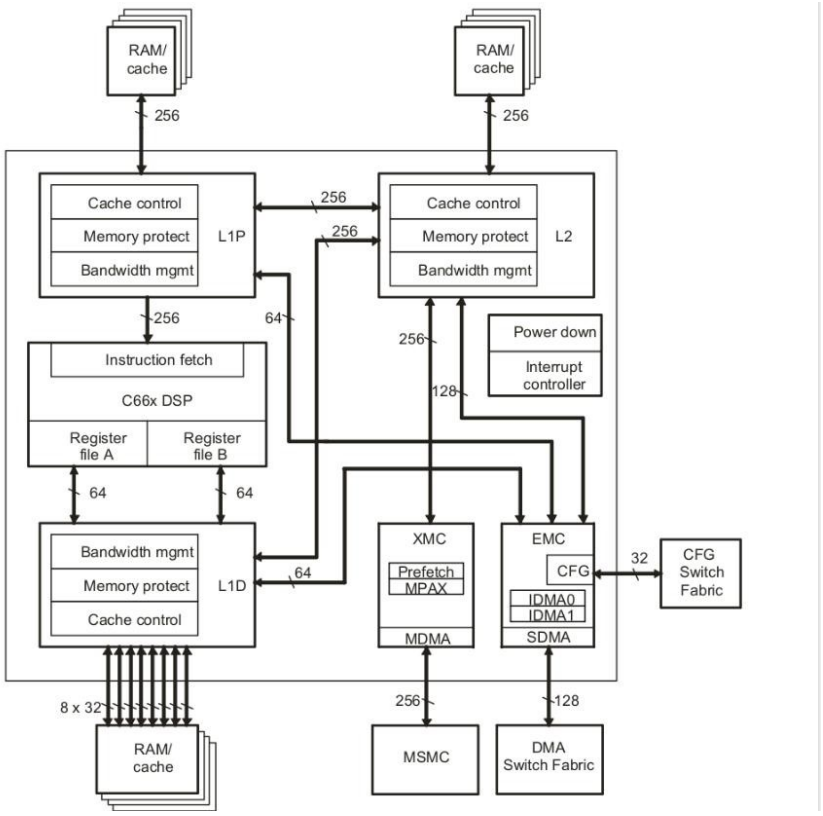
SOMMAIRE

4. MÉMOIRE CACHE ET MÉMOIRE SRAM ADRESSABLE

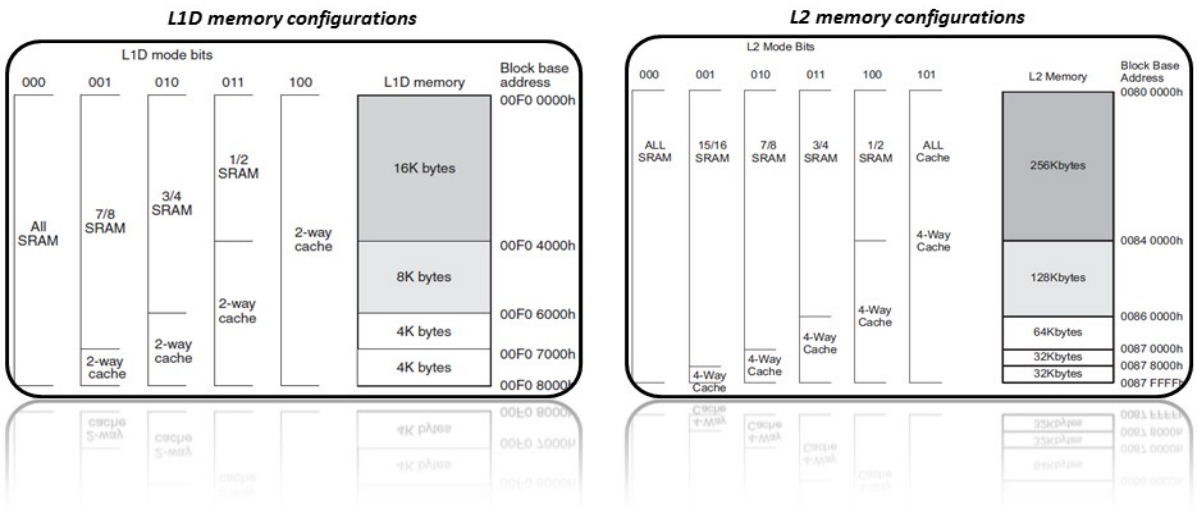
4.1. Mémoire locale SRAM adressable
4.2. Préchargement des données de DDR DRAM vers L2 SRAM
4.3. Préchargement des données de L2 SRAM vers L1D SRAM

4.1. Mémoire locale SRAM adressable

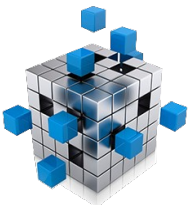
Nous allons maintenant chercher à passer outre les mémoires caches L1D et L2 en ne configurant qu'une partie des niveaux L1D et L2 en cache et le reste en mémoire SRAM adressable par octet. Ceci nous permettra de placer des données et le code souhaité (algorithme) dans des niveaux mémoire proches du CPU et ainsi de gagner en déterminisme (temps réel).

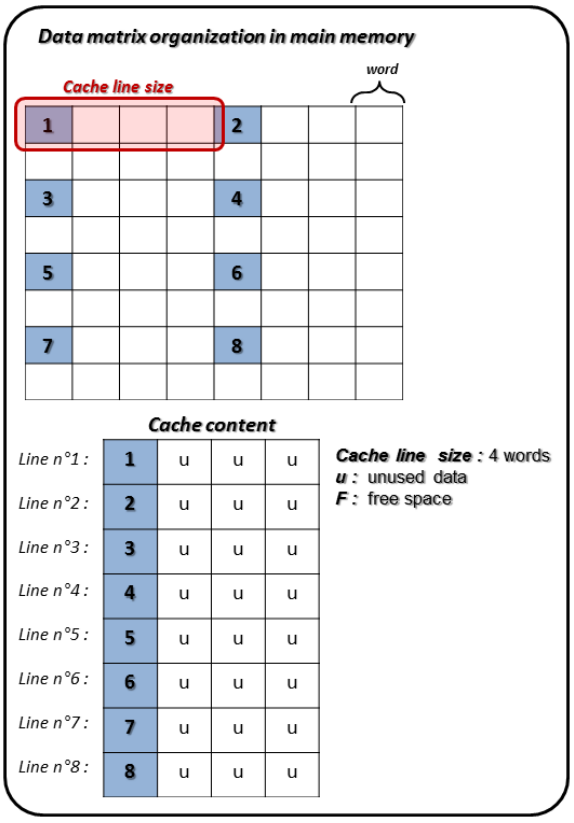


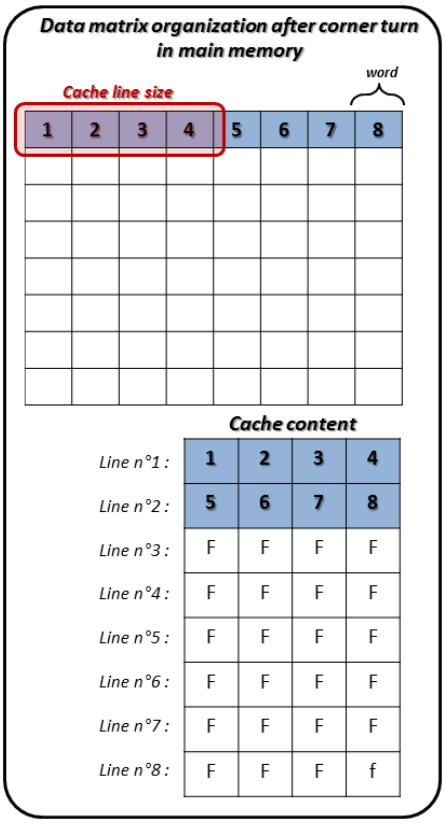
Dans l'exemple d'un produit scalaire, l'impact en performance restera minime. Néanmoins, pour grand nombre d'algorithmes du traitement numérique du signal (traitement d'image et d'antenne, etc), laisser opérer les contrôleurs cache sans ordonnancement des données avant traitement peut avoir une incidence énorme sur le temps d'exécution global de l'application. Dans l'exemple qui suit, nous pouvons observer une matrice de données manipulée par indexage (chargement/lecture donnée 1 puis 2 puis 3, etc). Rappelons également que les mémoires caches sont pilotées par des contrôleurs effectuant des copies d'informations depuis les niveaux hiérarchiques mémoire supérieurs. Ces contrôleurs implémentent des mécanismes de prédiction spéculatifs. Par exemple, chaque copie d'information se fait par ligne (ensemble d'octets).



Un cache étant chargé par ligne (principe de localité spatiale), nous sommes alors amenés à charger un grand nombre de données ''potentiellement'' inutilisées en cache. Une technique couramment utilisée, consiste à ré-implémenter l'algorithme de façon à traiter des données pré-ordonnancées en mémoire (technique du corner turn). Cette stratégie permet de diminuer le nombre d'accès au cache et assure une utilisation optimale des espaces de stockage en cache. Rappelons que l'accès aux ressources en mémoire est l'un des principaux goulots d'étranglement en termes de performance à notre époque. Nous avons également remarqué que notre architecture possède un grand nombre de registres de travail généralistes par CPU. L'objectif étant de charger les données locales à une fonction ou une boucle dans les registres du cœur, pour les traiter par la suite en minimisant au plus les allers-retours vers le cache. Écritures différées en mémoire en fin de traitement.

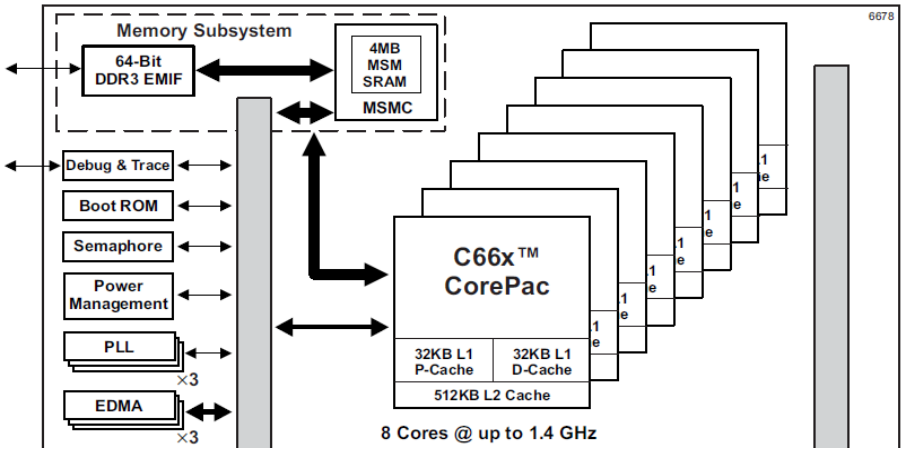






Dans notre cas, l'objectif est différent. L'algorithme d'un produit scalaire sans transformations mathématiques dans une optique de diminution de sa complexité, manipule les vecteurs d'entrée dans l'ordre de façon séquentielle. Cet exercice n'a donc pour objectif que d'illustrer la possibilité d'obtenir un espace de stockage adressable à accès rapide physiquement proche du cœur (L1D et L2). Nous maîtriserons alors avec certitude les données présentes aux niveaux L1D et L2 afin de garantir un déterminisme à l'exécution. Cette solution permet par exemple de s'affranchir localement du principe de corner turn précédemment présenté. En effet, nous allons manuellement effectuer le travail des contrôleurs de cache utilisant quant à eux des mécanismes de localité temporelle (LRU) pour leurs allocations/libérations de lignes de cache. Traitement impossible sur processeur généraliste GPP. Ceci permettra de rendre notre algorithmique quasiment insensible à la charge éventuelle du cache. Autre possibilité non explorée dans cette trame d'enseignement, la parallélisation des copies mémoire descendantes (DDR vers L1D), avec le calcul algorithmique (L1D et CPU) et avec les copies mémoire montantes (L1D vers DDR). Cette approche est implémentée par certains partenaires et offre des gains en performance considérables.

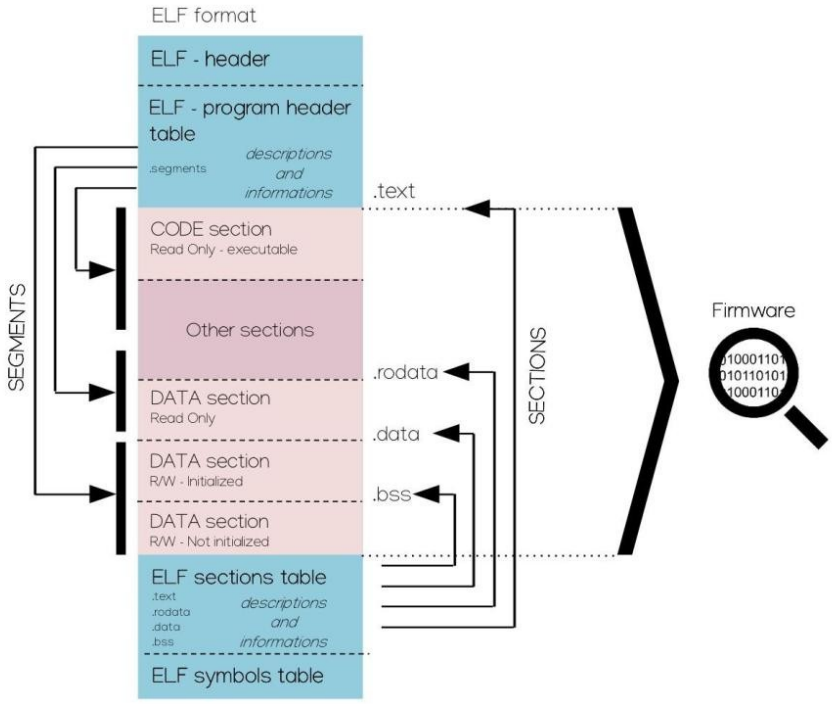
Nous avons possibilité de configurer les différents niveaux mémoire en cache ou en SRAM adressable de façon statique à la compilation ou dynamique à l'exécution. Rappelons avant tout ci-dessous l'architecture mémoire de notre processeur :



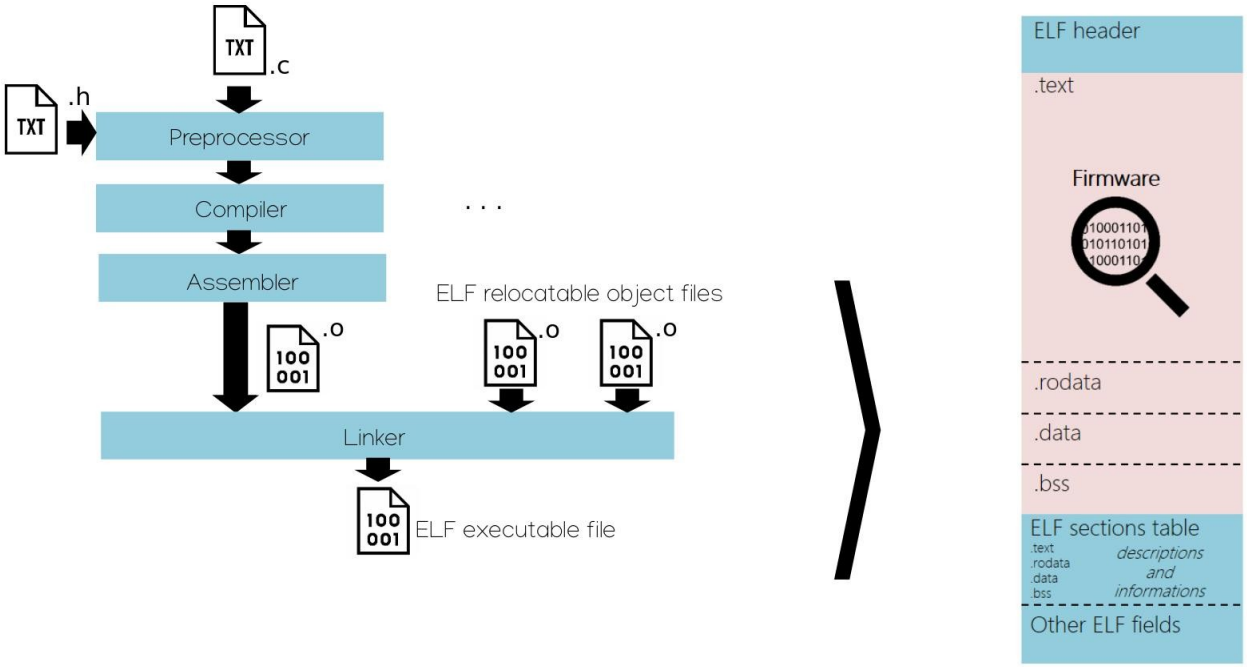
- Dans notre projet CCS, remplacer le script linker C6678_unified.cmd par le fichier /disco/c6678/test/map/C6678_unified_fir.cmd
- Analyser le fichier script Linker /disco/c6678/test/map/C6678_unified_fir.cmd

```
-stack 0x2000
-heap 0x2000

MEMORY
{
 LOCAL_L1P_SRAM:  origin = 0x00E00000 length = 0x00008000   /* 32kB LOCAL L1P/SRAM */
 LOCAL_L1D_SRAM:  origin = 0x00F00000 length = 0x00008000   /* 32kB LOCAL L1D/SRAM */
 LOCAL_L2_SRAM:   origin = 0x00800000 length = 0x00080000   /* 512kB LOCAL L2/SRAM */
 MSMCSRAM:        origin = 0x0C000000 length = 0x00400000   /* 4MB Multicore shared Memmory */

 EMIF16_CS2:     origin = 0x70000000 length = 0x04000000    /* 64MB EMIF16 CS2 Data Memory */
 EMIF16_CS3:     origin = 0x74000000 length = 0x04000000    /* 64MB EMIF16 CS3 Data Memory */
 EMIF16_CS4:     origin = 0x78000000 length = 0x04000000    /* 64MB EMIF16 CS4 Data Memory */
 EMIF16_CS5:     origin = 0x7C000000 length = 0x04000000    /* 64MB EMIF16 CS5 Data Memory */

 /* TMDSEVM6678L board specific */
 DDR3:           origin = 0x80000000 length = 0x20000000   /* 512MB external DDR3 SDRAM */
}

SECTIONS
{
 .text          >  MSMCSRAM
 .stack         >  MSMCSRAM
 .bss           >  MSMCSRAM
 .cio           >  MSMCSRAM
 .const         >  MSMCSRAM
 .data          >  MSMCSRAM
 .switch        >  MSMCSRAM
 .sysmem        >  MSMCSRAM
 .far           >  DDR3
 .args          >  MSMCSRAM
 .ppinfo        >  MSMCSRAM
 .ppdata        >  MSMCSRAM

 /* COFF sections */
 .pinit         >  MSMCSRAM
 .cinit         >  DDR3

 /* EABI sections */
 .binit         >  MSMCSRAM
 .init_array    >  MSMCSRAM
 .neardata      >  MSMCSRAM
 .fardata       >  DDR3
 .rodata        >  MSMCSRAM
 .c6xabi.exidx  >  MSMCSRAM
 .c6xabi.extab  >  MSMCSRAM

 /* project specific sections */

       /* user sections */
}
```

Rappelons (cf. cours 1A en Archi. des ordi.) que **les sections sont des zones de codes ou de données statiques allouées à la compilation et l'édition des liens, et présentes dans le fichier binaire exécutable de sortie (ELF, COFF, etc)**. Les sections seront par la suite mappées en mémoire physique du processeur. Dans notre cas, soit en DDR externe, soit en MSMC/L2/L1D/L1P interne.



- A quoi correspond la zone **MEMORY** du script ?
- En vous aidant de l'annexe ou de la documentation technique **disco/tp/doc/datasheet - optimizing compiler – spru187v.pdf**, préciser le rôle de la zone **SECTIONS** du script.
- Dans quelle section est rangée la **pile** ou stack système (allocations automatiques) ?
- Dans quelle section est rangé le **tas** ou heap système (allocations dynamiques) ?
- Dans quelle section est rangé le **code utilisateur** (s'aider de la page suivante) ?
- Analyser les parties du script permettant de configurer les niveaux mémoire L1D et L2 comme suit (à entourer sur la page précédente) :
    - **32Ko L1D** : 28Ko en L1D SRAM adressable et 4Ko en L1D cache
    - **512Ko L2** : 480Ko en L2 SRAM adressable et 32Ko en L2 cache
- Créer 3 nouvelles sections propres au projet et les placer tel que précisé ci-dessous :
    - **.ddrsdram** : section statique présente en DDR
    - **.l2sram** : section statique présente en L2 SRAM
    - **.l1dsram** : section statique présente en L1D SRAM
- Compiler puis exécuter le programme. Valider son bon fonctionnement.



Les allocations statiques représentent toutes les allocations de ressources mémoire réalisées à la compilation et l'édition des liens, et donc présentes dans le fichier binaire ELF de sortie. Les variables statiques admettent donc une existence sur un media de stockage de masse avant même l'exécution d'un programme en mémoire principale. Les références symboliques, ou adresses logiques, de chaque fonction et variable statique sont donc inchangées (statiques) durant la totalité de la vie d'un programme binaire sans nouvelle compilation et édition des liens du projet logiciel source. Contrairement aux variables locales (allocations automatiques ou dynamiques sur le segment de pile) et allocations dynamiques sur le segment de tas, pour lesquelles les allocations de ressources mémoire sont réalisées à l'exécution du programme dans des segments logiques dédiés (Pile ou Tas).

Une *section* est une zone logique du *firmware*. Un *Firmware* peut être également nommé micrologiciel en Français. Le *firmware* représente dans cet enseignement le code (forcément statique) et les données statiques binaires strictement utiles au fonctionnement d'un programme. Le *firmware* est quant à lui encapsulé dans un cartouche au format ELF (en-tête, table des sections, en-tête du programme, etc) proposant au noyau du système (Linux) une description et des informations sur le micrologiciel afin d'aider à sa manipulation (préparation des segments mémoire, association de propriétés et privilèges, etc). Sauf si un développeur créé explicitement de nouvelles sections en spécifiant des attributs spécifiques durant une déclaration d'une variable (ce que nous sommes en train de faire), une application pourra comporter au plus 4 sections applicatives par défaut afin de gérer l'ensemble des besoins standards en allocations statiques de ressources (les noms suivants sont hérités d'Unix) :

- .text (CODE - Read Only - executable) : section encapsulant le code binaire statique du programme
- .rodata (DATA - Read Only – not executable) : section encapsulant les données statiques accessibles en lecture seule
- .data (DATA - Read/Write – not executable) : section encapsulant les données statiques initialisées accessibles en lecture et écriture
- .bss (DATA - Read/Write – not executable) : section encapsulant les données statiques non-initialisées accessibles en lecture et écriture

4.2. Préchargement des données de DDR DRAM vers L2 SRAM

- A partir de maintenant et jusqu'à la fin de la trame de travaux pratiques, toutes nos futures optimisations (mémoire et périphériques d'accélération) utiliseront l'algorithme vectorisé fir_sp_opt_r4 précédemment développé et offrant un niveau optimum d'accélération.

- Remplacer le fichier main.c actuel par celui présent dans disco/c6678/sram_cache/main.c et analyser les déclarations des vecteurs statiques (variables globales) qui assureront des stockages partiels de nos vecteurs d'entrée et de sortie. Ces vecteurs stockeront temporairement en L2 SRAM et L1D SRAM des tronçons de vecteurs à traiter.

```
/* arrays allocations (bytes) :
* xk_sp (DDR)    |---------------- 256Kb or 4Mb --------------------|
* xk_sp_l2       |------- 128Kb + 256b - 4 ------| overloap
* xk_sp_l1d      |--- 8Kb + 256b - 4  ---| overloap
* a_sp_l1d         |- 256b -|
* yk_sp_l1d      |----- 8Kb -----|
* yk_sp_l2       |------------- 128Kb -------------|
* yk_sp (DDR)    |--------- (256Kb or 4Mb) - 256b + 4 ---------|
*/
float32_t xk_sp[XK_LENGTH];
float32_t yk_sp_ti[YK_LENGTH];
float32_t yk_sp[YK_LENGTH];
float32_t xk_sp_l2[L2_ARRAY_LENGTH + A_LENGTH - 1];
float32_t yk_sp_l2[L2_ARRAY_LENGTH];
float32_t xk_sp_l1d[L1D_ARRAY_LENGTH + A_LENGTH - 1];
float32_t yk_sp_l1d[L1D_ARRAY_LENGTH];
float32_t a_sp_l1d[A_LENGTH];

/* memory segmentation */
#pragma DATA_SECTION(xk_sp,".ddrsdram");
#pragma DATA_SECTION(yk_sp_ti,".ddrsdram");
#pragma DATA_SECTION(yk_sp,".ddrsdram");
#pragma DATA_SECTION(xk_sp_l2,".l2sram");
#pragma DATA_SECTION(yk_sp_l2,".l2sram");
#pragma DATA_SECTION(xk_sp_l1d,".l1dsram");
#pragma DATA_SECTION(yk_sp_l1d,".l1dsram");
#pragma DATA_SECTION(a_sp_l1d,".l1dsram");

/* arrays alignments - CPU data path length 64bits */
#pragma DATA_ALIGN(xk_sp, 8);
#pragma DATA_ALIGN(a_sp, 8);
#pragma DATA_ALIGN(yk_sp_ti, 8);
#pragma DATA_ALIGN(yk_sp, 8);
#pragma DATA_ALIGN(xk_sp_l2, 8);
#pragma DATA_ALIGN(yk_sp_l2, 8);
#pragma DATA_ALIGN(xk_sp_l1d, 8);
#pragma DATA_ALIGN(yk_sp_l1d, 8);
#pragma DATA_ALIGN(a_sp_l1d, 8);
```

- Quel est le rôle de la directive #pragma DATA_SECTION ? Quel étage du processus de compilation et d'édition des liens est concerné ?

- Quel est le rôle de la directive #pragma DATA_ALIGN ? Quel étage du processus de compilation et d'édition des liens est concerné ?

- Pourquoi les tailles des vecteurs placés en L2 SRAM occupent-elles ~128Ko ? Aurait-on pu prendre des vecteurs de 256Ko ? Si oui, quelle est la limite et qu'aurions nous dû modifier dans notre projet ?

- Remplacer le fichier firtest_perf.c actuel par celui présent dans disco/c6678/sram_cache/firtest_perf .c et analyser le programme fourni réalisant des copies des vecteurs d'entrée (xk_sp) et de sortie (yk_sp) par tronçons de DDR vers L2 SRAM adressable

```
...
#if ( TEST_FIR_L2SRAM_L1DCACHE != 0 )
  else if ( memoryModel == UMA_L2SRAM_L1DCACHE ) {
       /* caches levels initializations */
       CACHE_setL2Size(CACHE_32KCACHE);
       CACHE_setL1DSize(CACHE_L1_32KCACHE);
       CACHE_setL1PSize(CACHE_L1_32KCACHE);

       start = CSL_tscRead ();

       /* copy part of input array from DDR to L2 SRAM */
       for(i=0; i<DDR_ARRAY_LENGTH; i+=L2_ARRAY_LENGTH){

              /* memory copy from DDR to L2 SRAM */
              if( i < (DDR_ARRAY_LENGTH - L2_ARRAY_LENGTH) ){
                     memcpy(xk_sp_l2, xk_sp + i, (L2_ARRAY_LENGTH + A_LENGTH - 1)*sizeof(float32_t));
              } else {
                     memcpy(xk_sp_l2, xk_sp + i, L2_ARRAY_LENGTH*sizeof(float32_t));
              }

              /* call fir algorithm for performance test */
              (*fir_fct) (xk_sp_l2, a_sp, yk_sp_l2, A_LENGTH, L2_ARRAY_LENGTH);

              /* memory copy from L2 SRAM to DDR */
              if( i < (YK_LENGTH - L2_ARRAY_LENGTH) ){
                     memcpy(output + i, yk_sp_l2, L2_ARRAY_LENGTH*sizeof(float32_t));
              } else {
                     memcpy(output + i, yk_sp_l2, (L2_ARRAY_LENGTH - A_LENGTH + 1)*sizeof(float32_t));
              }
       }

       stop = CSL_tscRead ();
       duration += stop-start;
   }
#endif

...
```

- En vous aidant de la documentation technique de CSL (Chip Support Library ou bibliothèque de fonctions pilotes des DSP C6000) présente dans le répertoire de projet /tp/doc/csl/README.md et des macros et énumérateurs présents dans le fichier d'en-tête C:\ (sur Windows) ou /opt (sous GNU\Linux) /ti/pdk_C6678_<version>/packages/ti/csl/csl_cache.h, préciser les rôles des fonctions suivantes

```
CACHE_setL2Size(CACHE_32KCACHE);
CACHE_setL1DSize(CACHE_L1_32KCACHE);
CACHE_setL1PSize(CACHE_L1_32KCACHE);
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_L2SRAM_L1DCACHE à 1

```
#define TEST_FIR_ L2SRAM_L1DCACHE  1
```

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

4.3. Préchargement des données de L2 SRAM vers L1D SRAM

- Ouvrir et analyser le programme de prototypage Matlab présent dans disco/matlab/src/main_mem_ddr_l2_l1d.m

```
for i=1 : L2_ARRAY_LENGTH : DDR_ARRAY_LENGTH

    % memcpy DDR to L2 SRAM
    if i < DDR_ARRAY_LENGTH - L2_ARRAY_LENGTH
        for k=1 : L2_ARRAY_LENGTH + A_LENGTH - 1
            xk_sp_L2(k) = xk_sp_DDR(i+k-1);
        end
    else
        for k=1 : L2_ARRAY_LENGTH
            xk_sp_L2(k) = xk_sp_DDR(i+k-1);
        end
    end

    % copy part of input array from L2 SRAM to L1D SRAM
    for j=1 : L1D_ARRAY_LENGTH : L2_ARRAY_LENGTH

        % memcpy L2 to L1D
        for k=1 : L1D_ARRAY_LENGTH + A_LENGTH - 1
            xk_sp_L1D(k) = xk_sp_L2(j+k-1);
        end

        yk_sp_L1D = fir_sp(xk_sp_L1D, a_sp, A_LENGTH, L1D_ARRAY_LENGTH);

        % memcpy L1D SRAM to L2 SRAM - coherency of output L2 array
        for k=1 : L1D_ARRAY_LENGTH
             yk_sp_L2(j+k-1) = yk_sp_L1D(k);
        end

    end

    % memcpy L2 SRAM to DDR - coherency of output DDR array
    if i < YK_LENGTH - L2_ARRAY_LENGTH
        for k=1 : L2_ARRAY_LENGTH
            yk_sp_DDR(i+k-1) = yk_sp_L2(k);
        end
    else
        for k=1 : L2_ARRAY_LENGTH - A_LENGTH + 1
            yk_sp_DDR(i+k-1) = yk_sp_L2(k);
        end
    end

end
```

- Dans le fichier firtest.h, mettre la macro TEST_FIR_L2SRAM_L1DSRAM à 1

```
#define TEST_FIR_L2SRAM_L1DSRAM    1
```

- Compléter le fichier firtest_perf.c de façon à implémenter manuellement des copies mémoires de données de DDR vers L2SRAM et de L2SRAM vers L1SRAM (cf. programme Matlab ci-dessus).

```
#if ( TEST_FIR_L2SRAM_L1DSRAM != 0 )
       else if ( memoryModel == UMA_L2SRAM_L1DSRAM ) {
             /* TODO */
       }
#endif
```

- A ce niveau là de maîtrise du modèle mémoire, quelles données restent encore présentes en cache ?

- Après validation en mode Debug, lancer une exécution en **mode Release**, compiler, tester puis reporter les résultats des tests dans le tableau d'**analyse comparative** ou **Benchmarking** présent dans le document de Prélude.

- Une fois la mesure réalisée, se remettre en configuration initiale en **mode Debug**. Toujours garder le mode Debug pour les phases de développement et le mode Release pour les phases de mesure

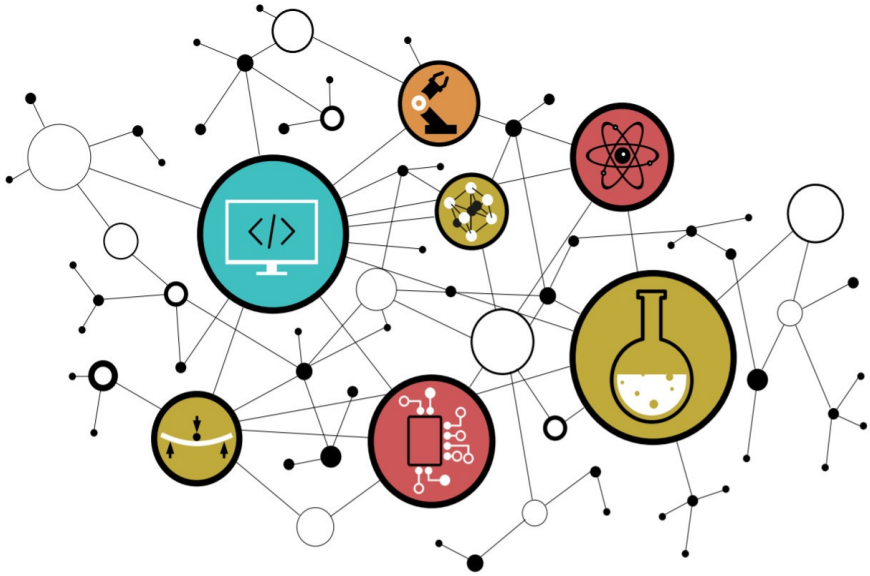
A ce stade de nos développements, nous constatons une légère perte de performance mais néanmoins un déterminisme garanti à l'exécution (maîtrise totale des données présentes dans les niveaux L2 et L1D). Juste pour information, c'était déjà partiellement le cas précédemment avec un modèle pleinement cachable dans le cadre d'un produit scalaire (données manipulées séquentiellement dans l'ordre). Néanmoins, pour grand nombre d'autres algorithmes DSP avec indexages complexes (exemple de la FFT, DCT, etc), nous aurions pu avoir de mauvaises surprises à n'utiliser que du cache.

Sans pour autant modifier notre modèle mémoire, nous avons la possibilité d'accélérer nos transferts. En effet, jusqu'à présent nous avons utilisé la fonction standard **memcpy** réalisant les transferts via le **CPU (instructions LDx/STx)**. Nous allons maintenant regarder de plus près les périphériques **DMA (Direct Memory Access)**, spécialisés dans les transferts mémoire autonomes sans passer par le CPU.


notes

TRAVAUX PRATIQUES

PERIPHERIQUES DE COPIE MÉMOIRE DMA



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

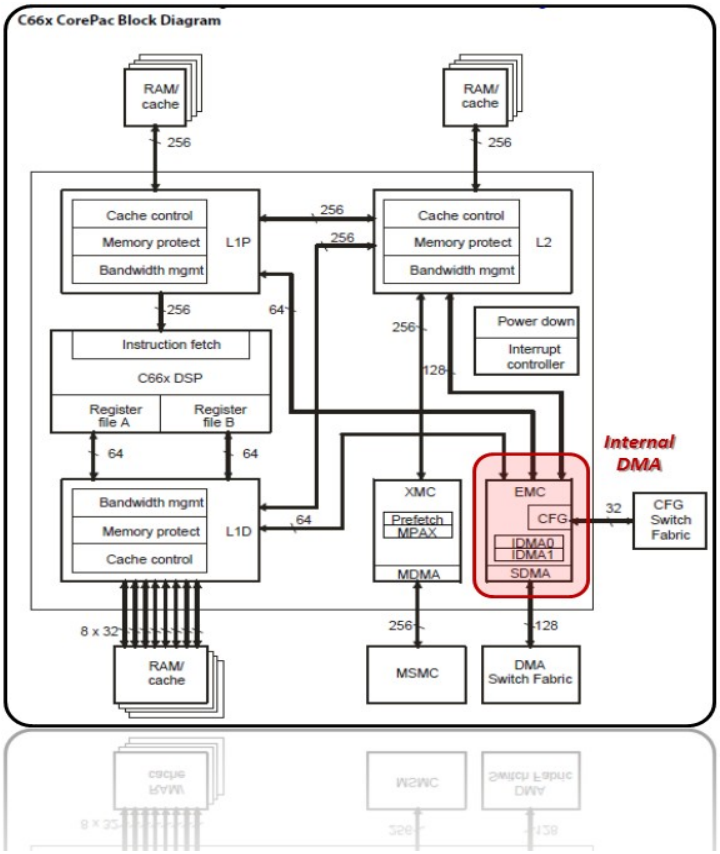
SOMMAIRE

5. PÉRIPHÉRIQUES DE COPIE MÉMOIRE DMA

5.1. Transferts par IDMA
5.2. Stratégie Ping Pong
5.3. Transferts par EDMA

5.1. Transferts par IDMA

Dans cette ultime partie, nous allons nous intéresser à des périphériques d'accélération standards sur architectures processeurs évoluées, les DMA (Direct Memory Access). Un DMA est un périphérique de copie d'information de mémoire à mémoire sans passer par le CPU. Un DMA possède ses propres bus et est, après configuration, une entité autonome de l'architecture du processeur. Un DMA classique effectue des copies d'une zone mémoire à une autre et est capable, comme tout périphérique, de prévenir le CPU de la fin d'un transfert par l'envoi d'une interruption matérielle (IRQ ou Interrupt Request). Dans un premier temps, nous travaillerons avec les IMDA (Internal DMA) de notre processeur. Ces IDMA sont présents dans chaque cœur et ne proposent que des services standards de copies (copies linéaires contiguës de mémoire à mémoire L2 SRAM, L1D SRAM ou L1P SRAM) :



Certains DMA plus évolués (exemple des Enhanced DMA chez TI, les Smart DMA chez Freescale, etc) sont capables d'effectuer :

- Des transferts avec modes d'adressages et indexations complexes (manipulation de matrices, de cubes de données, etc)
- De synchroniser des transferts sur événements matériels issus de périphériques
- De capturer et de garder de grands nombres de pré-configurations
- Une étude approfondie de ces types de DMA peut alors prendre des mois

Sur le schéma ci-dessus, nous pouvons observer plus finement l'architecture interne d'un corePac (ensemble CPU/cœur, caches associés et utilitaires matériels spécifiques, exemple des IDMA). Bien faire la distinction entre les mémoires caches (espaces de stockages) et les contrôleurs de caches (DMA autonome travaillant dans notre cas avec des mécanismes de localités temporelles LRU pour le remplacement des lignes de caches). Un contrôleur de cache effectuant des copies de mémoire à mémoire, il n'est donc qu'un DMA autonome.

- Dans le fichier firtest.h, mettre la macro TEST_FIR_L2SRAM_L1DIDMA à 1

```
#define TEST_FIR_ L2SRAM_L1DIDMA   1
```

- Remplacer l'ancien fichier firtest_perf.c par celui présent dans disco/c6678/idma/firtest_perf.c. A l'analysant (cf. ci-dessous), vous constaterez qu'il implémente le même code que celui écrit précédemment (exercice 4.3). Seulement maintenant les copies mémoires de L2 SRAM vers L1D SRAM sont réalisées par périphérique IDMA

```
#if ( TEST_FIR_L2SRAM_L1DIDMA != 0 )
   else if ( memoryModel == UMA_L2SRAM_L1DSRAM ) {
       /* caches levels initializations */
       CACHE_setL2Size(CACHE_32KCACHE);
       CACHE_setL1DSize(CACHE_L1_4KCACHE);
       CACHE_setL1PSize(CACHE_L1_32KCACHE);

       /* prepare coefficients in L1D SRAM */
       memcpy(a_sp_l1d, a_sp, A_LENGTH*sizeof(float32_t));

       start = CSL_tscRead ();

       /* copy part of input array from DDR to L2 */
       for(i=0; i<DDR_ARRAY_LENGTH; i+=L2_ARRAY_LENGTH){

          /* memory copy from DDR to L2 SRAM */
          if( i < (DDR_ARRAY_LENGTH - L2_ARRAY_LENGTH) ){
              memcpy(xk_sp_l2, xk_sp + i, (L2_ARRAY_LENGTH + A_LENGTH - 1)*sizeof(float32_t));
          } else {
              memcpy(xk_sp_l2, xk_sp + i, L2_ARRAY_LENGTH*sizeof(float32_t));
          }

          /* copy part of input array from L2 SRAM to L1D SRAM */
          for(j=0; j<L2_ARRAY_LENGTH; j+=L1D_ARRAY_LENGTH ){

              /* memory copy from L2 SRAM to L1D SRAM */
              idmacpy(xk_sp_l1d, xk_sp_l2 + j, (L1D_ARRAY_LENGTH + A_LENGTH - 1)*sizeof(float32_t));

              (*fir_fct) (xk_sp_l1d, a_sp, yk_sp_l1d, A_LENGTH, L1D_ARRAY_LENGTH);

              /* memory copy from L1D SRAM to L2 SRAM - coherency of output L2 array*/
              idmacpy(yk_sp_l2 + j, yk_sp_l1d, L1D_ARRAY_LENGTH*sizeof(float32_t));
          }

          /* memory copy from DDR to L2 */
          if( i < (YK_LENGTH - L2_ARRAY_LENGTH) ){
              memcpy(output + i, yk_sp_l2, L2_ARRAY_LENGTH*sizeof(float32_t));
          } else {
              memcpy(output + i, yk_sp_l2, (L2_ARRAY_LENGTH - A_LENGTH + 1)*sizeof(float32_t));
          }
       }

       stop = CSL_tscRead ();
       duration += stop-start;
   }
#endif
```

- Ajouter à votre projet la fonction disco/c6678/idmalib/src/idmacopy.c et valider la bonne compilation du projet. Résoudre les erreurs potentielles de compilation.

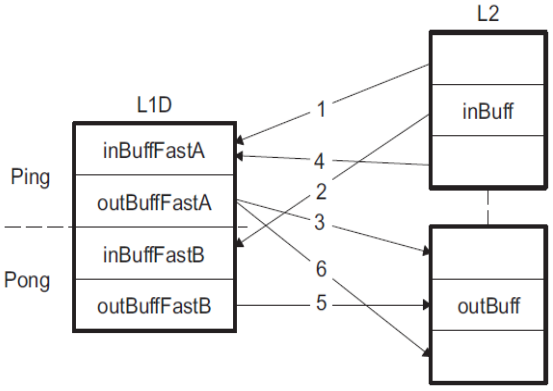
- En vous aidant de la documentation technique de CSL (Chip Support Library ou bibliothèque de fonctions pilotes des DSP C6000) présente dans le répertoire de projet /disco/tp/doc/csl/README.md et de la partie propre à l'IDMA, compléter la fonction idmacopy de façon à remplacer la fonction memcopy précédemment utilisée. *Le programme est très simple, 7 lignes dont une seule ligne de code utile.*

- Après validation en mode Debug, lancer une exécution en mode Release, compiler, tester puis reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude. Une fois la mesure réalisée, se remettre en configuration initiale en mode Debug.

5.2. Stratégie Ping Pong

Même si la solution précédente reste plus rapide qu'une copie par CPU, elle n'est pas optimale. Notre IDMA possède en réalité 2 canaux de transfert indépendants par cœur. Ces canaux peuvent alors être utilisés en parallèles en manipulant chacun deux vecteurs temporaires de stockage en L1D SRAM. Cette stratégie très générique de copie se nomme **Ping Pong**, lorsque le côté Ping est en cours de traitement (algorithme de traitement du signal), le côté Pong est en cours de chargement/sauvegarde, etc. Rappelons d'ailleurs que nous possédons 24Ko de mémoire L1D SRAM adressable, nous autorisant ainsi à pouvoir définir de nouveaux vecteurs temporaires de stockage. Observons par exemple une séquence de transferts Ping Pong en 6 étapes :

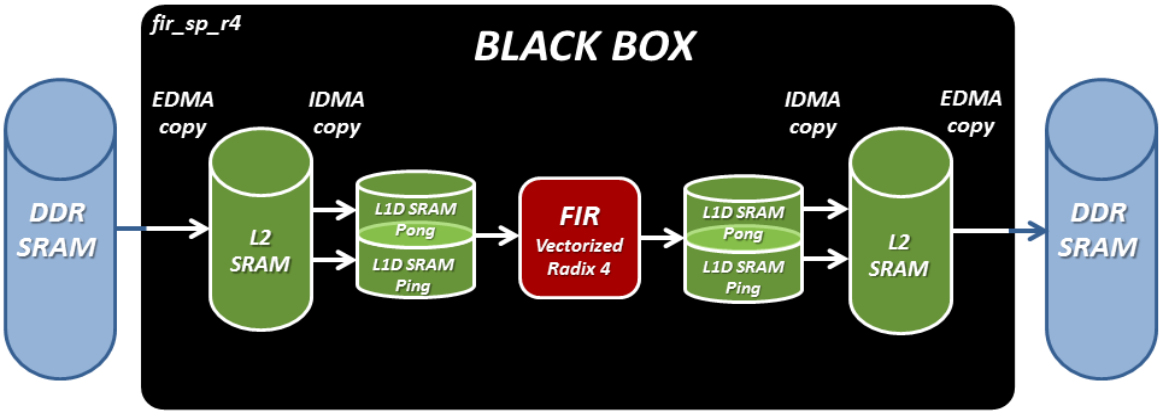
- **Canal 0** : Dédié aux chargements mémoires L2 SRAM vers L1D SRAM
- **Canal 1** : Dédié aux sauvegardes mémoires L1D SRAM vers L2 SRAM



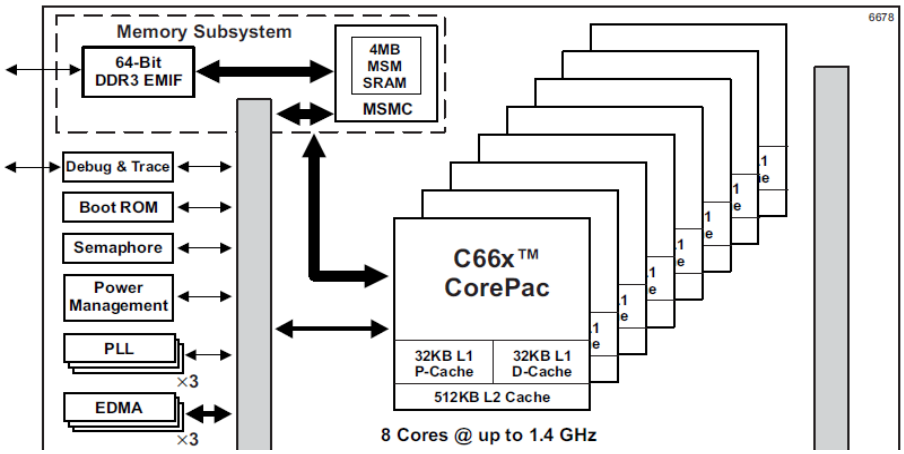
- **Étape 1** : Chargement donnée **xk_sp** du côté Ping de L2 SRAM vers L1D SRAM
- **Étape 2** : Chargement donnée **xk_sp** du côté Pong de L2 SRAM vers L1D SRAM et si chargement du côté Ping terminé, application de l'algorithme de filtrage sur les données précédemment chargées du côté Ping
- **Étape 3** : Si traitement algorithmique terminé, sauvegarde des données de sortie **yk_sp** du côté Ping de L1D SRAM vers L2 SRAM et si chargement du côté Pong terminé, application de l'algorithme de filtrage sur les données précédemment chargées du côté Pong
- **Étape 4** : Chargement donnée **xk_sp** du côté Ping de L2 SRAM vers L1D SRAM
- **Étape 5** : Si traitement algorithmique terminé, sauvegarde des données de sortie **yk_sp** du côté Pong de L1D SRAM vers L2 SRAM
- **Étape 6** : Si traitement algorithmique terminé, sauvegarde des données de sortie **yk_sp** du côté Ping de L1D SRAM vers L2 SRAM

- **Voilà, pour cette année la trame de travaux pratiques s'arrête officiellement ici !** Si vous le souhaitez et si vous avez le temps, vous pouvez tenter d'implémenter la stratégie **Ping Pong**. Bien entendu, cela entraînera un refactoring non négligeable de code.

5.3. Transfert par EDMA



Cette dernière partie dépasse les attentes minimales de la trame d'enseignement et n'est à réaliser que sur volonté propre. Afin de parfaire notre maîtrise de la machine, il resterait également à configurer l'EDMA (DMA système partagé entre cœurs) et enfin pour finaliser l'ensemble, développer un scheduler permettant de paralléliser les copies mémoires (EDMA/IDMA L2SRAM/L1SRAM) des calculs algorithmiques (fir_sp_r4). Il pourrait d'ailleurs s'agir d'un travail de stage et cela nécessiterait des semaines de développement et de test. En revanche, la copie simple (linéaire contiguë) par EDMA peut se réaliser en quelques heures de développement. Voici donc le dernier exercice proposé.



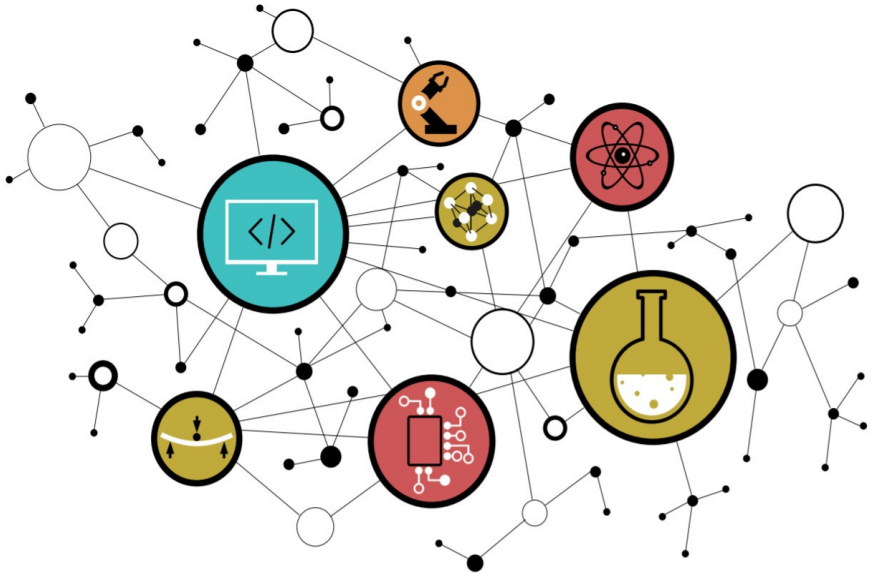
- Ajouter à votre projet la fonction disco/c6678/edmalib/src/edmacopy.c et valider la bonne compilation du projet. Résoudre les erreurs potentielles de compilation.

- En vous aidant de la documentation technique de CSL (Chip Support Library ou bibliothèque de fonctions pilotes des DSP C6000) présente dans le répertoire de projet /disco/tp/doc/csl/README.md et de la partie propre à l'EDMA3, compléter la fonction edmacopy de façon à remplacer la fonction memcopy précédemment utilisée. *Bonne chance !*


notes

TRAVAUX PRATIQUES

PREPARATIONS DES TRAVAUX PRATIQUES



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

1. PRÉLUDE

2. PROGRAMMATION VECTORIELLE SUR DSP VLIW C6600

4. MÉMOIRE CACHE ET MÉMOIRE SRAM ADRESSABLE

5. PÉRIPHÉRIQUES DE COPIE MÉMOIRE DMA

1. Prélude

Lire le document de prélude et installer les outils de développement !

1. (2pts) Il existe deux grandes familles de filtres numériques, les filtres FIR (Finite Impulse Response) et IIR (Infinite Impulse Response). Quels sont les avantages et inconvénients des filtres FIR et citer des exemples d'application dans lesquels nous pouvons les rencontrer ?

2. (1pt) Quel est l'ordre du filtre numérique implémenté en TP ? *A titre indicatif, il s'agit du même ordre de grandeur que celui rencontré dans une chaîne radar réelle pour la réalisation d'une convolution discrète*

3. (2pt) Que signifie, développement guidé par le test ? *S'aider d'internet*

4. (1pt) Quel est le rôle du qualificateur de type const ? *Ne pas répondre ''définir une constante''*

5. (1pt) Quel est le rôle des directives pré-processeur #if et #endif ? *Donner des exemples d'utilisation*

6. (1pt) Quel est le rôle du qualificateur de type restrict ? Expliquer dans quels cas nous pouvons l'utiliser ainsi que son utilité dans le cadre de cette trame d'enseignement ? *S'aider d'internet*

7. (2pt) Rappeler ce qu'est un Timer pour un processeur numérique (MCU, DSP, GPP, etc) ?

2. Programmation vectorielle sur DSP VLIW C6600

1. (2pt) Chez Texas Instruments, qu'est-ce que CSL (Chip Support Library) ?

2. (2pt) Chaque cœur de notre DSP possède un timer 64 bits mis à zéro au reset, extrêmement simple d'utilisation et travaillant à la fréquence de travail du cœur (soit 1,4Ghz dans notre cas). Ces timers se nomment TSC (Time Stamp Counter) et servent typiquement à de la mesure de performance. Pour information, tout processeur GPP Intel actuel possède également un timer nommé TSC dans chaque cœur. En vous aidant de la documentation technique de CSL présente dans le répertoire de projet tp/doc/csl/README.md , détailler l'API (Application Programming Interface) proposée par TI afin d'utiliser ces timers ? Préciser le rôle de chaque fonction ?

3. (6pt) Proposer un code ASM C6000 canonique (sans optimisation) implémentant la fonction de filtrage FIR réalisée en TP mais cette fois au format entier 16bits en représentation virgule fixe Q1.15. Utiliser les instructions entières 16bits suivantes et s'aider de la solution au format flottant proposée en cours. Attention un piège se glisse dans les multiplications entières signées. *S'aider d'internet* :

- MPY
- ADD
- SUB
- LDH
- STH
- SHR

S'aider également de la vidéo Texas Instruments ci-dessous concernant leur architecture C6000 : C6000 architecture (2 of 15)

https://training.ti.com/c6000-architecture-2-15?context=1134423-1134325

```
.global fir_q15_asm

; C prototype :                              match registers :
;
; void fir_q15_asm ( const short * restrict xk,          -> A4
;                    const short * restrict a,           -> B4
;                    short* restrict yk,                 -> A6
;                    short na,                           -> B6
;                    short nyk);                         -> A8
fir_q15_asm:
            ; user code

            B           B3
            NOP         5
```

4. Mémoire cache et mémoire SRAM adressable

1. **(4pt)** Rappeler le principe de fonctionnement d'une mémoire cache. *Illustrer votre réponse à l'aide d'un schéma commenté*

2. **(2pt)** Qu'est-ce-qu'un contrôleur cache et quel est son rôle ? *Illustrer votre réponse à l'aide d'un schéma commenté*

3. **(2pt)** Qu'est-ce-qu'une ligne de cache ? *Illustrer votre réponse à l'aide d'un schéma commenté*

4. **(2pt)** Comme pour beaucoup d'architectures actuelles, la famille DSP C6000 utilise une politique de remplacement de ligne de cache du type LRU (Least Recently Used). Que cela signifie-t-il ? *Illustrer votre réponse à l'aide d'un schéma commenté*

5. Périphériques de copie mémoire DMA

1. (3pt) Qu'est-ce-qu'un DMA ? *Illustrer votre réponse à l'aide d'un schéma commenté*

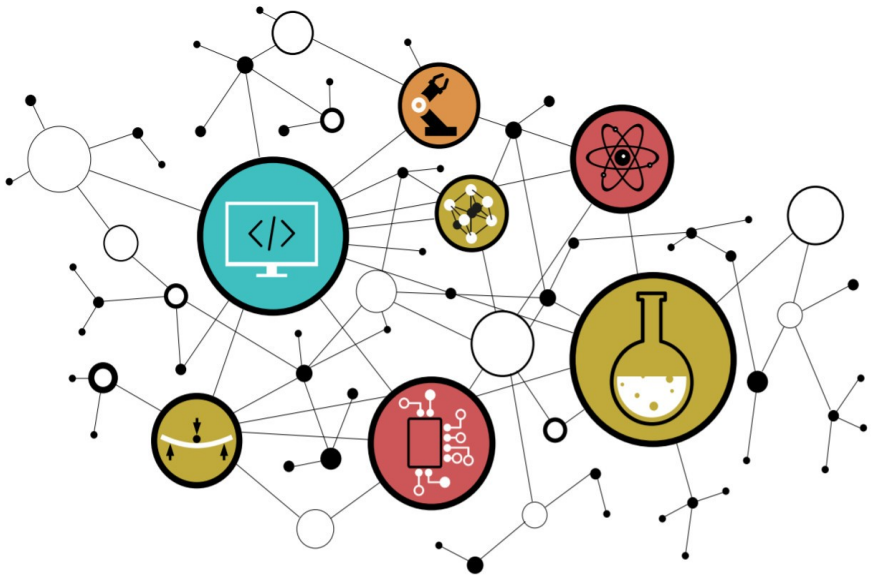
2. (2pt) Qu'est-ce qu'un canal DMA (DMA channel) pour un DMA ? *Illustrer votre réponse à l'aide d'un schéma commenté*

3. (2pt) En s'aidant de la documentation technique présente dans le répertoire tp/doc/datasheet/datasheet - corepac – sprugw0c.pdf, combien de canaux possède chaque IDMA de notre processeur. De même, combien d'IDMA possède notre processeur DSP C6678 ?

4. (3pt) En vous aidant de la documentation technique de CSL tp/doc/csl/README.md, présenter l'API utile à l'utilisation du canal 0 de l'IDMA. Expliquer son fonctionnement fonction par fonction ?

TRAVAUX PRATIQUES

ANNEXES



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

1. PRÉSENTATION DE CCSTUDIO

2. CRÉATION D'UNE BIBLIOTHÈQUE STATIQUE

3. EXTRAITS DATASHEET – SPRU187V – OPTIMIZING COMPILER

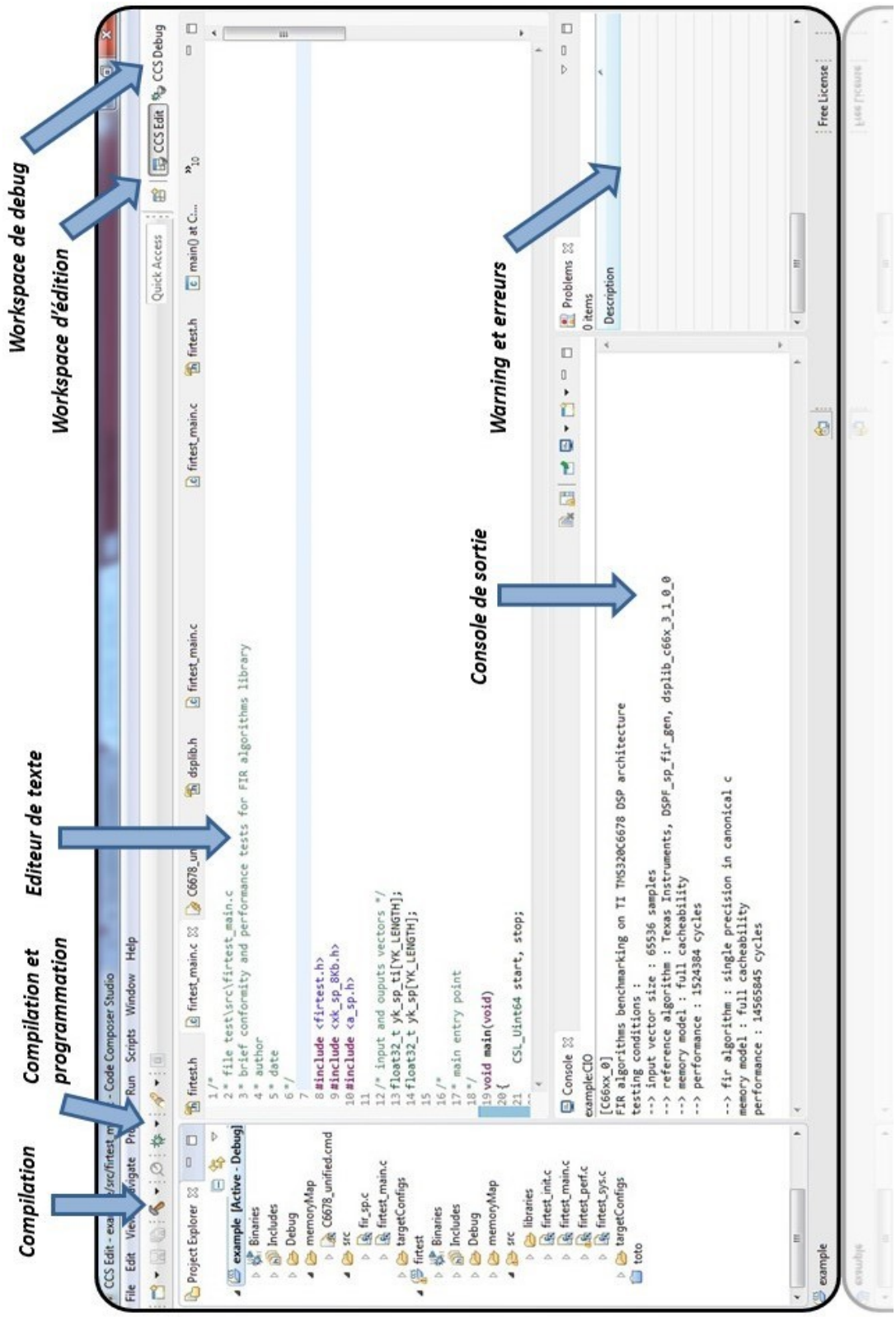
4. EXTRAITS DATASHEET – SPRS691E – TMS320C6678

5. EXTRAITS DATASHEET – SPRABK5A1 – THROUGHPUT PERFORMANCE

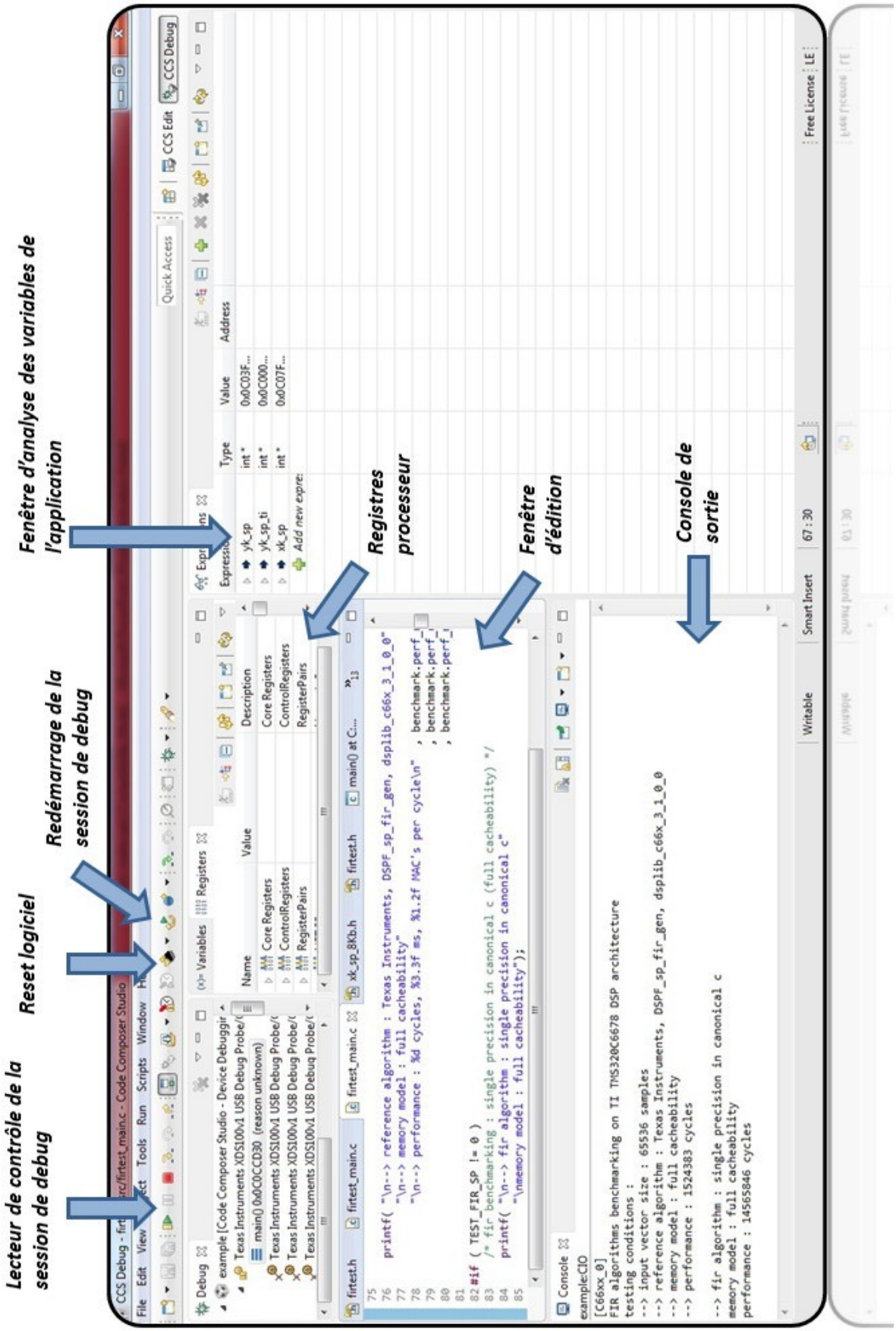
6. EXTRAITS DATASHEET – SPRUGH7 – CPU AND INSTRUCTION SET

1. PRÉSENTATION DE CCSTUDIO

Présentation du **workspace d'édition** proposé par l'IDE Code Composer Studio. Le framework présenté est basé sur le plugin CDT (C/C++ Development Tools) classiquement utilisé sous IDE éclipse pour du développement C/C++.



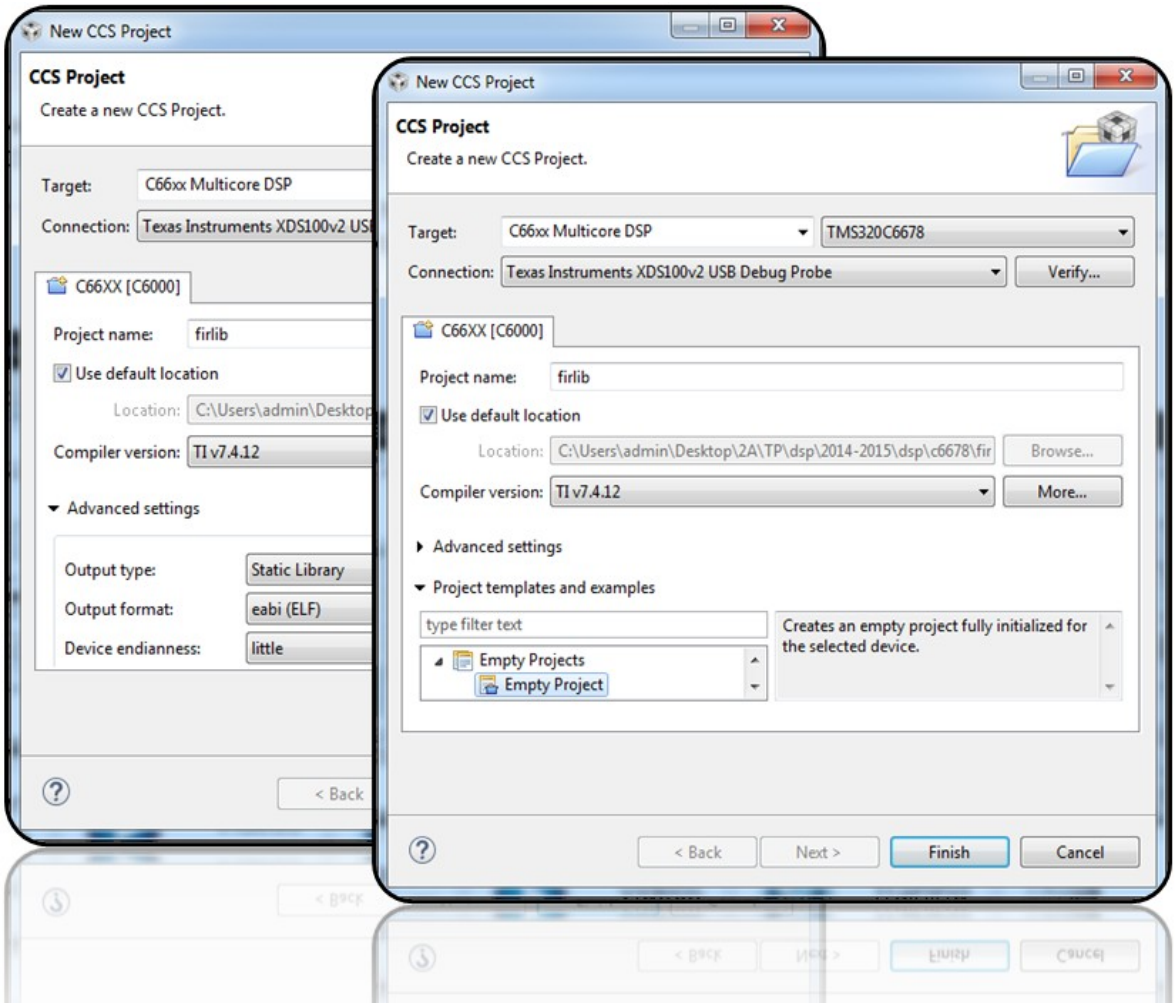
Présentation du **workspace de debug**, d'analyse et de communication avec la sonde de programmation XDS100 (carré rouge pour le fermer ce workspace). **Ne pas éditer dans cet espace de travail (bug non corrigé sur cette version de l'IDE)** !



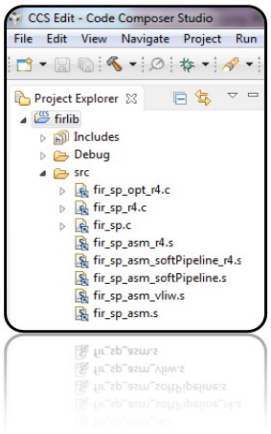
2. CRÉATION D'UNE BIBLIOTHÈQUE STATIQUE

Nous allons donc maintenant nous intéresser au processus de génération de **bibliothèque statique** sous Code Composer Studio (environnement Eclipse). Rappelons qu'une bibliothèque statique n'est qu'une archive (concaténation) de fichiers binaires objets pré-compilés (fichiers ELF dans le cas de notre ABI). Toujours préférer une fonction par fichier source plutôt que d'inclure toutes les fonctions d'une bibliothèque dans un seul fichier avant compilation. Ainsi par la suite, le linker sera apte à n'inclure à l'édition des liens que les binaires pré-compilés (objets relogeables) des fonctions réellement appelées par l'applicatif et non tout le contenu de la bibliothèque statique. Sélection des fichiers objets utiles.

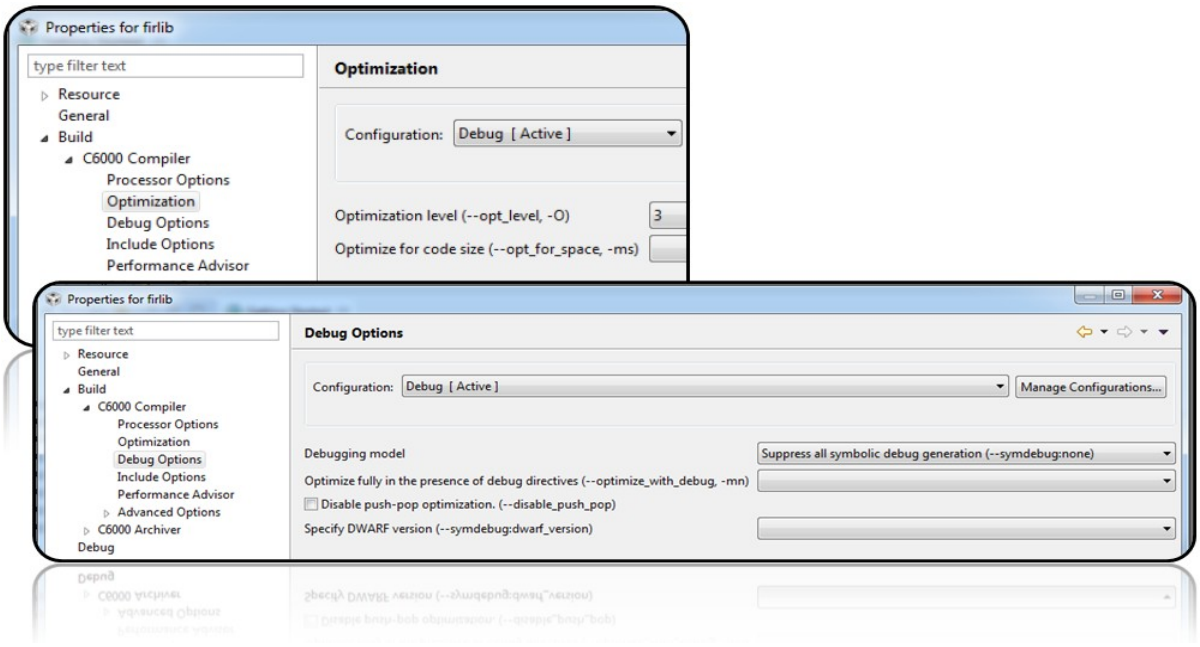
- Créer un nouveau projet dans un nouveau répertoire propre à la génération de la bibliothèque statique. Dans le cadre de cette trame de travaux pratiques, placer ce projet par exemple dans **/disco/c6678/firlib/pjct/**. Sélectionner néanmoins dans les options avancées le format du fichier de sortie (sélection de l'archiver plutôt que du linker) : **Advanced settings > Static Library**



- Créer un répertoire logique src et y placer les sources propres à la génération de la bibliothèque statique. N'inclure que les fonctions de la bibliothèque et retirer tous les sources relatifs au test (cf. ci-dessous)



- Dans les options de compilation du projet, lever les options d'optimisation -O3 et couper toute forme de génération de code de Debug :



- Compiler le projet et voilà, c'est fini. Effectuer une recherche Windows ou GNU\Linux afin de rechercher le fichier nom_projet.lib puis le copier dans le répertoire /disco/c6678/firlib/lib/.

```
'Building target: firlib.lib'
'Invoking: C6000 Archiver'
"C:/ti/ccsv6/tools/compiler/c6000_7.4.12/bin/ar6x" r "firlib.lib" "./src/fir_sp.obj" "./src/fir_sp_asm.obj" "./src/fir_sp_asm_r4.obj"
"./src/fir_sp_asm_softPipeline.obj" "./src/fir_sp_asm_softPipeline_r4.obj" "./src/fir_sp_asm_vliw.obj" "./src/fir_sp_opt_r4.obj" "./src/fir_sp_r4.obj"
  ==>  new archive 'firlib.lib'
  ==>  building archive 'firlib.lib'
'Finished building target: firlib.lib'
' '

**** Build Finished ****
```

- Dans vos futurs projets, vous pourrez maintenant retirer les fichiers sources des projets pour n'inclure que la bibliothèque statique (options de compilation, partie linker)

3. EXTRAITS DATASHEET – SPRU187V – OPTIMIZING COMPILER

Le schéma ci-dessous présente le workflow typique de la chaîne de compilation C6000 développée par Texas Instruments

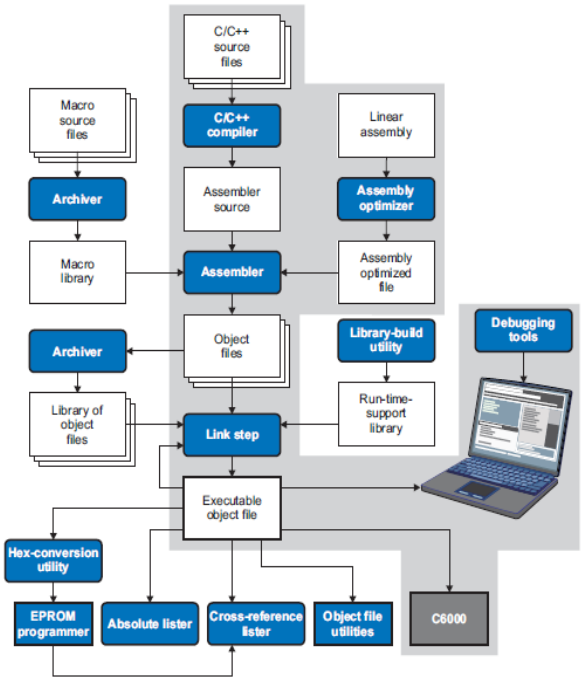


Table 6-2. TMS320C6000 C/C++ EABI Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	8 bits	ASCII	-128	127
unsigned char _Bool, bool	8 bits	ASCII	0	255
short	16 bits	2s complement	-32 768	32 767
unsigned short, wchar_t (1)	16 bits	Binary	0	65 535
int, signed int	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
__int40_t	40 bits	2s complement	-549 755 813 888	549 755 813 887
unsigned __int40_t	40 bits	Binary	0	1 099 511 627 775

(1) This is the default type for wchar_t. You can use the --wchar_t option to change the wchar_t type to a 32-bit unsigned int type.

Table 6-2. TMS320C6000 C/C++ EABI Data Types (continued)

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum (2)	32 bits	2s complement	-2 147 483 648	2 147 483 647
float	32 bits	IEEE 32-bit	1.175 494e-38 (3)	3.40 282 346e+38
float complex (4)	64 bits	Array of 2 IEEE 32-bit	1.175 494e-38 for real and imaginary portions separately	3.40 282 346e+38 for real and imaginary portions separately
double	64 bits	IEEE 64-bit	2.22 507 385e-308 (3)	1.79 769 313e+308
double complex (4)	128 bits	Array of 2 IEEE 64-bit	2.22 507 385e-308 for real and imaginary portions separately	1.79 769 313e+308 for real and imaginary portions separately
long double	64 bits	IEEE 64-bit	2.22 507 385e-308 (3)	1.79 769 313e+308
long double complex (4)	128 bits	Array of 2 IEEE 64-bit	2.22 507 385e-308 for real and imaginary portions separately	1.79 769 313e+308 for real and imaginary portions separately
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

(2) For details about the size of an enum type, see Section 6.4.1.
(3) Figures are minimum precision.
(4) To use complex data types, you must include the <complex.h> header file. See Section 6.5.1 for more about complex data types.

Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations. See Section 7.1.1 for a complete description of how the compiler uses these sections.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 5-1 summarizes the initialized sections created under the COFF ABI mode. Table 5-2 summarizes the initialized sections created under the EABI mode. Table 5-3 summarizes the uninitialized sections. Be aware that the COFF ABI .cinit and .pinit (.init_array in EABI) tables have different formats in EABI.

Table 5-1. Initialized Sections Created by the Compiler for COFF ABI

Name	Contents
.args	Command argument for host-based loader; read-only (see the --arg_size option).
.cinit	Tables for explicitly initialized global and static variables.
.const	Global and static const variables, including string constants and initializers for local variables.
.ppdata	Data tables for compiler-based profiling (see the --gen_profile_info option).
.ppinfo	Correlation tables for compiler-based profiling (see the --gen_profile_info option).
.switch	Jump tables for large switch statements.
.text	Executable code and constants.

Table 5-2. Initialized Sections Created by the Compiler for EABI Only

Name	Contents
.binit	Boot time copy tables (See the *TMS320C6000 Assembly Language Tools User's Guide* for information on BINIT in linker command files.)
.cinit	In EABI mode, the compiler does not generate a .cinit section. However, when the --rom_mode linker option is specified, the linker creates this section, which contains tables for explicitly initialized global and static variables.
.c6xabi.exidx	Index table for exception handling; read-only (see --exceptions option).
.c6xabi.extab	Unwinded instructions for exception handling; read-only (see --exceptions option).
.data	Global and static non-const variables that are explicitly initialized.
.fardata	Far non-const global and static variables that are explicitly initialized.
.init_array	Table of constructors to be called at startup.
.name.load	Compressed image of section *name*; read-only (See the *TMS320C6000 Assembly Language Tools User's Guide* for information on copy tables.)
.neardata	Near non-const global and static variables that are explicitly initialized.
.rodata	Global and static variables that have near and const qualifiers.

Table 5-3. Uninitialized Sections Created by the Compiler for Both ABIs

Name	Contents
.bss	Uninitialized global and static variables
.cio	Buffers for stdio functions from the run-time support library
.far	Global and static variables declared far
.stack	Stack
.sysmem	Memory pool (heap) for dynamic memory allocation (malloc, etc)

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. With the exception of code sections, the initialized and uninitialized sections created by the compiler cannot be allocated into internal program memory.

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the *TMS320C6000 Assembly Language Tools User's Guide*.

Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. The C6000 C/C++ compiler supports the following pragmas:

- CHECK_MISRA (See Section 6.9.1)
- CLINK (See Section 6.9.2)
- CODE_SECTION (See Section 6.9.3)
- DATA_ALIGN (See Section 6.9.4)
- DATA_MEM_BANK (See Section 6.9.5)
- DATA_SECTION (See Section 6.9.6)
- diag_suppress, diag_remark, diag_warning, diag_error, and diag_default (See Section 6.9.7)
- FUNC_ALWAYS_INLINE (See Section 6.9.8)
- FUNC_CANNOT_INLINE (See Section 6.9.9)
- FUNC_EXT_CALLED (See Section 6.9.10)
- FUNC_INTERRUPT_THRESHOLD (See Section 6.9.11)
- FUNC_IS_PURE (See Section 6.9.12)
- FUNC_IS_SYSTEM (See Section 6.9.13)
- FUNC_NEVER_RETURNS (See Section 6.9.14)
- FUNC_NO_GLOBAL_ASG (See Section 6.9.15)
- FUNC_NO_IND_ASG (See Section 6.9.16)
- FUNCTION_OPTIONS (See Section 6.9.17)
- INTERRUPT (See Section 6.9.18)
- LOCATION (EABI only; see Section 6.9.19)
- MUST_ITERATE (See Section 6.9.20)
- NMI_INTERRUPT (See Section 6.9.21)
- NO_HOOKS (See Section 6.9.22)
- PACK (See Section 6.9.23)
- PROB_ITERATE (See Section 6.9.24)
- RESET_MISRA (See Section 6.9.25)
- RETAIN (See Section 6.9.26)
- SET_CODE_SECTION (See Section 6.9.27)
- SET_DATA_SECTION (See Section 6.9.27)
- STRUCT_ALIGN (See Section 6.9.28)
- UNROLL (See Section 6.9.29)

The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* in C, or the next symbol declared in C++, to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2. The maximum alignment is 32768.

The DATA_ALIGN pragma cannot be used to reduce an object's natural alignment.

The syntax of the pragma in C is:

#pragma DATA_ALIGN (*symbol* **,** *constant* **)**

Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 7-2 summarizes how the compiler uses the TMS320C6000 registers.

The registers in Table 7-2 are available to the compiler for allocation to register variables and temporary expression results. If the compiler cannot allocate a register of a required type, spilling occurs. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

Objects of type double, long, long long, or long double are allocated into an odd/even register pair and are always referenced as a register pair (for example, A1:A0). The odd register contains the sign bit, the exponent, and the most significant part of the mantissa. The even register contains the least significant part of the mantissa. The A4 register is used with A5 for passing the first argument if the first argument is a double, long, long long, or long double. The same is true for B4 and B5 for the second parameter, and so on. For more information about argument-passing registers and return registers, see Section 7.4.

Table 7-2. Register Usage

Register	Function Preserved By	Special Uses	Register	Function Preserved By	Special Uses
A0	Parent	–	B0	Parent	–
A1	Parent	–	B1	Parent	–
A2	Parent	–	B2	Parent	–
A3	Parent	Structure register (pointer to a returned structure)(1)	B3	Parent	Return register (address to return to)
A4	Parent	Argument 1 or return value	B4	Parent	Argument 2
A5	Parent	Argument 1 or return value with A4 for doubles, longs and long longs	B5	Parent	Argument 2 with B4 for doubles, longs and long longs
A6	Parent	Argument 3	B6	Parent	Argument 4
A7	Parent	Argument 3 with A6 for doubles, longs, and long longs	B7	Parent	Argument 4 with B6 for doubles, longs, and long longs
A8	Parent	Argument 5	B8	Parent	Argument 6
A9	Parent	Argument 5 with A8 for doubles, longs, and long longs	B9	Parent	Argument 6 with B8 for doubles, longs, and long longs
A10	Child	Argument 7	B10	Child	Argument 8
A11	Child	Argument 7 with A10 for doubles, longs, and long longs	B11	Child	Argument 8 with B10 for doubles, longs, and long longs
A12	Child	Argument 9	B12	Child	Argument 10
A13	Child	Argument 9 with A12 for doubles, longs, and long longs	B13	Child	Argument 10 with B12 for doubles, longs, and long longs
A14	Child	–	B14	Child	Data page pointer (DP)
A15	Child	Frame pointer (FP)	B15	Child	Stack pointer (SP)
A16-A31	Parent	C6400, C6400+, C6700+, C6740, and C6600 only	B16-B31	Parent	C6400, C6400+, C6700+, C6740, and C6600 only
ILC	Child	C6400+, C6740, and C6600 only, loop buffer counter	NRP	Parent	
IRP	Parent		RILC	Child	C6400+, C6740, and C6600 only, loop buffer counter

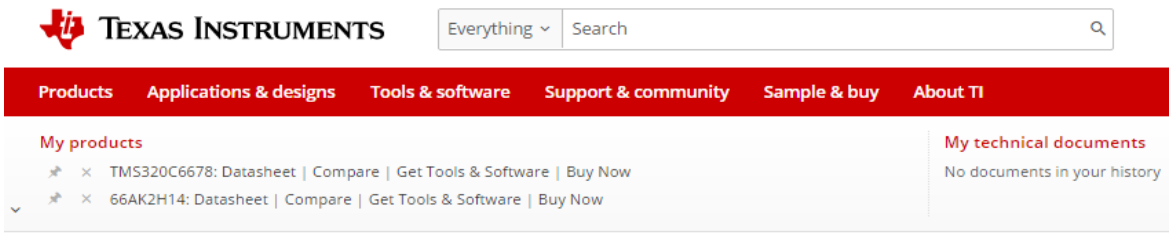
(1) For EABI, structs of size 64 or less are passed by value in registers instead of by reference using a pointer in A3.

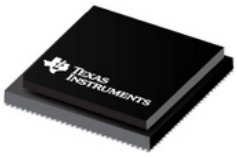
Figure 7-10. Register Argument Conventions

```
int func1( int a,    int b,      int c);
A4         A4        B4          A6
int func2( int a,    float b,    int c)        struct A d,     float e,   int f,   int g);
A4         A4        B4          A6            B6              A8         B8       A10
int func3( int a,    double b,   float c)      long double d);
A4         A4        B5:B4       A6            B7:B6
/*NOTE: The following function has a variable number of arguments. */
int vararg(int a,    int b,      int c,        int d);
A4         A4        B4          A6            stack
struct A func4(      int y);
A3                   A4
__x128_t func5(        __x128_t a);
A7:A6:A5:A4          A7:A6:A5:A4
void func6(int a,    int b,      __x128_t c);
           A4        B4          A11:A10:A9:A8
void func7(int a,    int b,      __x128_t c,   int d,          int e,     int f,   __x128_t g,  int h);
           A4        B4          A11:A10:A9:A8 A6              B6         B8       stack        B10
```

4. EXTRAITS DATASHEET – SPRS691E – TMS320C6678

http://www.ti.com/product/tms320c6678





TI Home > Semiconductors > Processors > Digital Signal Processors > C6000 DSP > C66x DSP >

TMS320C6678 (ACTIVE)

Multicore Fixed and Floating-Point Digital Signal Processor

TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. E)

TMS320C6678 Multcore Fixed & Floating-Point DSP Silicon Errata (Revs 1.0, 2.0) (Rev. H)

http://www.ti.com/lit/ds/symlink/tms320c6678.pdf

1 TMS320C6678 Features and Description

1.1 Features

- Eight TMS320C66x™ DSP Core Subsystems (C66x CorePacs), Each with
  - 1.0 GHz, 1.25 GHz, or 1.4 GHz C66x Fixed/Floating-Point CPU Core
    - 44.8 GMAC/Core for Fixed Point @ 1.4 GHz
    - 22.4 GFLOP/Core for Floating Point @ 1.4 GHz
  - Memory
    - 32K Byte L1P Per Core
    - 32K Byte L1D Per Core
    - 512K Byte Local L2 Per Core
- Multicore Shared Memory Controller (MSMC)
  - 4096KB MSM SRAM Memory Shared by Eight DSP C66x CorePacs
  - Memory Protection Unit for Both MSM SRAM and DDR3_EMIF
- Multicore Navigator
  - 8192 Multipurpose Hardware Queues with Queue Manager
  - Packet-Based DMA for Zero-Overhead Transfers
- Network Coprocessor
  - Packet Accelerator Enables Support for
    - Transport Plane IPsec, GTP-U, SCTP, PDCP
    - L2 User Plane PDCP (RoHC, Air Ciphering)
    - 1-Gbps Wire-Speed Throughput at 1.5 MPackets Per Second
  - Security Accelerator Engine Enables Support for
    - IPSec, SRTP, 3GPP, WiMAX Air Interface, and SSL/TLS Security
    - ECB, CBC, CTR, F8, A5/3, CCM, GCM, HMAC, CMAC, GMAC, AES, DES, 3DES, Kasumi, SNOW 3G, SHA-1, SHA-2 (256-bit Hash), MD5
    - Up to 2.8 Gbps Encryption Speed
- Peripherals
  - Four Lanes of SRIO 2.1
    - 1.24/2.5/3.125/5 GBaud Operation Supported Per Lane
    - Supports Direct I/O, Message Passing
    - Supports Four 1×, Two 2×, One 4×, and Two 1× + One 2× Link Configurations
  - PCIe Gen2
    - Single Port Supporting 1 or 2 Lanes
    - Supports Up To 5 GBaud Per Lane
  - HyperLink
    - Supports Connections to Other KeyStone Architecture Devices Providing Resource Scalability
    - Supports up to 50 Gbaud
  - Gigabit Ethernet (GbE) Switch Subsystem
    - Two SGMII Ports
    - Supports 10/100/1000 Mbps Operation
  - 64-Bit DDR3 Interface (DDR3-1600)
    - 8G Byte Addressable Memory Space
  - 16-Bit EMIF
  - Two Telecom Serial Ports (TSIP)
    - Supports 1024 DS0s Per TSIP
    - Supports 2/4/8 Lanes at 32.768/16.384/8.192 Mbps Per Lane
  - UART Interface
  - I^2C Interface
  - 16 GPIO Pins
  - SPI Interface
  - Semaphore Module
  - Sixteen 64-Bit Timers
  - Three On-Chip PLLs
- Commercial Temperature:
  - 0°C to 85°C
- Extended Temperature:
  - -40°C to 100°C

Figure 2-17 C66x DSP Device Nomenclature (including the TMS320C6678)

TMS 320 C6678 (_) (_) CYP (_) (_)

PREFIX
TMX = Experimental device
TMS = Qualified device

DEVICE FAMILY
320 = TMS320 DSP family

DEVICE
C66x DSP: C6678

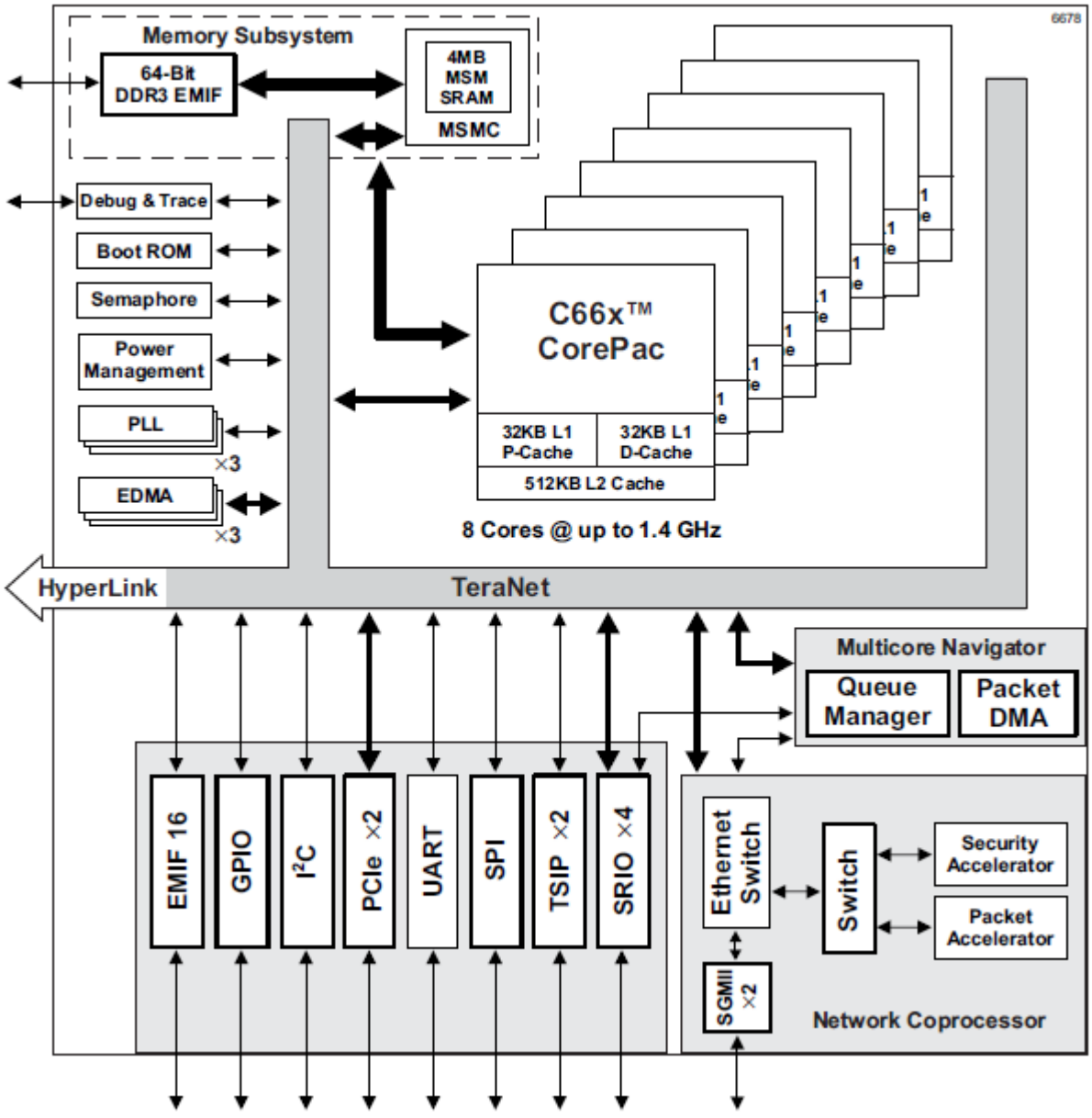
SILICON REVISION
Blank = Initial Silicon 1.0
A = Silicon Revision 2.0

DEVICE SPEED RANGE
Blank = 1 GHz
25 = 1.25 GHz
4 = 1.4 GHz

TEMPERATURE RANGE
Blank = 0°C to +85°C (default case temperature)
A = Extended temperature range (-40°C to +100°C)

PACKAGE TYPE
CYP = 841-pin plastic ball grid array, with Pb-free die bumps and Pb-free solder balls
GYP = 841-pin plastic ball grid array, with Pb-free die bumps and Pb-based solder balls

ENCRYPTION
Blank = Encryption NOT enabled
X = Encryption enabled

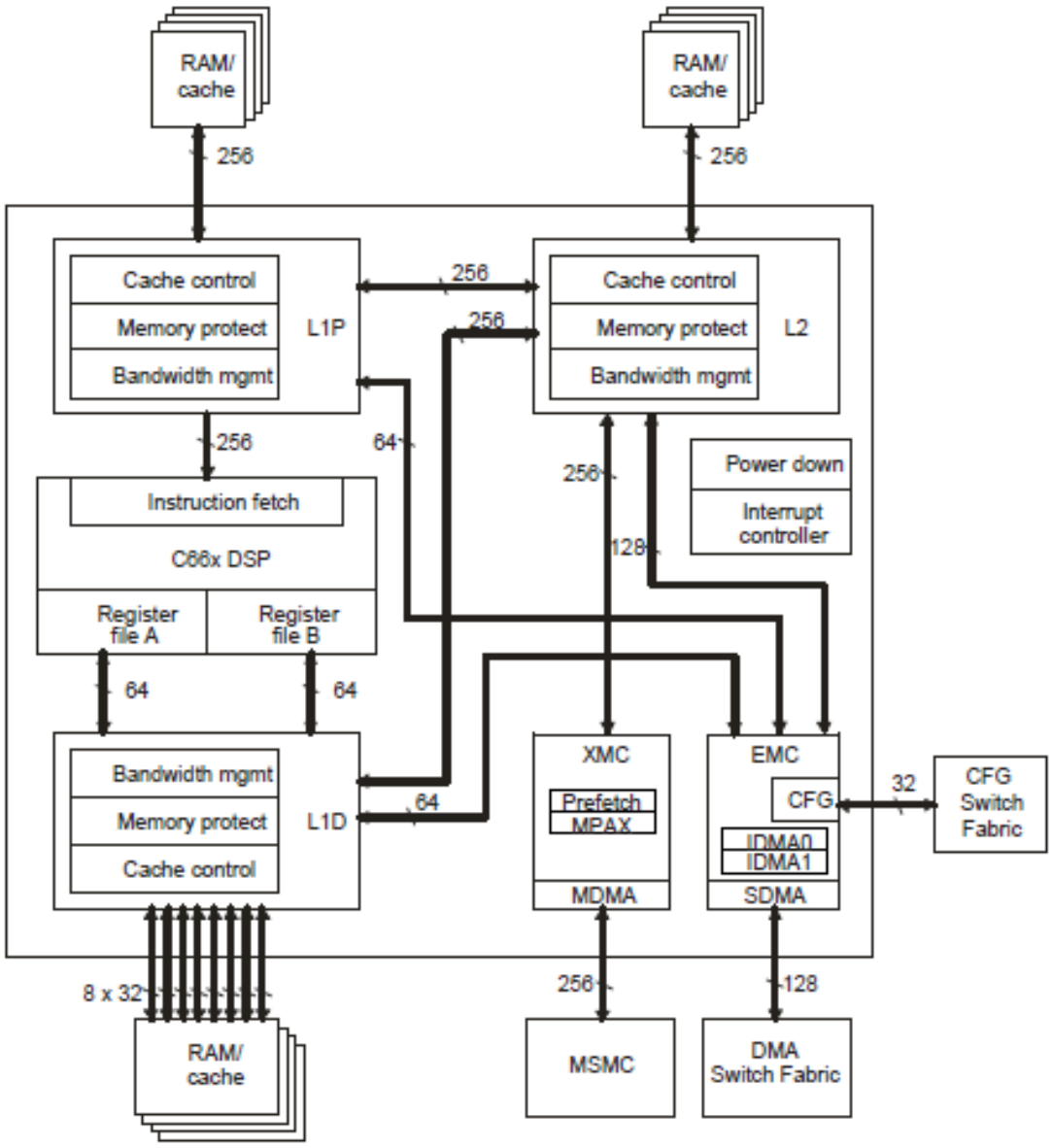


1.1 Introduction

C66x CorePac is the name used to designate the hardware that includes the following components: C66x DSP, Level 1 program (L1P) memory controller, Level 1 data (L1D) memory controller, Level 2 (L2) memory controller, Internal DMA (IDMA), external memory controller (EMC), extended memory controller (XMC), bandwidth management (BWM), interrupt controller (INTC) and powerdown controller (PDC).

A block diagram of the C66x CorePac is shown in Figure 1-1.

Figure 1-1 C66x CorePac Block Diagram



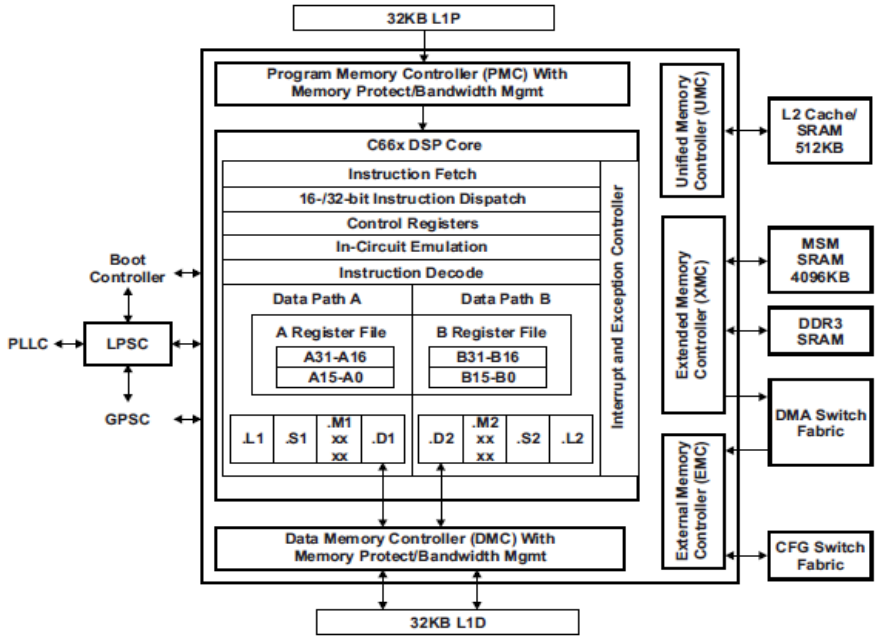
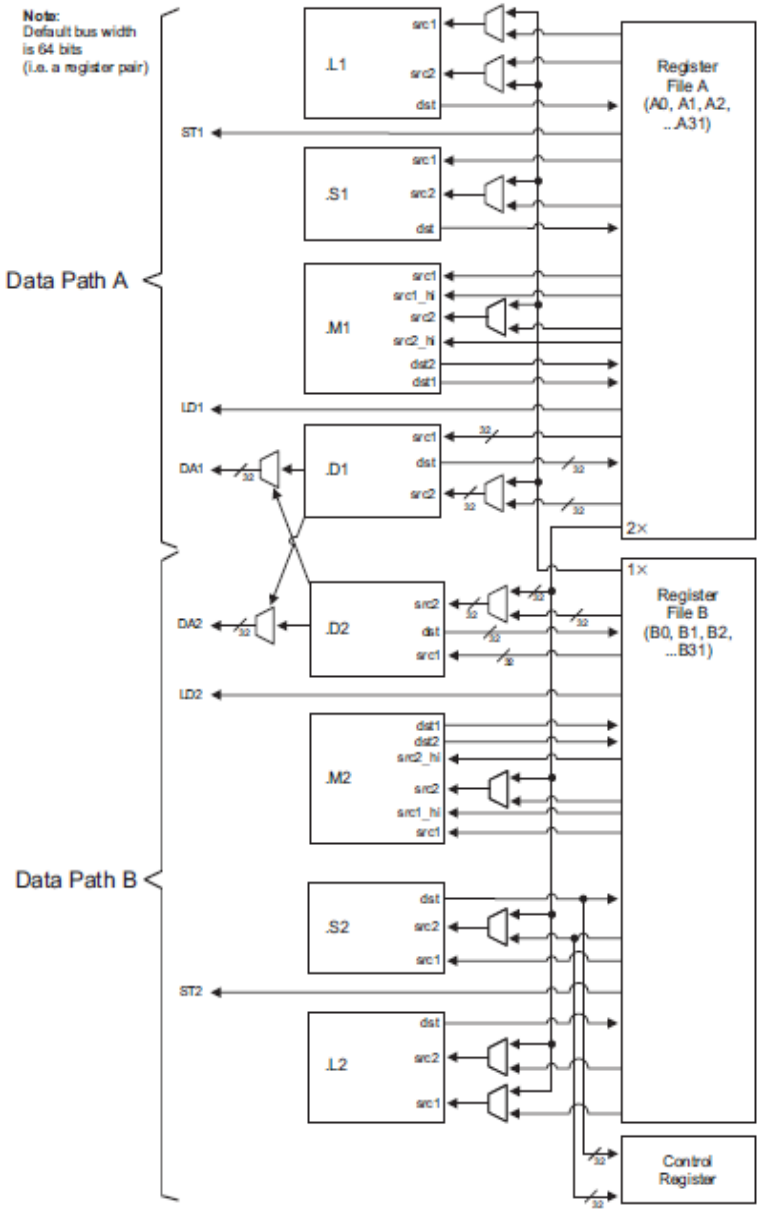


Figure 2-1 DSP Core Data Paths



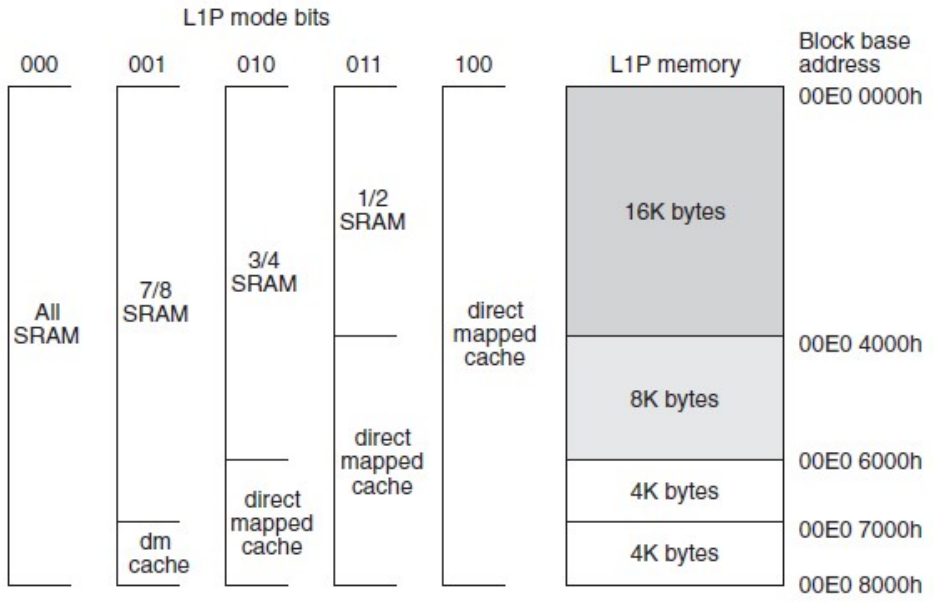
5.1.1 L1P Memory

The L1P memory configuration for the C6678 device is as follows:

- 32K bytes with no wait states

Figure 5-2 shows the available SRAM/cache configurations for L1P.

Figure 5-2 L1P Memory Configurations



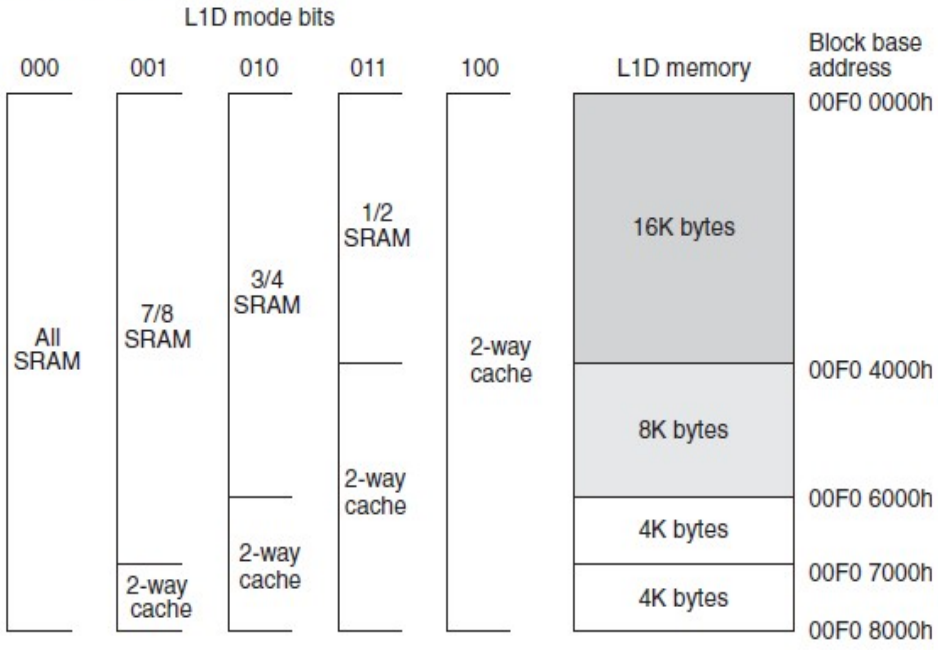
5.1.2 L1D Memory

The L1D memory configuration for the C6678 device is as follows:

- 32K bytes with no wait states

Figure 5-3 shows the available SRAM/cache configurations for L1D.

Figure 5-3 L1D Memory Configurations



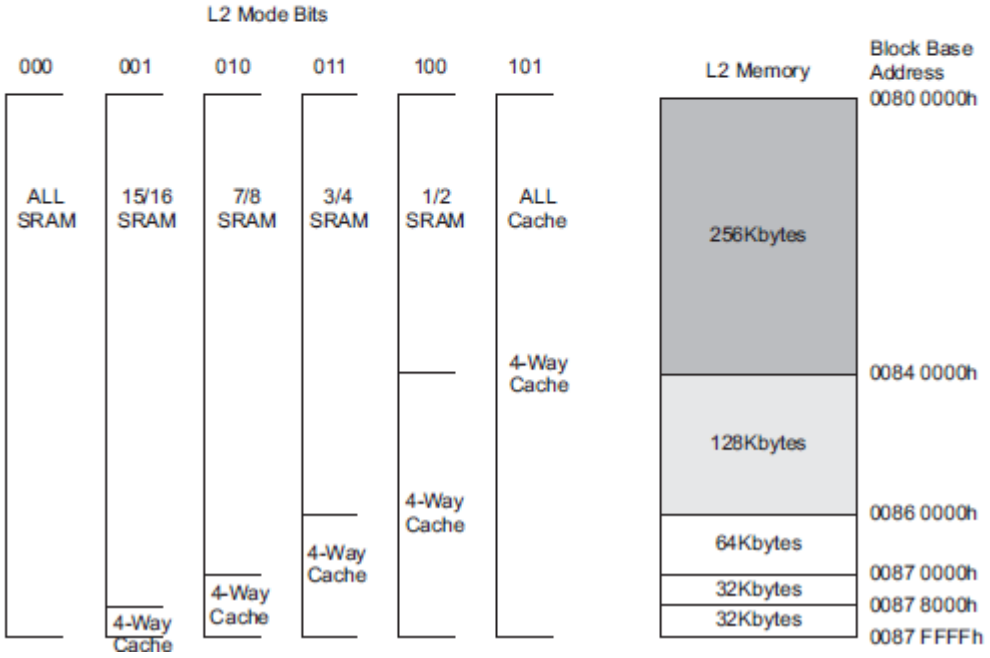
5.1.3 L2 Memory

The L2 memory configuration for the C6678 device is as follows:

- Total memory size is 4096KB
- Each core contains 512KB of memory
- Local starting address for each core is 0080 0000h

L2 memory can be configured as all SRAM, all 4-way set-associative cache, or a mix of the two. The amount of L2 memory that is configured as cache is controlled through the L2MODE field of the L2 Configuration Register (L2CFG) of the C66x CorePac. Figure 5-4 shows the available SRAM/cache configurations for L2. By default, L2 is configured as all SRAM after device reset.

Figure 5-4 L2 Memory Configurations



5.1.4 MSM SRAM

The MSM SRAM configuration for the C6678 device is as follows:

- Memory size is 4096KB
- The MSM SRAM can be configured as shared L2 and/or shared L3 memory
- Allows extension of external addresses from 2GB to up to 8GB
- Has built in memory protection features

The MSM SRAM is always configured as all SRAM. When configured as a shared L2, its contents can be cached in L1P and L1D. When configured in shared L3 mode, it's contents can be cached in L2 also. For more details on external memory address extension and memory protection features, see the *Multicore Shared Memory Controller (MSMC) for KeyStone Devices User Guide* in "Related Documentation from Texas Instruments" on page 72.

5.1.5 L3 Memory

The L3 ROM on the device is 128KB. The ROM contains software used to boot the device. There is no requirement to block accesses from this portion to the ROM.

5. EXTRAITS DATASHEET – SPRABK5A1 – THROUGHPUT PERFORMANCE

TeraNet and Memory Access Diagram

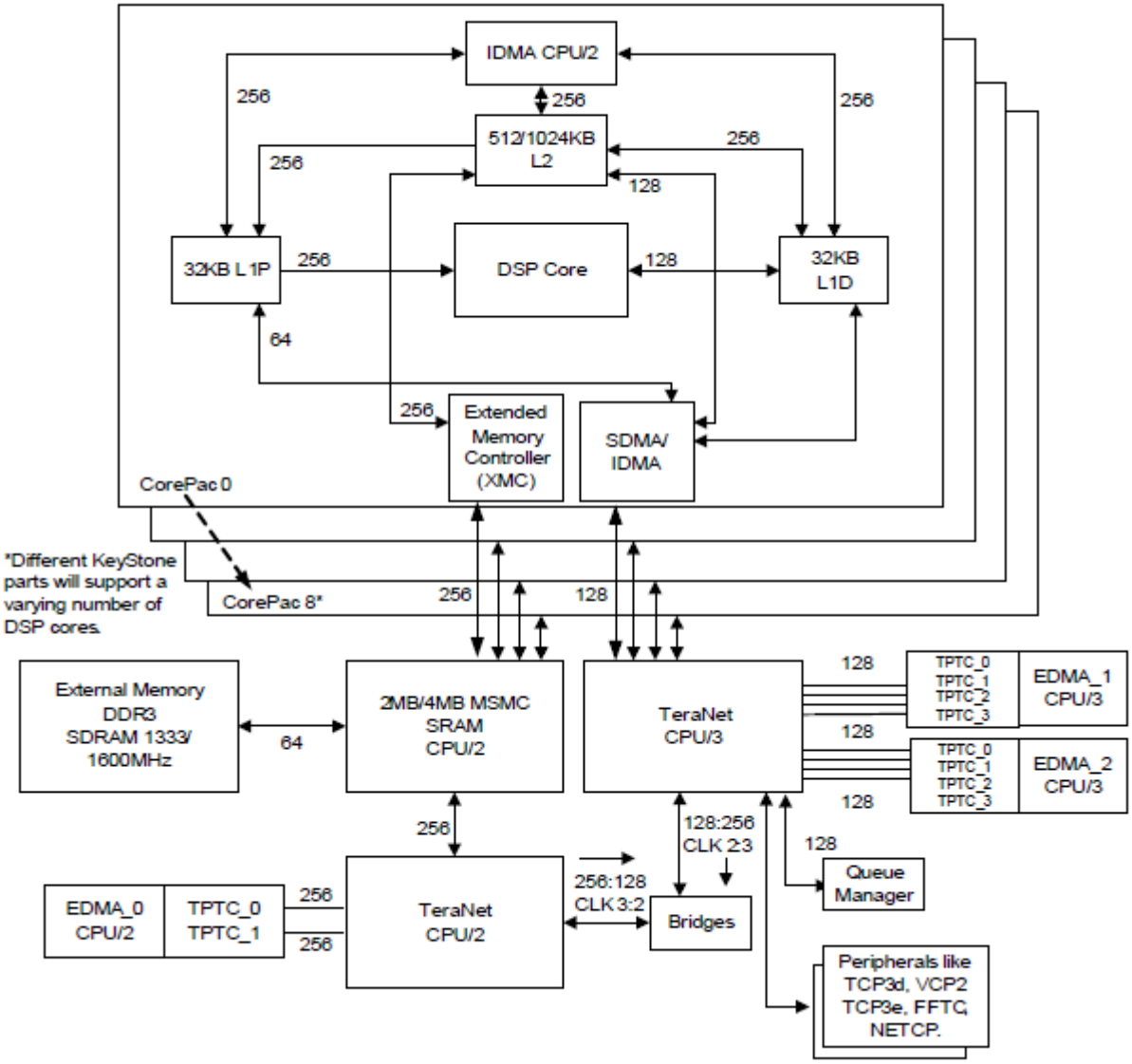


Table 2 Theoretical Bandwidth of Core, IDMA and EDMA

Master	Maximum Bandwidth MB/s	Comments
C66x Core	16000	(128bits)/(8bit/byte)*(1000M)=16000MB/s
IDMA	16000	(256bits)/(8bit/byte)*(1000M/2)=16000MB/s
EDMA0(Single TC)	16000	(256bits)/(8bit/byte)*(1000M/2)=16000MB/s
EDMA1(Single TC)	5333	(128bits)/(8bit/byte)*(1000M/3)=5333MB/s
EDMA2(Single TC)	5333	(128bits)/(8bit/byte)*(1000M/3)=5333MB/s

Table 3 Theoretical Bandwidth of Different Memories

Master	Maximum Bandwidth MB/s	Comments
L1D	32000	(256bits)/(8bit/byte)*(1000M)=32000MB/s
L1P	32000	(256bits)/(8bit/byte)*(1000M)=32000MB/s
L2	16000	(256bits)/(8bit/byte)*(1000M/2)=16000MB/s
MSMC RAM	64000	4*(256bits)/(8bit/byte)*(1000M/2)=64000MB/s
DDR3 RAM	10664	(64bits)/(8bit/byte)*(666.5M)*2=10664MB/s

Table 4 Memory Read Performance

				DSP Stalls (In Cycles)			
				Single Read		Burst Read	
Source	L1 Cache	L2 Cache	Prefetch	No Victim	Victim	No Victim	Victim
ALL	Hit	NA	NA	0	NA	0	NA
Local L2 SRAM	Miss	NA	NA	7	7	3.5	10
MSMC RAM (SL2)	Miss	NA	Hit	7.5	7.5	7.4	11
MSMC RAM (SL2)	Miss	NA	Miss	19.8	20.1	9.5	11.6
MSMC RAM (SL3)	Miss	Hit	NA	9	9	4.5	4.5
MSMC RAM (SL3)	Miss	Miss	Hit	10.6	15.6	9.7	129.6
MSMC RAM (SL3)	Miss	Miss	Miss	22	28.1	11	129.7
DDR RAM (SL2)	Miss	NA	Hit	9	9	23.2	59.8
DDR RAM (SL2)	Miss	NA	Miss	84	113.6	41.5	113
DDR RAM (SL3)	Miss	Hit	NA	9	9	4.5	4.5
DDR RAM (SL3)	Miss	Miss	Hit	12.3	59.8	30.7	287
DDR RAM (SL3)	Miss	Miss	Miss	89	123.8	43.2	183
End of Table 4							

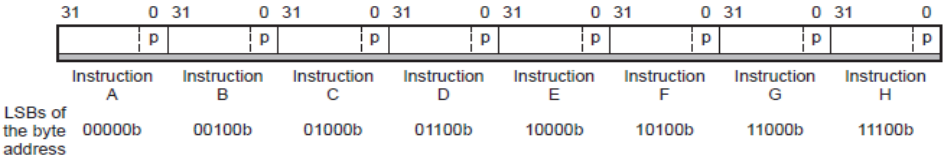
Table 5 Memory Write Performance

				DSP Stalls (In Cycles)			
				Single Write		Burst Write	
Source	L1 Cache	L2 Cache	Prefetch	No Victim	Victim	No Victim	Victim
ALL	Hit	NA	NA	0	NA	0	NA
Local L2 SRAM	Miss	NA	NA	0	0	1	1
MSMC RAM (SL2)	Miss	NA	Hit	0	0	2	2
MSMC RAM (SL2)	Miss	NA	Miss	0	0	2	2
MSMC RAM (SL3)	Miss	Hit	NA	0	0	3	3
MSMC RAM (SL3)	Miss	Miss	Hit	0	0	6.7	14.6
MSMC RAM (SL3)	Miss	Miss	Miss	0	0	6.7	16.7
DDR RAM (SL2)	Miss	NA	Hit	0	0	4.7	4.7
DDR RAM (SL2)	Miss	NA	Miss	0	0	5	5
DDR RAM (SL3)	Miss	Hit	NA	0	0	3	3
DDR RAM (SL3)	Miss	Miss	Hit	0	0	16	114.3
DDR RAM (SL3)	Miss	Miss	Miss	0	0	18.2	115.5
End of Table 5							

3.5 Parallel Operations

Instructions are always fetched eight words at a time. This constitutes a *fetch packet*. CPU, this may be as many as 14 instructions due to the existence of compact instructions in a header based fetch packet. The basic format of a fetch packet is shown in Figure 3-3. Fetch packets are aligned on 256-bit (8-word) boundaries.

Figure 3-3 Basic Format of a Fetch Packet



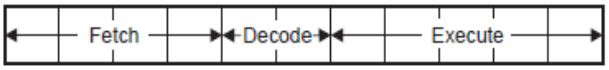
Pipeline Operation Overview

The pipeline phases are divided into three stages:

- Fetch
- Decode
- Execute

All instructions in the DSP pipeline are shown in Figure 5-1.

Figure 5-1 Pipeline Stages

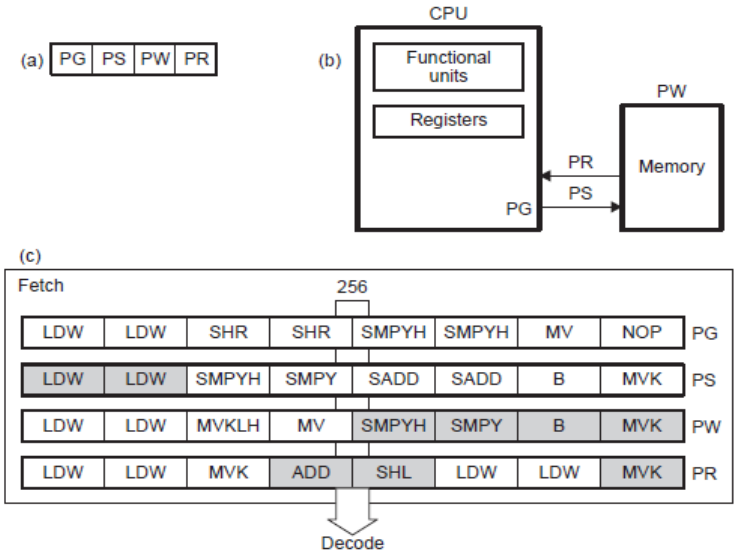


5.1.1 Fetch

The fetch phases of the pipeline are:

- **PG:** Program address generate
- **PS:** Program address send
- **PW:** Program access ready wait
- **PR:** Program fetch packet receive

Figure 5-2 Fetch Phases of the Pipeline

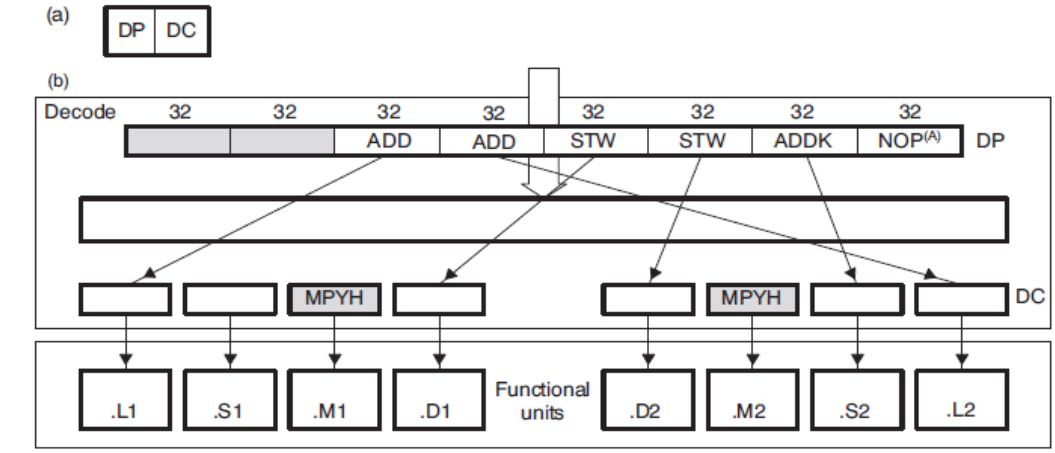


5.1.2 Decode

The decode phases of the pipeline are:

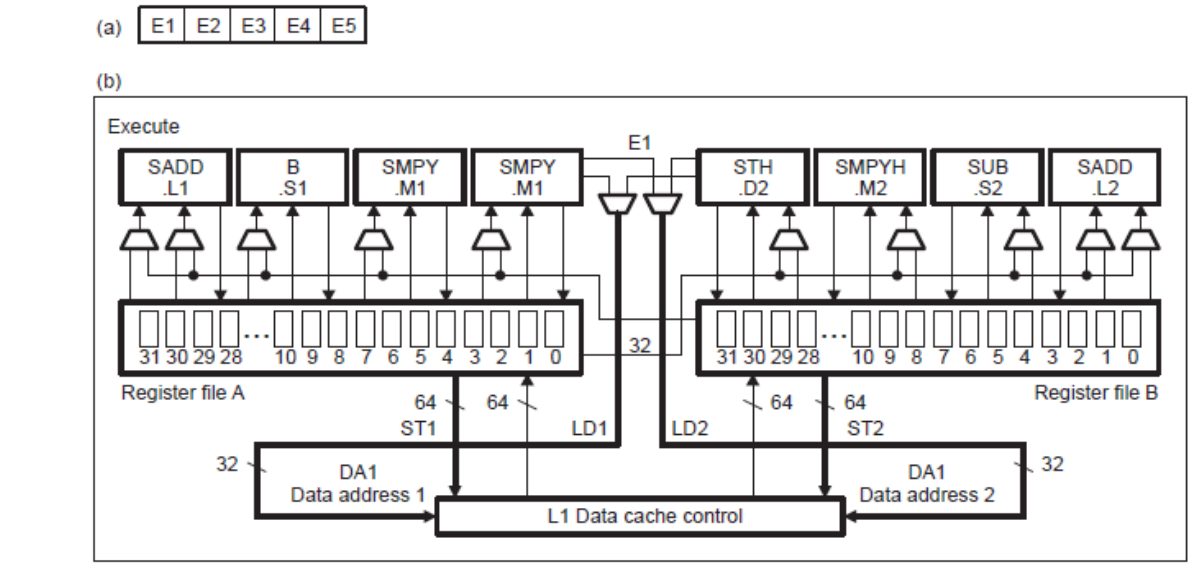
- **DP:** Instruction dispatch
- **DC:** Instruction decode

Figure 5-3 Decode Phases of the Pipeline



(A) NOP is not dispatched to a functional unit.

Figure 5-4 Execute Phases of the Pipeline



5.1.4 Pipeline Operation Summary

Figure 5-5 shows all the phases in each stage of the pipeline in sequential order, from left to right.

Figure 5-5 Pipeline Phases

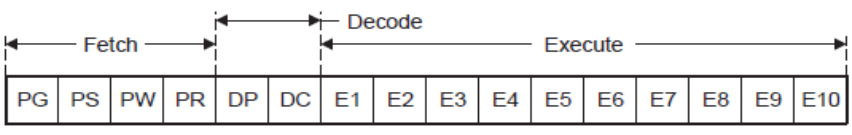
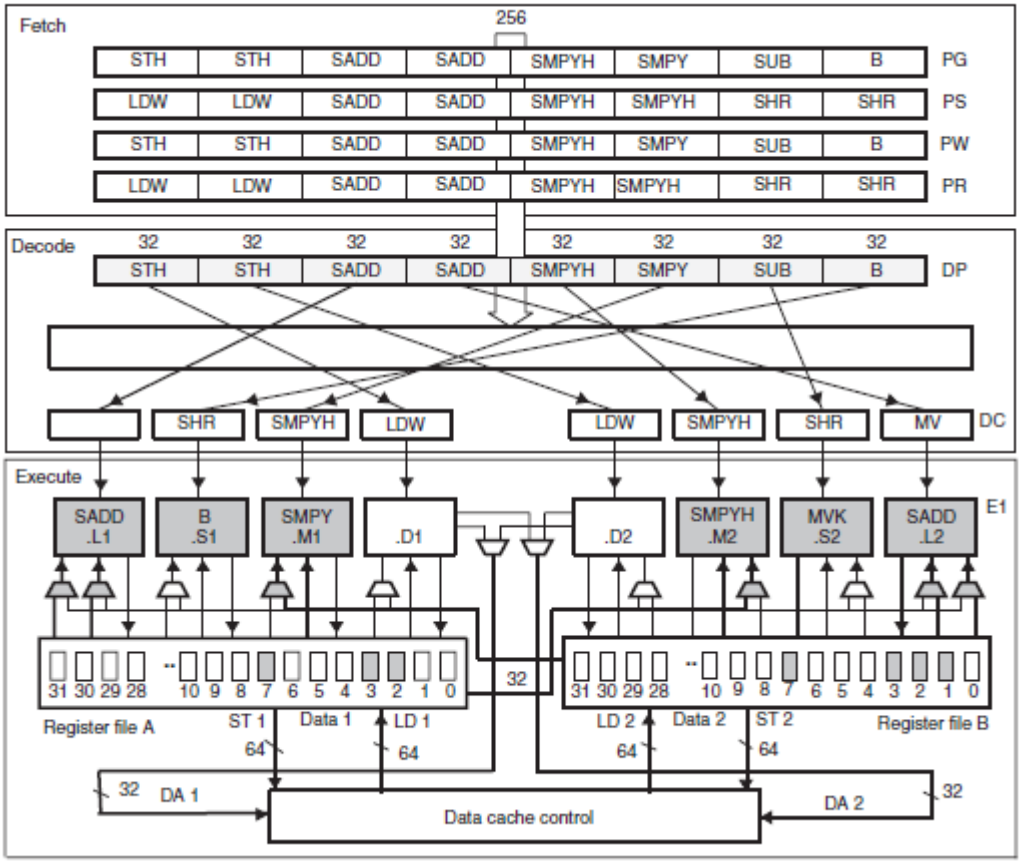


Table 5-1 Operations Occurring During Pipeline Phases (Part 2 of 2)

Stage	Phase	Symbol	During This Phase
Execute	Execute 1	E1	For all instruction types, the conditions for the instructions are evaluated and operands are read.
			For load and store instructions, address generation is performed and address modifications are written to a register file.[1]
			For branch instructions, branch fetch packet in PG phase is affected.[1]
			For single-cycle instructions, results are written to a register file.[1]
			For DP compare, ADDDP/SUBDP, and MPYDP instructions, the lower 32-bits of the sources are read. For all other instructions, the sources are read.[1]
			For MPYSPDP instruction, the *src1* and the lower 32 bits of *src2* are read.[1]
			For 2-cycle DP instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 2	E2	For load instructions, the address is sent to memory. For store instructions, the address and data are sent to memory.[1]
			Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.[1]
			For multiply unit, nonmultiply instructions, results are written to a register file.[2]
			For multiply, 2-cycle DP, and DP compare instructions, results are written to a register file.[1]
			For DP compare and ADDDP/SUBDP instructions, the upper 32 bits of the source are read.[1]
			For MPYDP instruction, the lower 32 bits of *src1* and the upper 32 bits of *src2* are read.[1]
			For MPYI and MPYID instructions, the sources are read.[1]
			For MPYSPDP instruction, the *src1* and the upper 32 bits of *src2* are read.[1]
	Execute 3	E3	Data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.[1]
			For MPYDP instruction, the upper 32 bits of *src1* and the lower 32 bits of *src2* are read.[1]
			For MPYI and MPYID instructions, the sources are read.[1]
	Execute 4	E4	For load instructions, data is brought to the CPU boundary.[1]
			For multiply extensions, results are written to a register file.[2]
			For MPYI and MPYID instructions, the sources are read.[1]
			For MPYDP instruction, the upper 32 bits of the sources are read.[1]
			For MPYI and MPYID instructions, the sources are read.[1]
			For 4-cycle instructions, results are written to a register file.[1]
			For INTDP and MPYSP2DP instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 5	E5	For load instructions, data is written into a register.[1]
			For INTDP and MPYSP2DP instructions, the upper 32 bits of the result are written to a register file.[1]
	Execute 6	E6	For ADDDP/SUBDP and MPYSPDP instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 7	E7	For ADDDP/SUBDP and MPYSPDP instructions, the upper 32 bits of the result are written to a register file.[1]
	Execute 8	E8	Nothing is read or written.
	Execute 9	E9	For MPYI instruction, the result is written to a register file.[1]
			For MPYDP and MPYID instructions, the lower 32 bits of the result are written to a register file.[1]
	Execute 10	E10	For MPYDP and MPYID instructions, the upper 32 bits of the result are written to a register file.[1]

1. This assumes that the conditions for the instructions are 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. Multiply unit, nonmultiply instructions are **AVG2, AVG4, BITC4, BITR, DEAL, ROT, SHFL, SSHVL,** and **SSHVR.**

Figure 5-7 Pipeline Phases Block Diagram



Example 5-1 Execute Packet in Figure 5-7

```
SADD.L1 A2,A7,A2; E1 Phase
||  SADD.L2 B2,B7,B2
||  SMPYH.M2XB3,A3,B2
||  SMPY.M1XB3,A3,A2
||  B    .S1 LOOP1
||  MVK .S2 117,B1
    LDW .D2 *B4++,B3; DC Phase
||  LDW .D1 *A4++,A3
||  MV  .L2XA1,B0
||  SMPYH.M1A2,A2,A0
||  SMPYH.M2B2,B2,B10
||  SHR .S1 A2,16,A5
||  SHR .S2 B2,16,B5
LOOP1:
    STH .D1 A5,*A8++[2]; DP, PW, and PG Phases
||  STH .D2 B5,*B8++[2]
||  SADD.L1 A2,A7.A2
||  SADD.L2 B2,B7,B2
||  SMPYH.M2XB3,A3,B2
||  SMPY.M1XB3,A3,A2
||  [B1] B.S1LOOP1
||  [B1] SUB.S2B1,1,B1
    LDW .D2 *B4++,B3: PR and PS Phases
||  LDW .D1 *A4++,A3
||  SADD.L1 A0,A1,A1
||  SADD.L2 B10,B0,B0
||  SMPYH.M1A2,A2,A0
||  SMPYH.M2B2,B2,B10
||  SHR .S1 A2,16,A5
||  SHR .S2 B2,16,B5
```

End of Example 5-1

C.1 Instructions Executing in the .D Functional Unit

Table C-1 lists the instructions that execute in the .D functional unit.

Table C-1 Instructions Executing in the .D Functional Unit

Instruction	Instruction
ADD	OR
ADDAB	STB
ADDAD	STB[1] (15-bit offset)
ADDAH	STDW
ADDAW	STH
ADD2	STH[1] (15-bit offset)
AND	STNDW
ANDN	STNW
LDB and LDB(U)	STW
LDB and LDB(U)[1] (15-bit offset)	STW[1] (15-bit offset)
LDDW	SUB
LDH and LDH(U)	SUBAB
LDH and LDH(U)[1]	SUBAH
LDNDW	SUBAW
LDNW	SUB2
LDW	XOR
LDW[1] (15-bit offset)	ZERO
MV	
MVK	

1. D2 only

D.1 Instructions Executing in the .L Functional Unit

Table D-1 lists the instructions that execute in the .L functional unit.

Table D-1 Instructions Executing in the .L Functional Unit

Instruction	Instruction	Instruction	Instruction
ABS	DPACK2	NORM	SPTRUNC
ABS2	DPACKX2	NOT	SSUB
ADD	DPINT	OR	SSUB2
ADDDP	DPSP	PACK2	SUB
ADDSP	DPTRUNC	PACKH2	SUBABS4
ADDSUB	INTDP	PACKH4	SUBC
ADDSUB2	INTDPU	PACKHL2	SUBDP
ADDU	INTSP	PACKLH2	SUBSP
ADD2	INTSPU	PACKL4	SUBU
ADD4	LMBD	SADD	SUB2
AND	MAX2	SADDSUB	SUB4
ANDN	MAXU4	SADDSUB2	SWAP2
CMPEQ	MIN2	SAT	SWAP4
CMPGT	MINU4	SHFL3	UNPKHU4
CMPGTU	MV	SHLMB	UNPKLU4
CMPLT	MVK	SHRMB	XOR
CMPLTU	NEG	SPINT	ZERO

H.1 Instructions Executing With No Unit Specified

Table H-1 on page H-2 lists the instructions that execute with no unit specified.

Table H-1 Instructions Executing With No Unit Specified

Instruction
DINT
IDLE
NOP
RINT
SPKERNEL
SPKERNELR
SPLOOP
SPLOOPD
SPLOOPW
SPMASK
SPMASKR
SWE
SWENR

E.1 Instructions Executing in the .M Functional Unit

Figure E-1 lists the instructions that execute in the .M functional unit.

Table E-1 Instructions Executing in the .M Functional Unit

Instruction	Instruction	Instruction	Instruction
AVG2	DOTPUS4	MPYIL	MPY32 (32-bit result)
AVGU4	DOTPU4	MPYILR	MPY32 (64-bit result)
BITC4	GMPY	MPYLH	MPY32SU
BITR	GMPY4	MPYLHU	MPY32U
CMPY	MPY	MPYLI	MPY32US
CMPYR	MPYDP	MPYLIR	MVD
CMPYR1	MPYH	MPYLSHU	ROTL
DDOTP4	MPYHI	MPYLUHS	SHFL
DDOTPH2	MPYHIR	MPYSP	SMPY
DDOTPH2R	MPYHL	MPYSPDP	SMPYH
DDOTPL2	MPYHLU	MPYSP2DP	SMPYHL
DDOTPL2R	MPYHSLU	MPYSU	SMPYLH
DEAL	MPYHSU	MPYSU4	Multiply Signed by Signed, 16 LSB × 16 LSB and 16 MSB × 16 MSB With Left Shift and Saturation
DOTP2	MPYHU	MPYU	SMPY32
DOTPN2	MPYHULS	MPYU4	SSHVL
DOTPNRSU2	MPYHUS	MPYUS	SSHVR
DOTPNRUS2	MPYI	MPYUS4	XORMPY
DOTPRSU2	MPYID	MPY2	XPND2
DOTPRUS2	MPYIH	MPY2IR	XPND4
DOTPSU4	MPYIHR		

F.1 Instructions Executing in the .S Functional Unit

Table F-1 lists the instructions that execute in the .S functional unit.

Table F-1 Instructions Executing in the .S Functional Unit

Instruction	Instruction	Instruction	Instruction
ABSDP	CMPEQ2	MVKH/MVKLH	SET
ABSSP	CMPEQ4	MVKL	SHL
ADD	CMPEQDP	MVKH/MVKLH	SHLMB
ADDDP	CMPEQSP	NEG	SHR
ADDK	CMPGT2	NOT	SHR2
ADDKPC[1]	CMPGTDP	OR	SHRMB
ADDSP	CMPGTSP	PACK2	SHRU
ADD2	CMPGTU4	PACKH2	SHRU2
AND	CMPLT2	PACKHL2	SPACK2
ANDN	CMPLTDP	PACKLH2	SPACKU4
B displacement	CMPLTSP	RCPDP	SPDP
B register[1]	CMPLTU4	RCPSP	SSHL
B IRP[1]	DMPYU4	RPACK2	SUB
B NRP[1]	EXT	RSQRDP	SUBDP
BDEC	EXTU	RSQRSP	SUBSP
BNOP displacement	MAX2	SADD	SUB2
BNOP register	MIN2	SADD2	SWAP2
BPOS	MV	SADDSU2	UNPKHU4
CALLP	MVC[1]	SADDUS2	UNPKLU4
CLR	MVK	SADDU4	XOR
			ZERO

1. S2 only

4.6 ADD

Add Two Signed Integers Without Saturation

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADDU, ADD2, SADD

Examples **Example 1**

```
ADD .L2X    A1,B1,B2
```

	Before instruction			1 cycle after instruction	
A1	0000325Ah	12,890	A1	0000325Ah	
B1	FFFFFF12h	-238	B1	FFFFFF12h	
B2	xxxxxxxxh		B2	0000316Ch	12,652

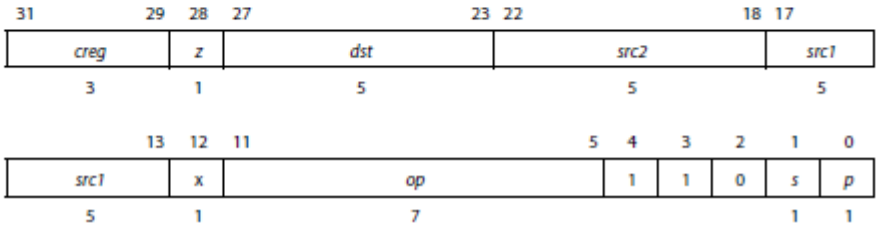
4.11 ADDDP

Add Two Double-Precision Floating-Point Values

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

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

Opcode



Instruction Type ADDDP/SUBDP

Delay Slots 6

Functional Unit Latency 2

See Also ADD, ADDSP, ADDU, SUBDP

Example

```
ADDDP .L1X  B1:B0,A3:A2,A5:A4
```

	Before instruction			7 cycles after instruction		
B1:B0	4021 3333h	3333 3333h	B1:B0	4021 3333h	4021 3333h	8.6
A3:A2	C004 0000h	0000 0000h	A3:A2	C004 0000h	0000 0000h	-2.5
A5:A4	xxxx xxxxh	xxxx xxxxh	A5:A4	4018 6666h	6666 6666h	6.1

ADDSP

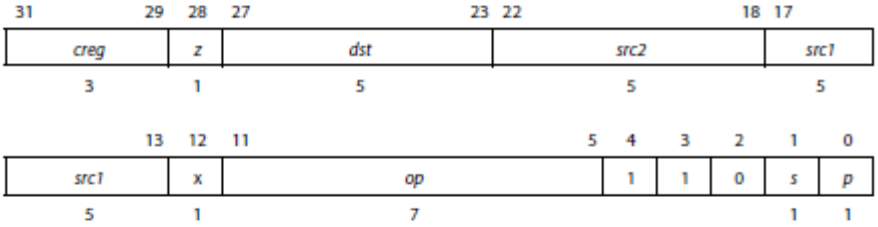
Add Two Single-Precision Floating-Point Values

Syntax

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

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

Opcode



Opcode map field used...	For operand type...	Unit	Opfield
src1 *src2* *dst*	sp xsp sp	.L1, .L2	0010000
src1 *src2* *dst*	sp xsp sp	.S1, .S2	1110000

Pipeline

Pipeline Stage	E1	E2	E3	E4
Read	*src1, src2*			
Written				*dst*
Unit in use	.L or .S			

Instruction Type 4-cycle

Delay Slots 3

B

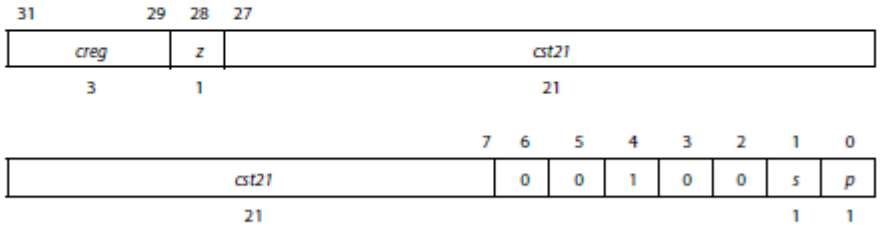
Branch Using a Displacement

Syntax

B (.unit) label

unit = .S1 or .S2

Opcode



Opcode map field used...	For operand type...	Unit
cst21	*scst21*	.S1, .S2

Pipeline

		Target Instruction					
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read							
Written							
Branch taken							✓
Unit in use	.S						

Instruction Type Branch

Delay Slots 5

BDEC

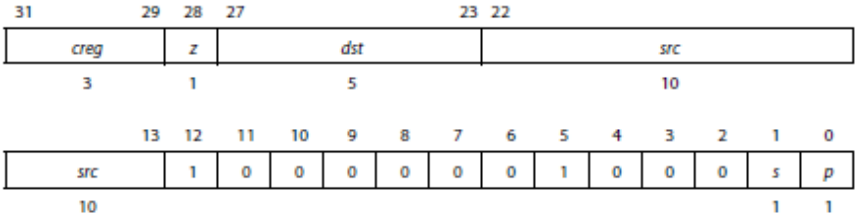
Branch and Decrement

Syntax

BDEC (.unit) *src, dst*

unit = .S1 or .S2

Opcode



Opcode map field used...	For operand type...	Unit
src	scst10	.S1, .S2
dst	int	

Pipeline

		Target Instruction					
Pipeline Stage	**E1**	**PS**	**PW**	**PR**	**DP**	**DC**	**E1**
Read	*dst*						
Written	*dst*, PC						
Branch taken							✓
Unit in use	.S						

Instruction Type Branch

Delay Slots 5

CLR

Clear a Bit Field

Syntax

CLR (.unit) *src2, csta, cstb, dst*

or

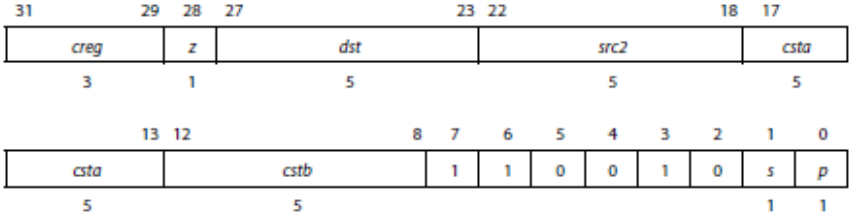
CLR (.unit) *src2, src1, dst*

unit = .S1 or .S2

Instruction Format

Unit	Opcode Format	Figure
.S	Sc5	Figure F-22

Opcode Constant form



Pipeline

Pipeline Stage	E1
Read	*src1, src2*
Written	*dst*
Unit in use	.S

Instruction Type Single-cycle

Delay Slots 0

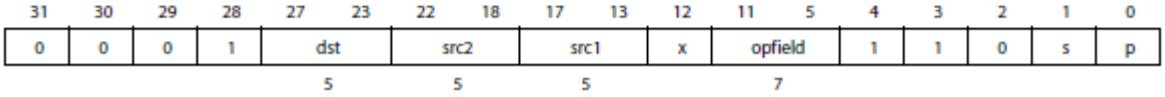
4.67 DADDSP

2-Way SIMD Single Precision Floating Point Addition

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

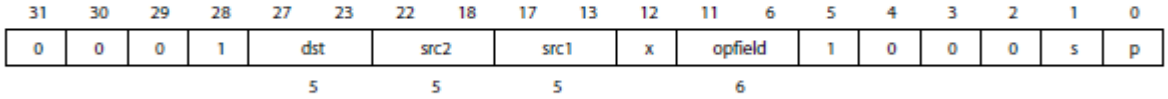
unit = .L1, .L2, .S1, or .S2

Opcode Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode



Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0111100

Opcode Opcode for .S Unit, 1/2 src, unconditional



Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	101100

Instruction Type 3-cycle

Delay Slots 2

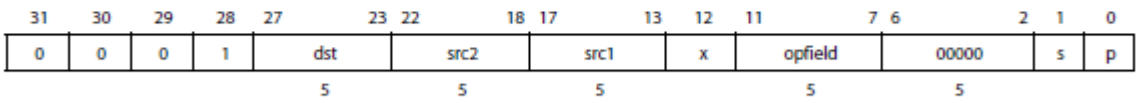
4.101 DMPYSP

2-Way SIMD Multiply, Packed Single Precision Floating Point

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

unit = .M1 or .M2

Opcode Opcode for .M Unit, 32-bit, unconditional



Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	11100

Instruction Type 4-cycle

Delay Slots 3

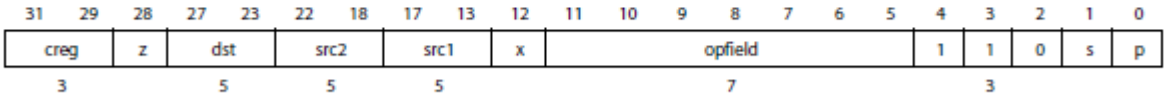
4.152 FADDSP

Fast Single-Precision Floating Point Add

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

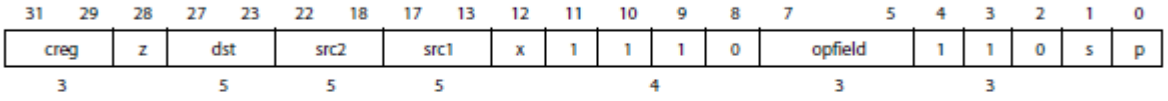
unit = .L1, .L2, .S1, or .S2

Opcode Opcode for .L Unit, 1/2 src



Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	0111100

Opcode Opcode for .S Unit, 2 src



Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.S1 or .S2	100

Description *src2* is added to *src1*. The result is placed in *dst*. This instruction is the fast version of ADDSP, with smaller Delay Slots.

Instruction Type 3-cycle

Delay Slots 2

4.167 LDDW

Load Doubleword From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

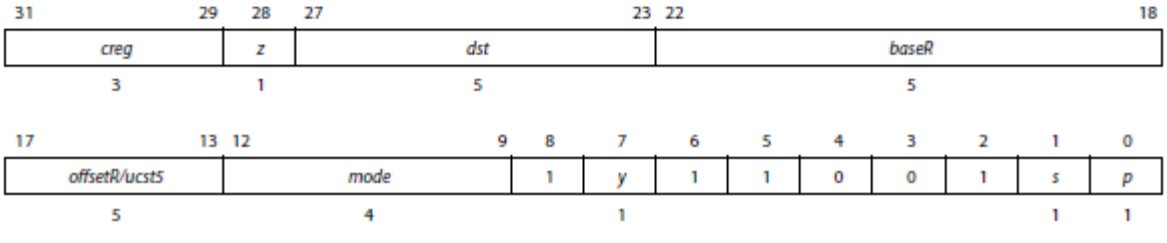
Syntax

```
Register Offset                    Unsigned Constant Offset
LDDW (.unit) *+baseR[offsetR], dst  LDDW (.unit) *+baseR[ucst5], dst
unit = .D1 or .D2
```

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-9
	DindDW	Figure C-11
	DincDW	Figure C-13
	DdecDW	Figure C-15
	Dpp	Figure C-21

Opcode



Instruction Type Load

Delay Slots 4

4.170 LDNDW

Load Nonaligned Doubleword From Memory With Constant or Register Offset

Syntax

Register Offset

```
LDNDW (.unit) *+baseR[offsetR], dst
unit = .D1 or .D2
```

Unsigned Constant Offset

```
LDNDW (.unit) *+baseR[ucst5], dst
```

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-9
	DindDW	Figure C-11
	DincDW	Figure C-13
	DdecDW	Figure C-16

Opcode

31 29	28	27 24	23	22 18
creg	z	dst	sc	baseR
3	1	4	1	5

17 13	12 9	8	7	6	5	4	3	2	1	0
offsetR/ucst5	mode	1	y	0	1	0	0	1	s	p
5	4		1						1	1

Opcode map field used...	For operand type...	Unit
baseR *offsetR* *dst*	uint uint ullong	.D1, .D2
baseR *offsetR* *dst*	uint ucst5 ullong	.D1, .D2

Instruction Type Load

Delay Slots 4 for loaded value

4.172 LDW

Load Word From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax

Register Offset

```
LDW (.unit) *+baseR[offsetR], dst
unit = .D1 or .D2
```

Unsigned Constant Offset

```
LDW (.unit) *+baseR[ucst5], dst
```

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-8
	Dind	Figure C-10
	Dinc	Figure C-12
	Ddec	Figure C-14

Opcode

31 29	28	27 23	22 18
creg	z	dst	baseR
3	1	5	5

17 13	12 9	8	7	6	5	4	3	2	1	0
offsetR/ucst5	mode	0	y	1	1	0	0	1	s	p
5	4		1						1	1

Instruction Type Load

Delay Slots 4 for loaded value

0 for address modification from pre/post increment/decrement

For more information on delay slots for a load, see Chapter 5 "Pipeline" on page 5-1.

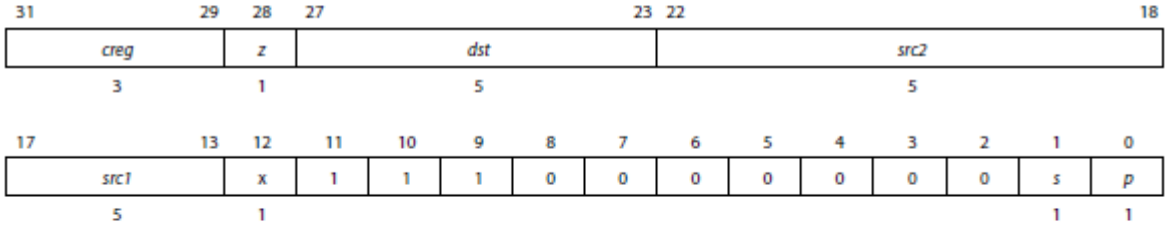
4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

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

unit = .M1 or .M2

Opcode



Opcode map field used...	For operand type...	Unit
src1	sp	.M1, .M2
src2	xsp	
dst	sp	

Instruction Type Four-cycle

Delay Slots 3

4.222 MV

Move From Register to Register

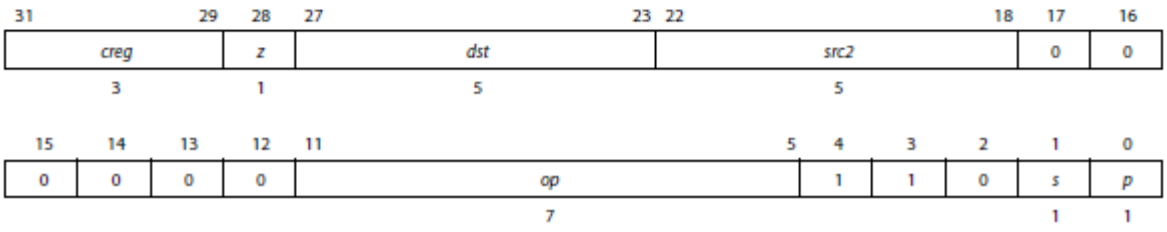
Syntax **MV** (.unit) *src2, dst*

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

Compact Instruction Format

Unit	Opcode Format	Figure
.L, .S, .D	LSDmvto	Figure G-1
	LSDmvtfr	Figure G-2

Opcode .L unit



Description The **MV** pseudo-operation moves a value from one register to another. The assembler will either use the **ADD** (.unit) 0, *src2, dst* or the **OR** (.unit) 0, *src2, dst* operation to perform this task.

Execution

```
if (cond)   0 + src2 → dst
else nop
```

Instruction Type Single-cycle

Delay Slots 0

4.226 MVKH/MVKLH

Move 16-Bit Constant Into Upper Bits of Register

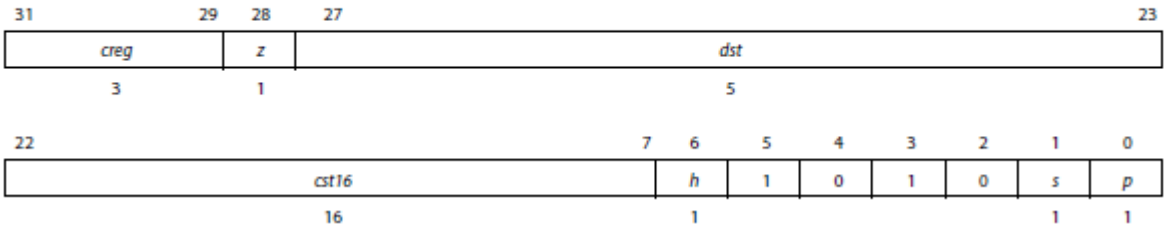
Syntax **MVKH** (.unit) *cst, dst*

or

MVKLH (.unit) *cst, dst*

unit = .S1 or .S2

Opcode



Instruction Type Single-cycle

Delay Slots 0

Examples **Example 1**

```
MVKH .S1   0A329123h,A1
```

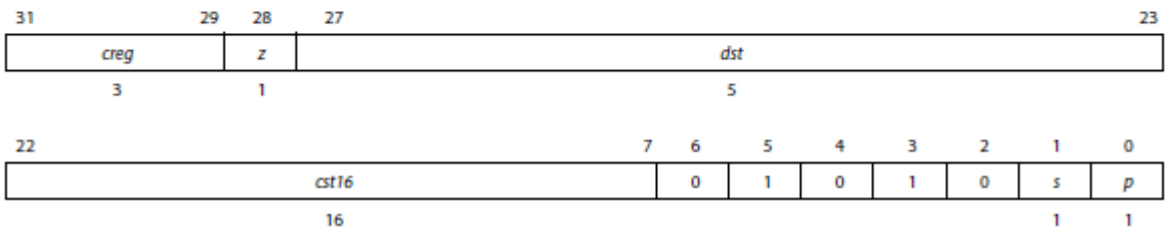
4.227 MVKL

Move Signed Constant Into Register and Sign Extend

Syntax **MVKL** (.unit) *cst, dst*

unit = .S1 or .S2

Opcode



Instruction Type Single cycle

Delay Slots 0

See Also MVK, MVKH/MVKLH

Examples **Example 1**

```
MVKL .S1   5678h,A8
```

4.229 NOP

No Operation

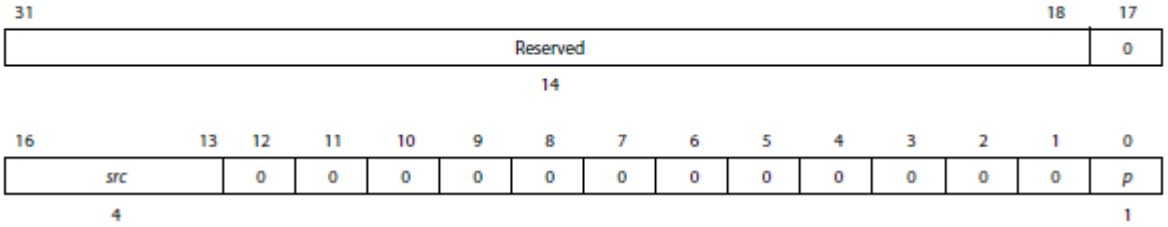
Syntax **NOP** [*count*]

unit = none

Compact Instruction Format

Unit	Opcode Format	Figure
none	Unop	Figure H-7

Opcode



Instruction Type **NOP**

Delay Slots 0

Examples **Example 1**

```
NOP MVK .S1   125h,A1
```

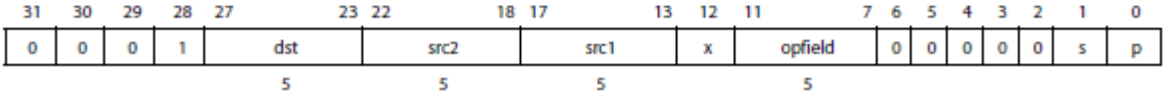
4.240 QMPYSP

4-Way SIMD Floating Point Multiply, Packed Single-Precision Floating Point

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

unit = .M1 or .M2

Opcode Opcode for .M Unit, 32-bit, unconditional



Instruction Type 4-cycle

Delay Slots 3

Example

```
FA3 == 0x80000000
A2 == 0x80000000
A1 == 0x7FFFFFFF
A0 == 0xFFFFFFFF

A11 == 0xFFFFFFF
A10 == 0x8000000
A9 == 0x7FFFFFF
A8 == 0xFFFFFFF
QMPY32 .M .....
A15 == 0x8000000
A14 == 0x0000000
A13 == 0x0000001
A12 == 0x0000001
```

4.293 STDW

Store Doubleword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax

Register Offset

STDW (.unit) *src*, *+*baseR*[*offsetR*]

unit = .D1 or .D2

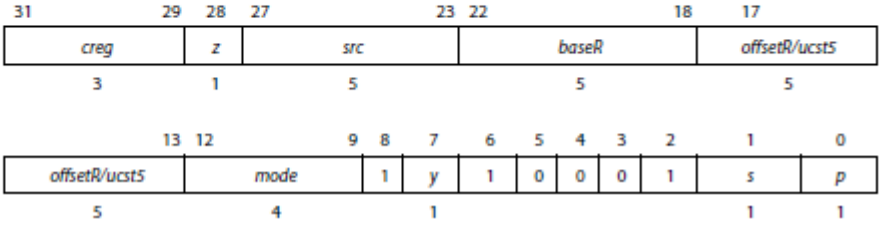
Unsigned Constant Offset

STDW (.unit) *src*, *+*baseR*[*ucst5*]

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	Figure C-9
	DindDW	Figure C-11
	DincDW	Figure C-13
	DdecDW	Figure C-15
	Dpp	Figure C-21

Opcode



Instruction Type Store

Delay Slots 0

4.294 STH

Store Halfword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax

Register Offset

STH (.unit) *src*, *+*baseR*[*offsetR*]

unit = .D1 or .D2

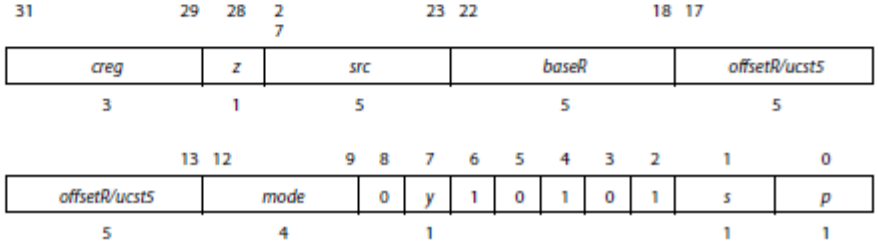
Unsigned Constant Offset

STH (.unit) *src*, *+*baseR*[*ucst5*]

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-8
	Dind	Figure C-10
	Dinc	Figure C-12
	Ddec	Figure C-14

Opcode



Instruction Type Store

Delay Slots 0

4.298 STW

Store Word to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax

Register Offset

STW (.unit) *src*, *+*baseR*[*offsetR*]

unit = .D1 or .D2

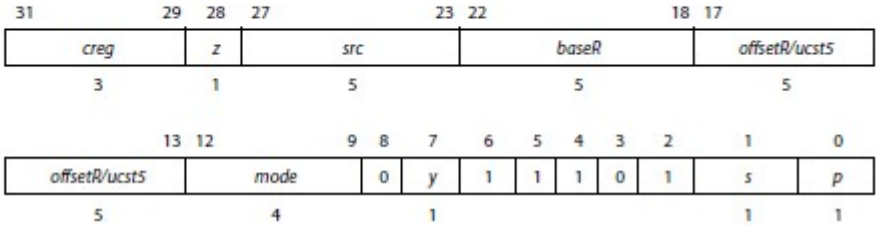
Unsigned Constant Offset

STW (.unit) *src*, *+*baseR*[*ucst5*]

Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	Figure C-8
	Dind	Figure C-10
	Dinc	Figure C-12
	Ddec	Figure C-14

Opcode



Instruction Type Store

Delay Slots 0

4.300 SUB

Subtract Two Signed Integers Without Saturation

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

Instruction Type Single-cycle

Delay Slots 0

See Also ADD, SUBC, SUBU, SSUB, SUB2

Example

```
SUB .L1     A1,A2,A3
```

	Before Instruction		1 cycle after Instruction	
A1	0000325Ah	12,890	0000325Ah	
A2	FFFFFF12h	-238	FFFFFF12h	
A3	xxxxxxxxh		00003348h	13,128

4.324 ZERO

Zero a Register

Syntax **ZERO** (.unit) *dst*

Instruction Type Single-cycle

Delay Slots 0

See Also MVK, SUB

Examples **Example 1**

```
ZERO .D1 A1
```

	Before instruction		1 cycle after instruction
A1	B174 6CA1h	A1	0000 0000h

Prise de notes

Prise de notes

Prise de notes

Prise de notes

Prise de notes

Prise de notes

Prise de notes

Choisissez un travail que vous aimez
et vous n'aurez pas à travailler
un seul jour de votre vie.

Confucius