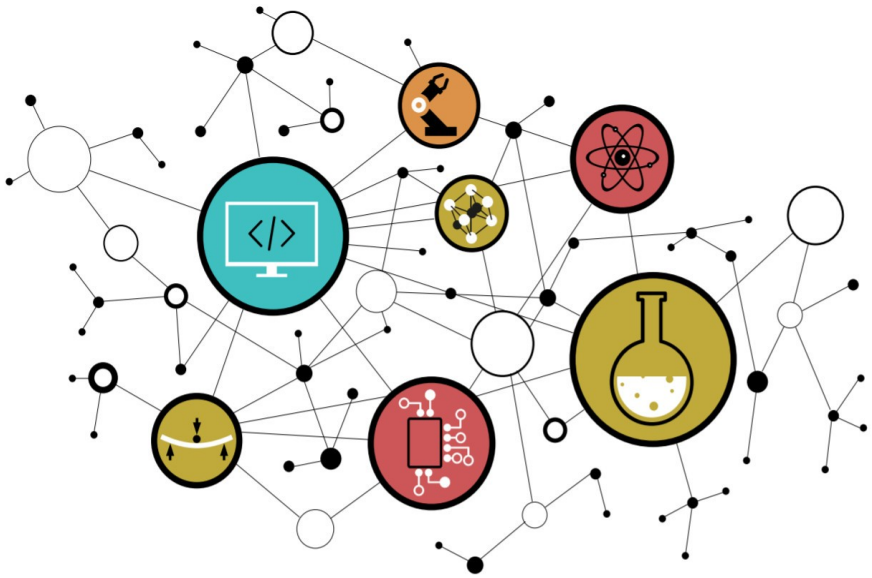
# TRAVAUX PRATIQUES

## ANNEXES

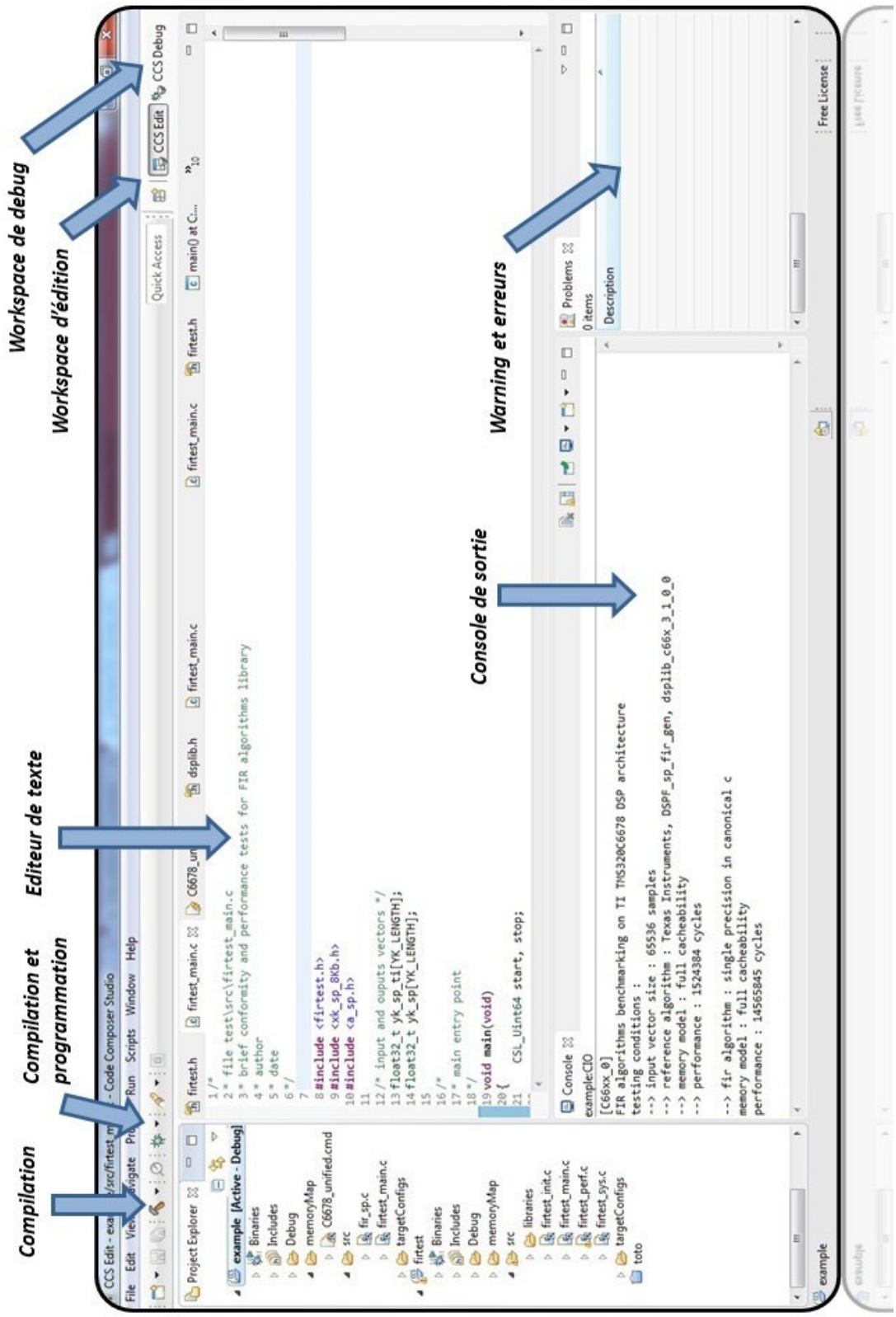


Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

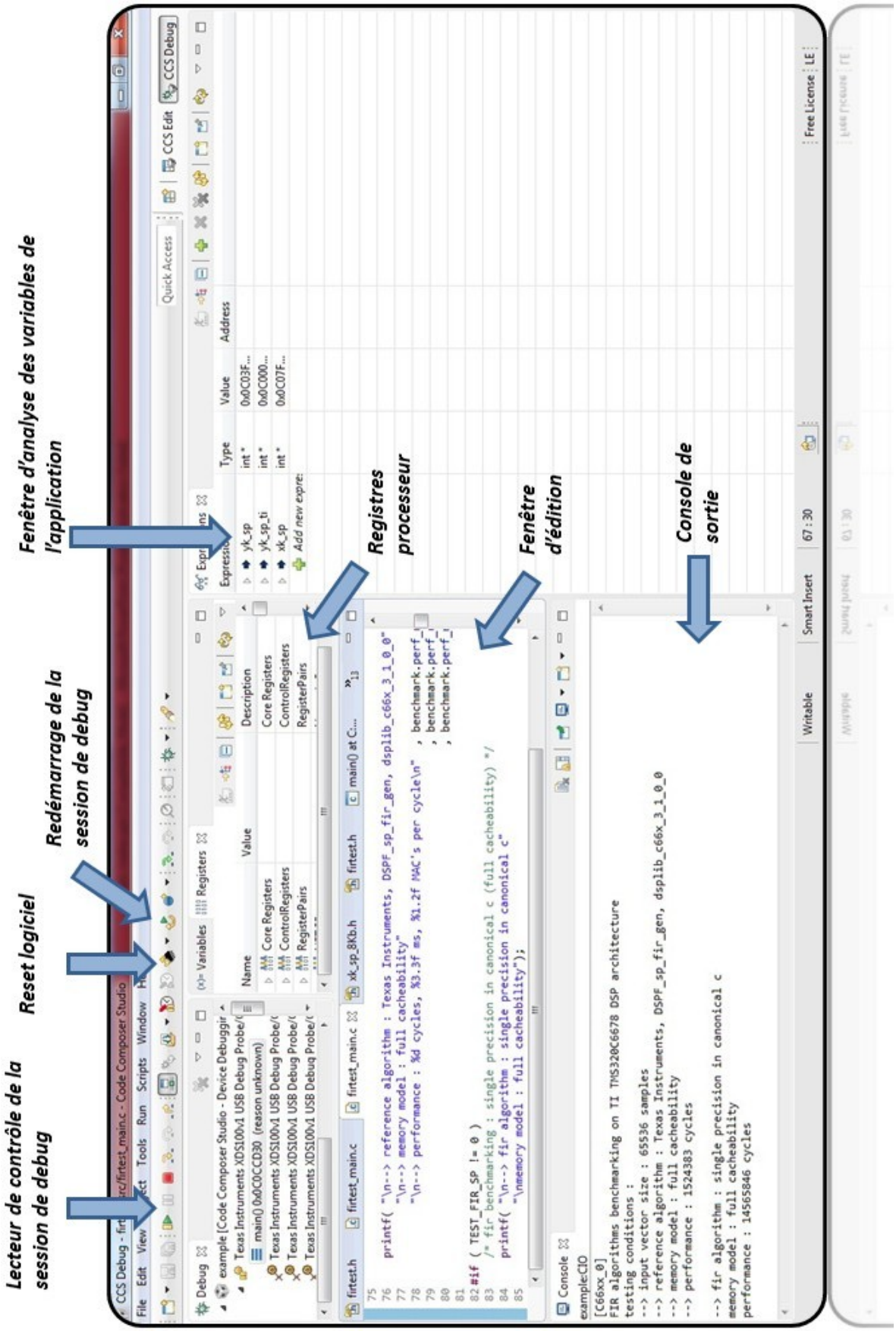
# SOMMAIRE

# 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++.



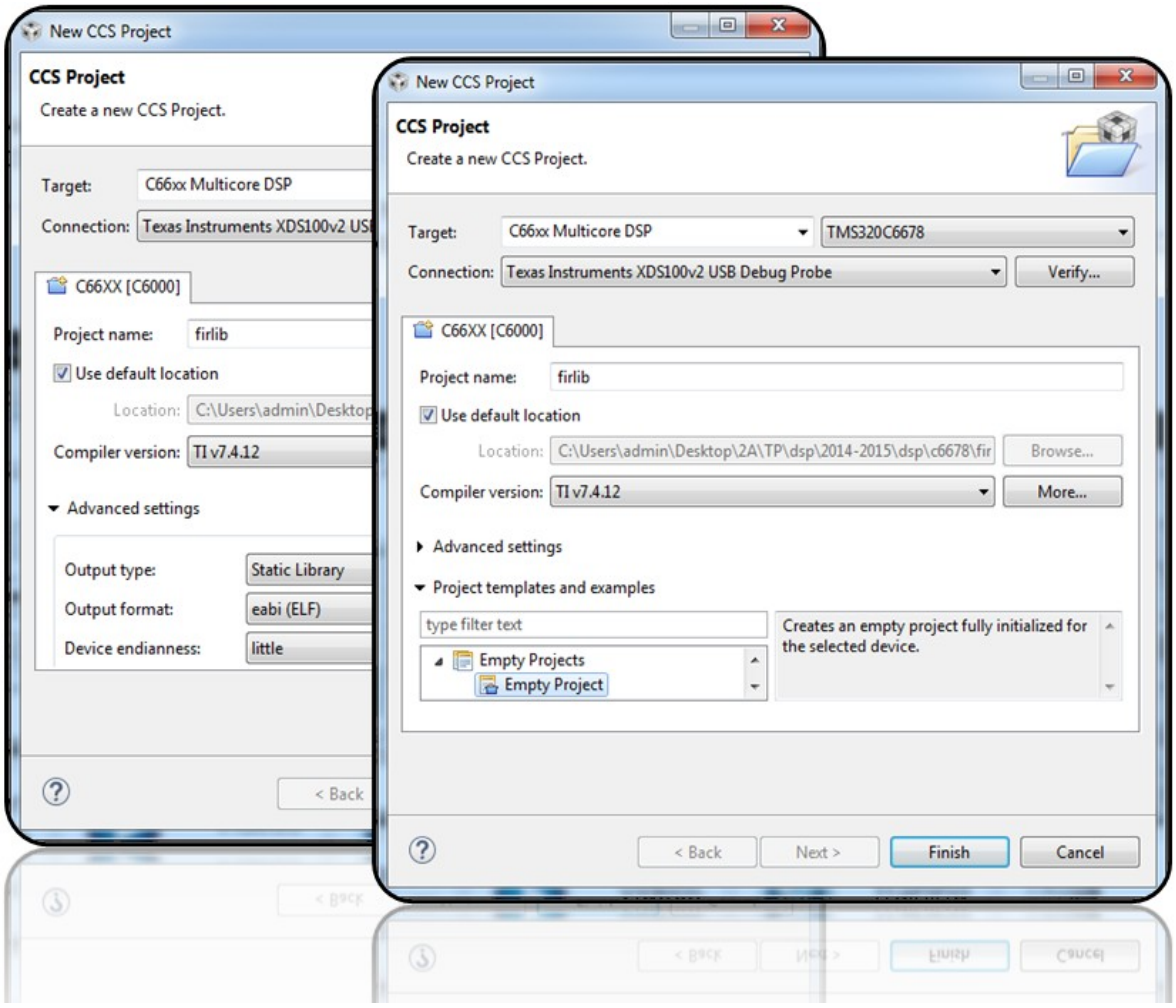
Présentation du **workspace de debug**, d'analyse et de communication avec la sonde de programmation XDS100 (carré rouge pour le fermer ce workspace). **Ne pas éditer dans cet espace de travail (bug non corrigé sur cette version de l'IDE)** !



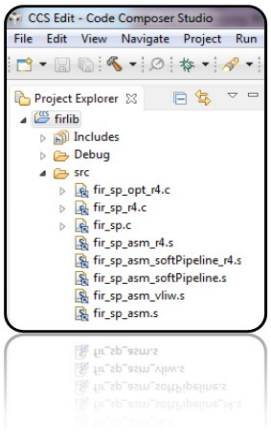
## 2. CRÉATION D'UNE BIBLIOTHÈQUE STATIQUE

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.

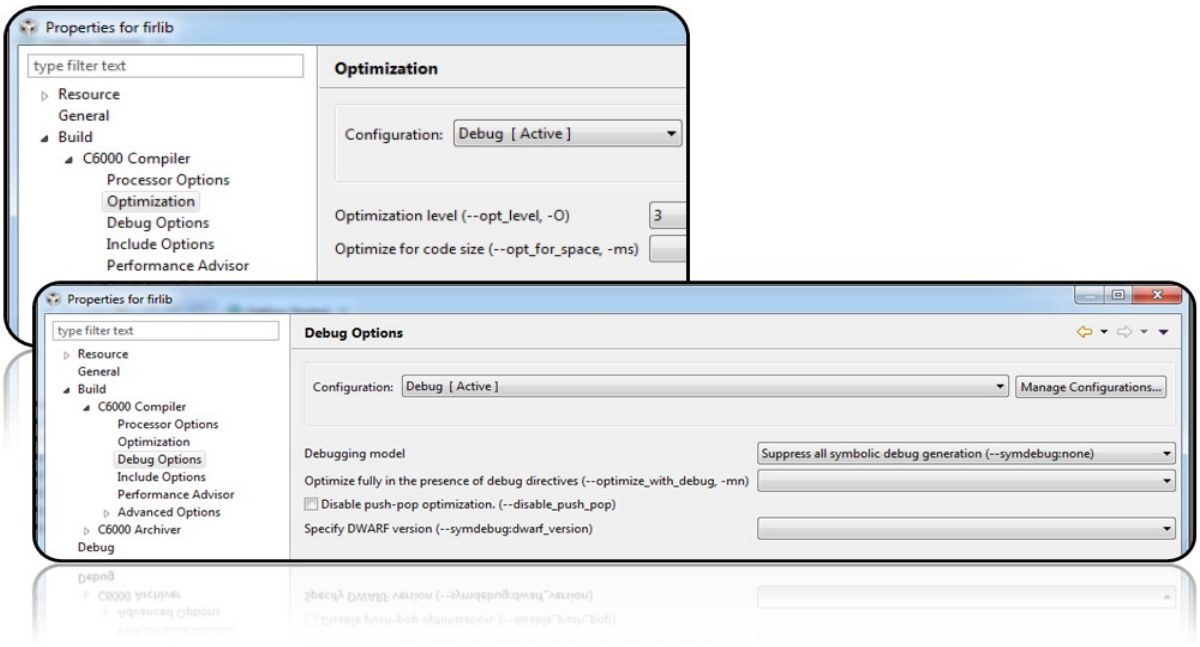

- Créer un nouveau projet dans un nouveau répertoire propre à la génération de la bibliothèque statique. Dans le cadre de cette trame de travaux pratiques, placer ce projet par exemple dans **/disco/c6678/firlib/pjct/**. Sélectionner néanmoins dans les options avancées le format du fichier de sortie (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

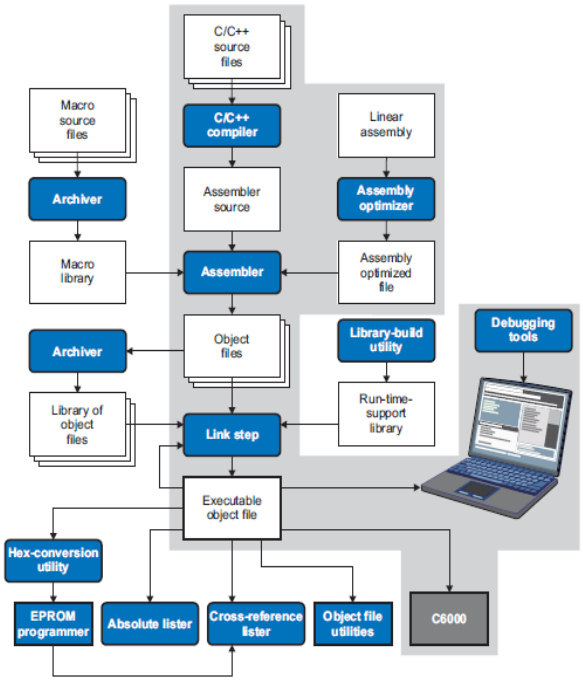



**Table 6-2. TMS320C6000 C/C++ EABI Data Types**

| | | | 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 |

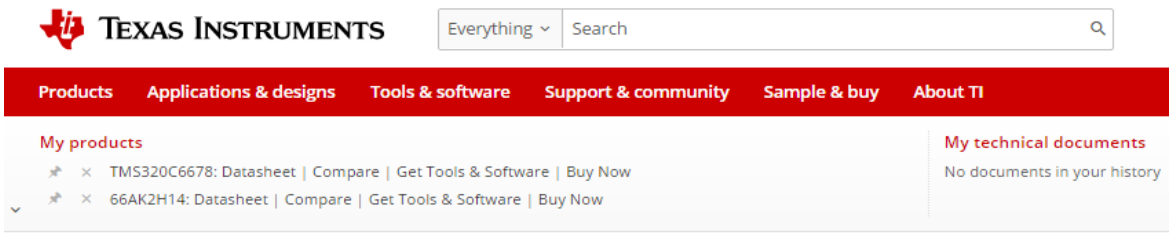
(1) For EABI, structs of size 64 or less are passed by value in registers instead of by reference using a pointer in A3.

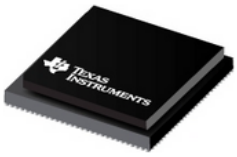
**Figure 7-10. Register Argument Conventions**

```
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

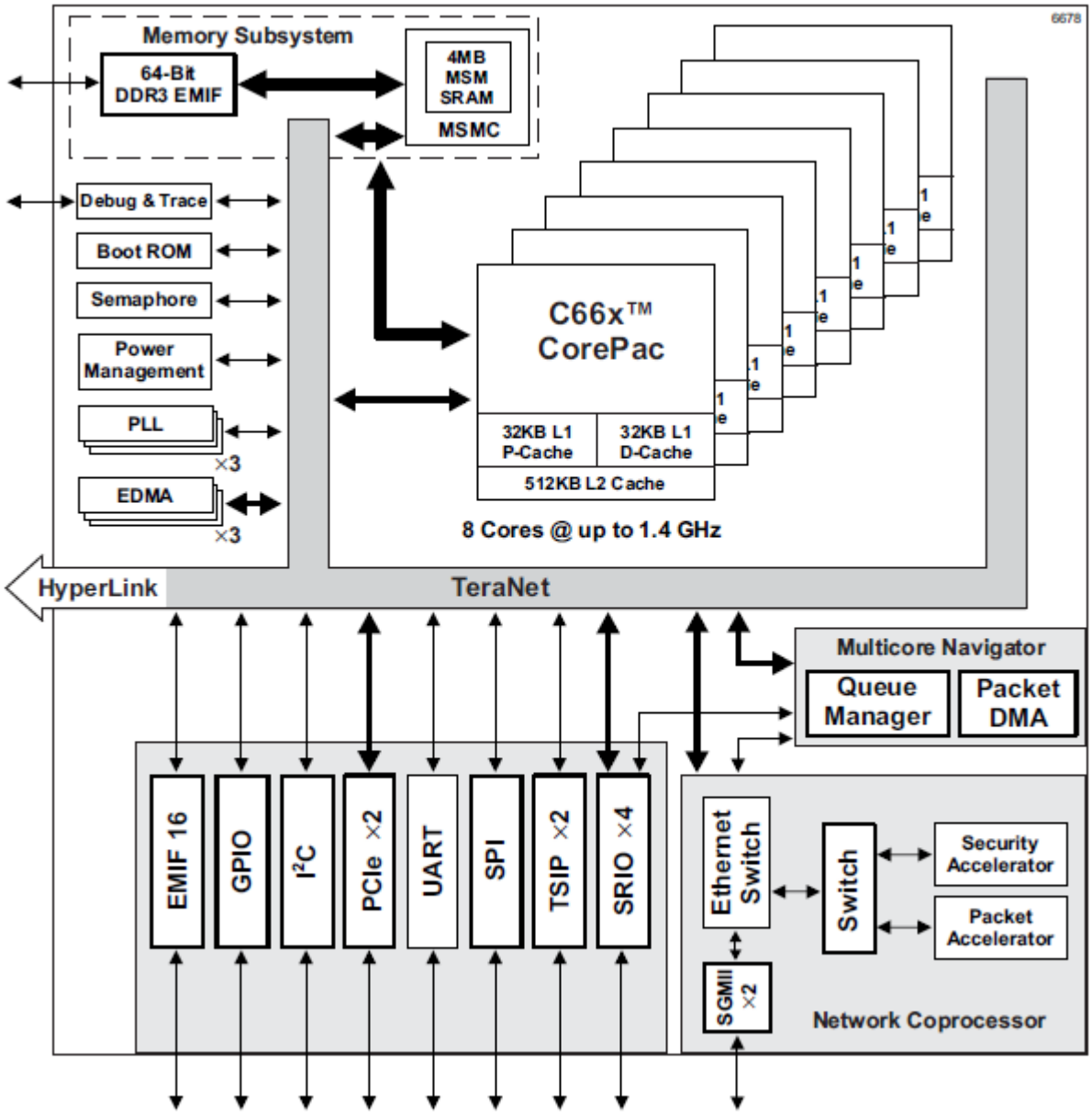
**SILICON REVISION**
Blank = Initial Silicon 1.0
A = Silicon Revision 2.0

**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

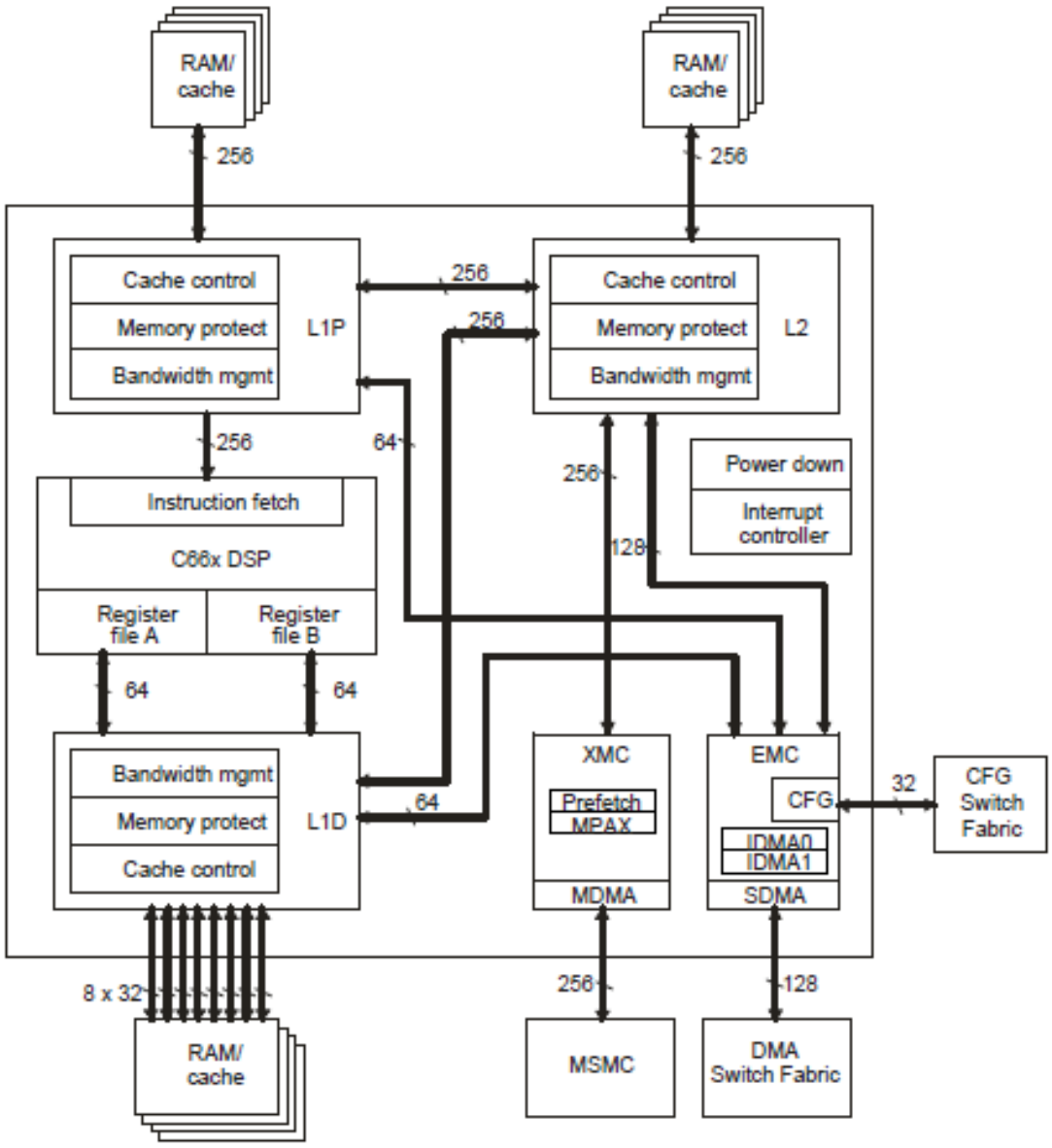


## 1.1 Introduction

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**



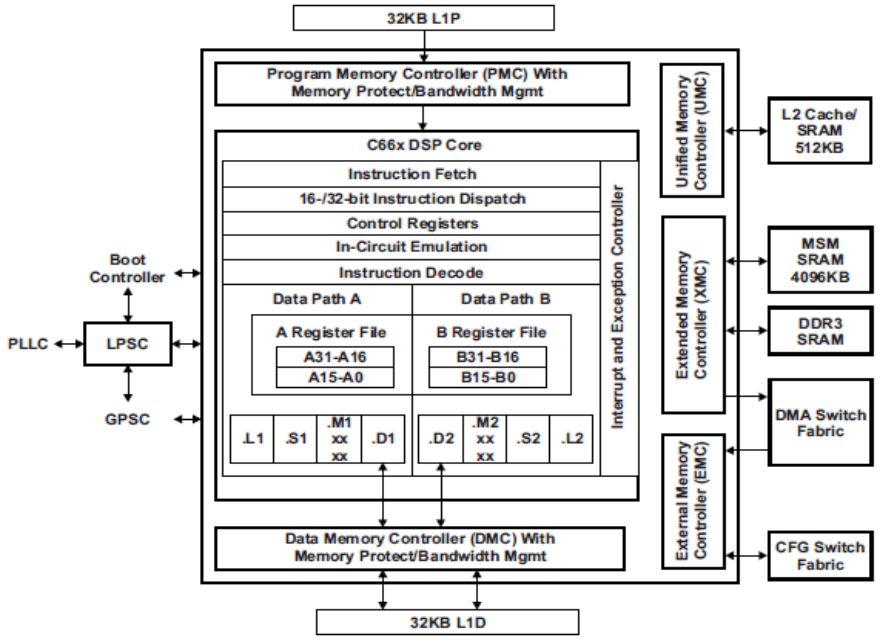
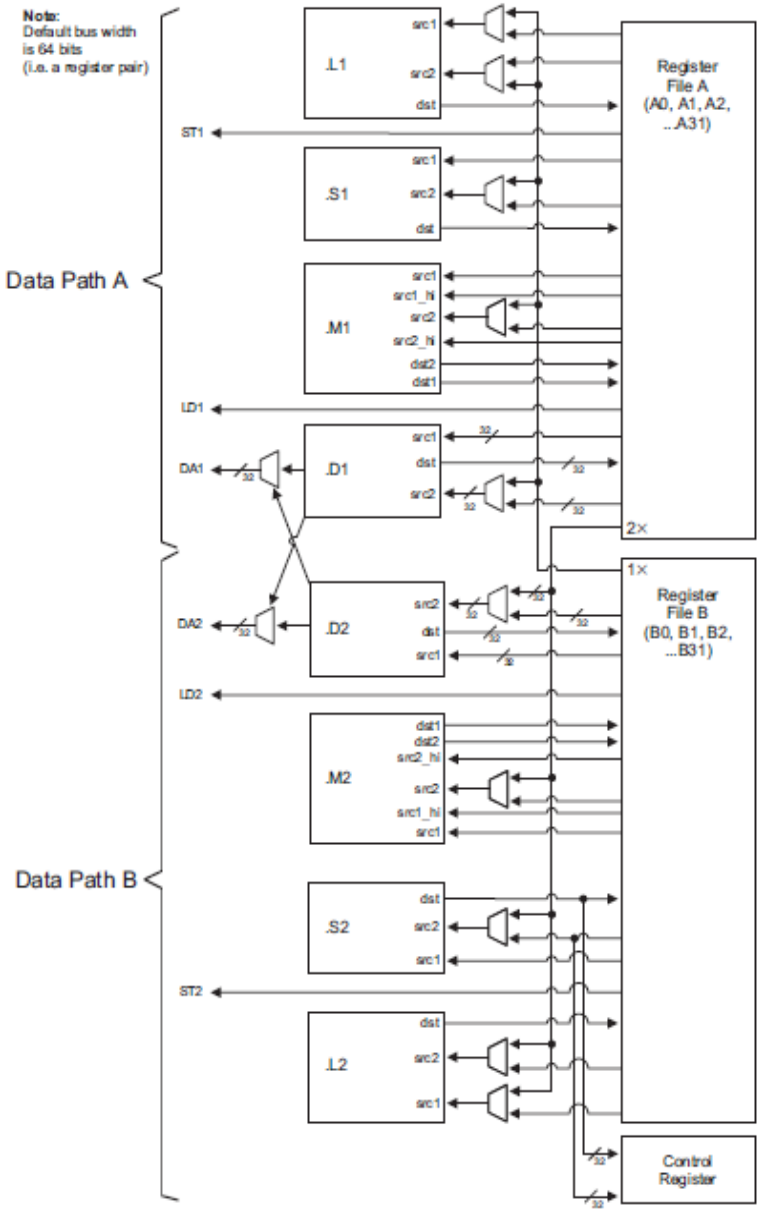



**Figure 2-1 DSP Core Data Paths**



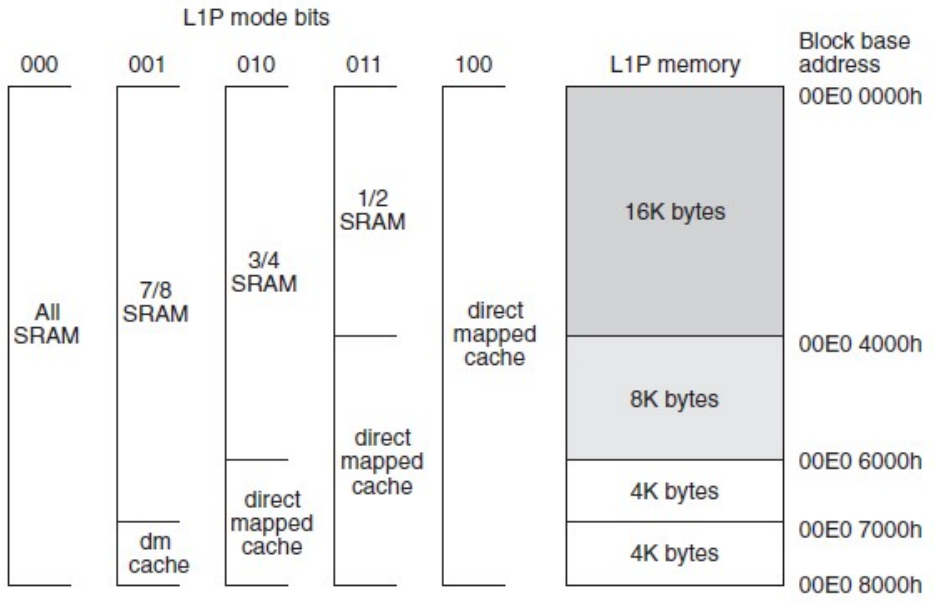
### 5.1.1 L1P Memory

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**



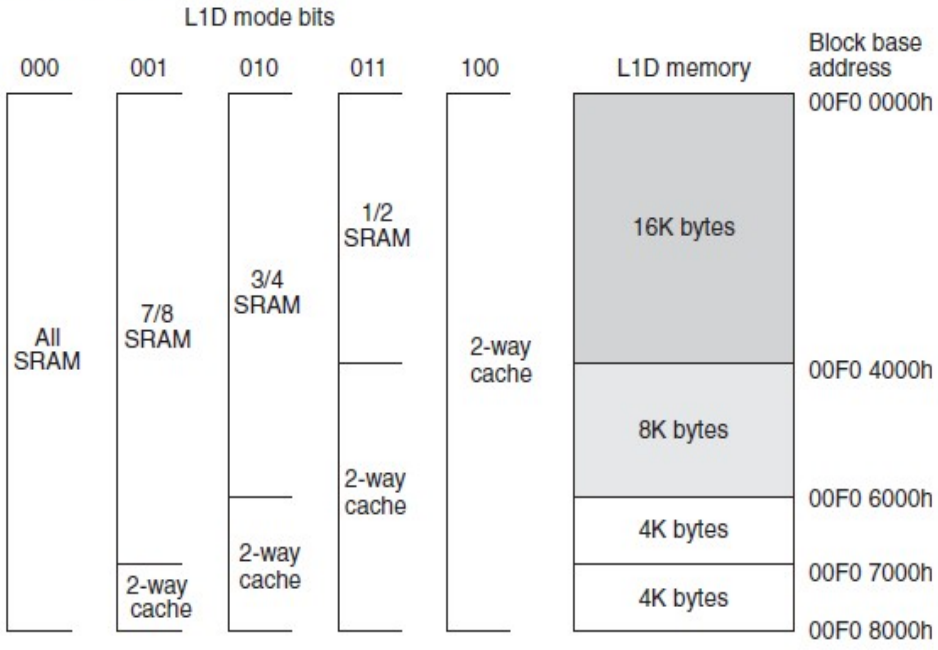

### 5.1.2 L1D Memory

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**



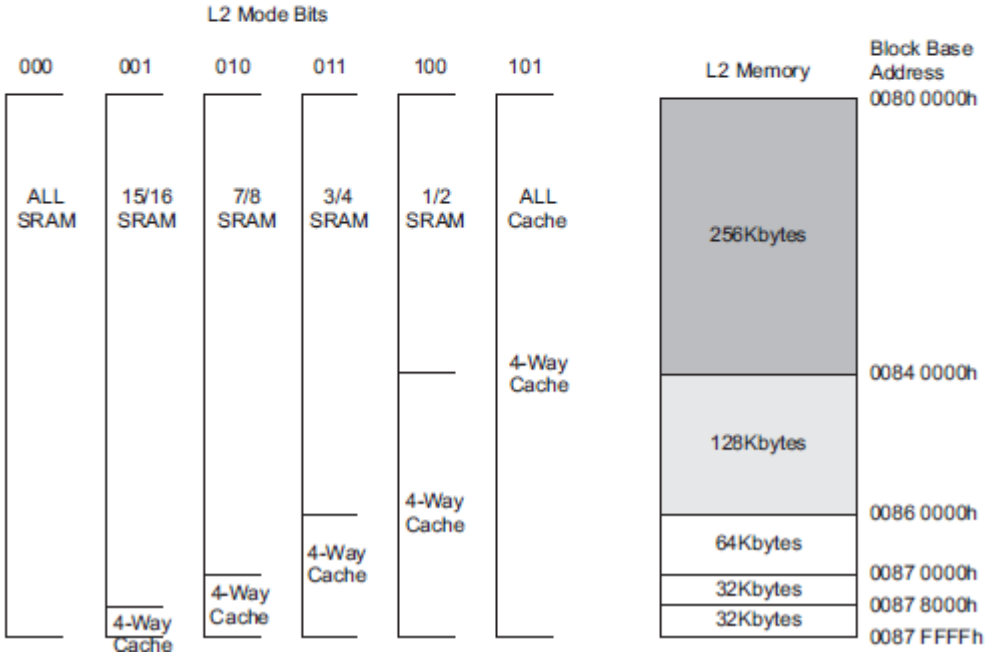
### 5.1.3 L2 Memory

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**

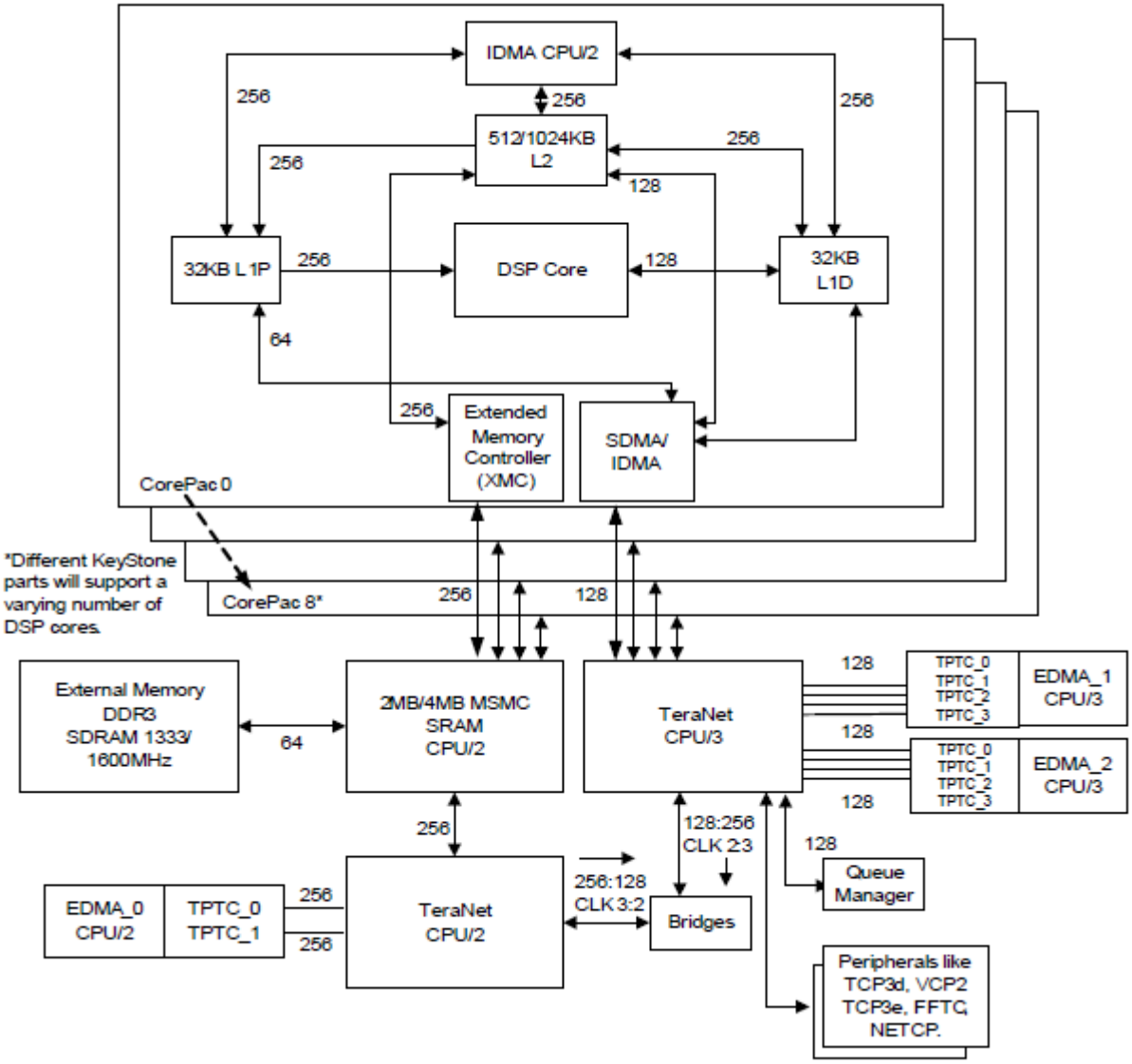



**Table 2 Theoretical Bandwidth of Core, IDMA and EDMA**

| 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 | | | | | | | |

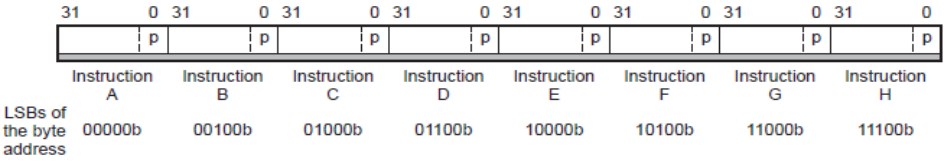
**Table 5 Memory Write Performance**

| | | | | 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**



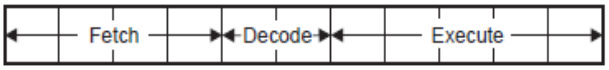

## Pipeline Operation Overview

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**

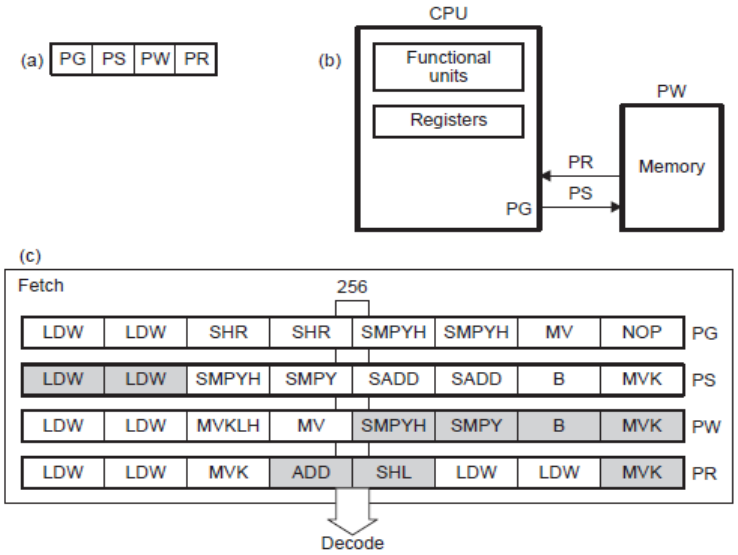



### 5.1.1 Fetch

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**

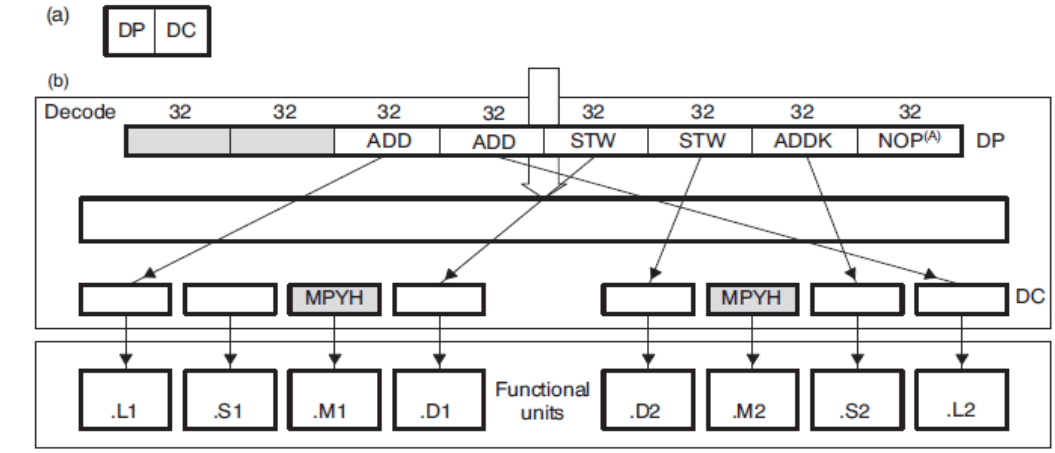



### 5.1.2 Decode

The decode phases of the pipeline are:

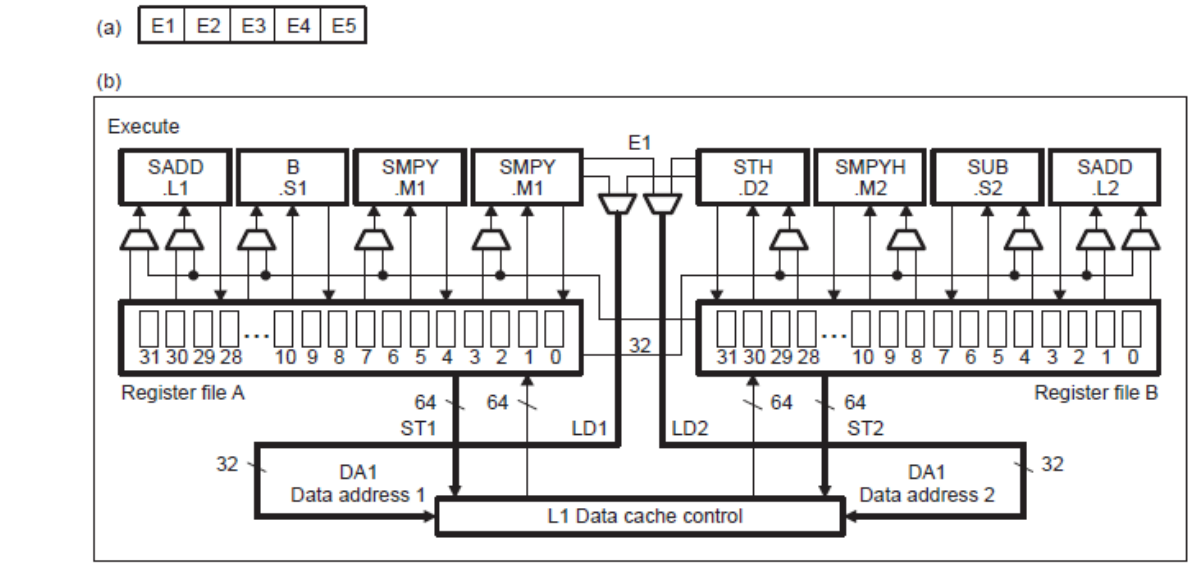
- **DP:** Instruction dispatch
- **DC:** Instruction decode

**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**

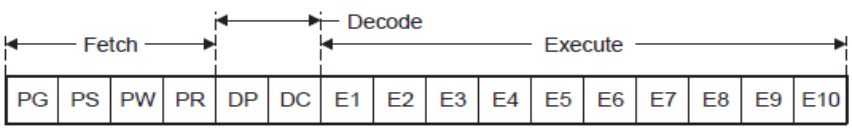
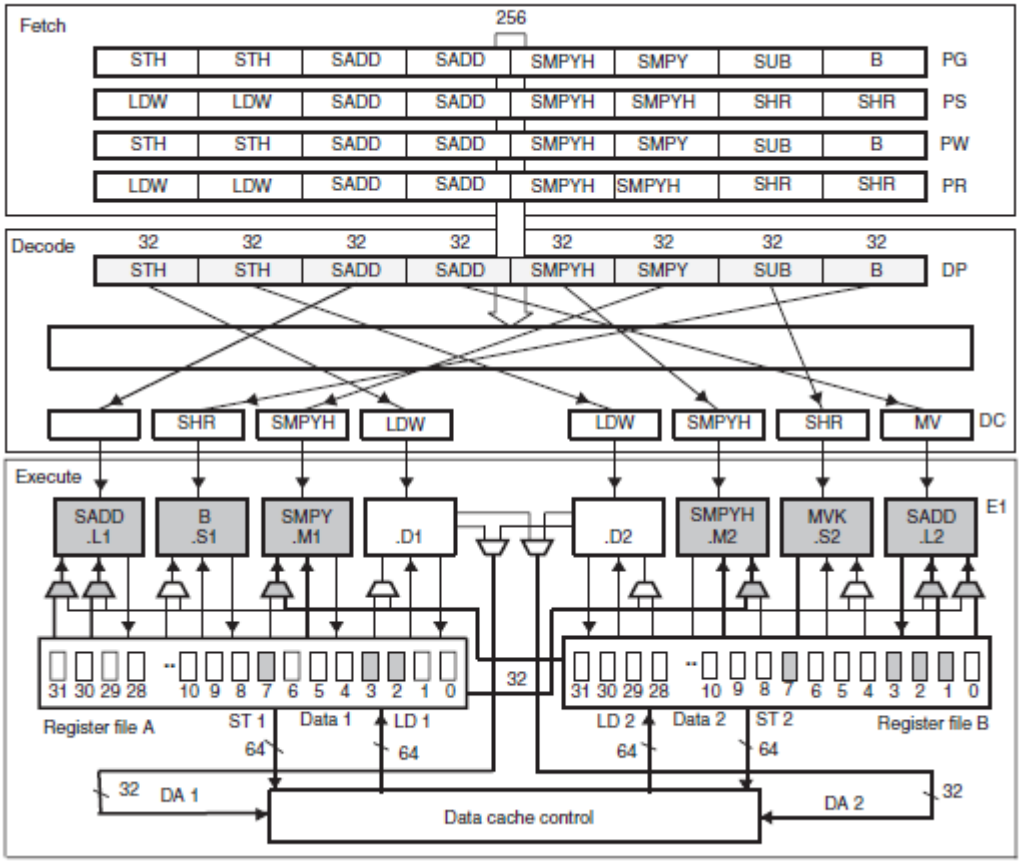


**Table 5-1 Operations Occurring During Pipeline Phases (Part 2 of 2)**

| 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 |

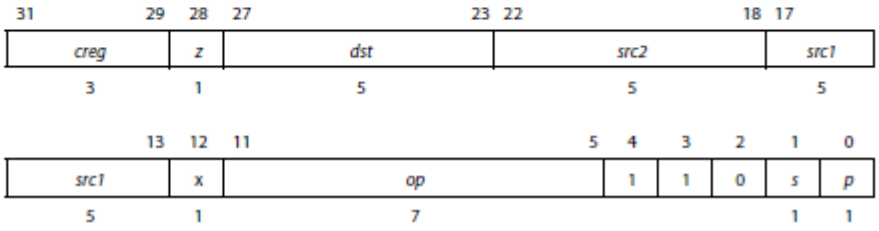
## 4.11 ADDDP

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

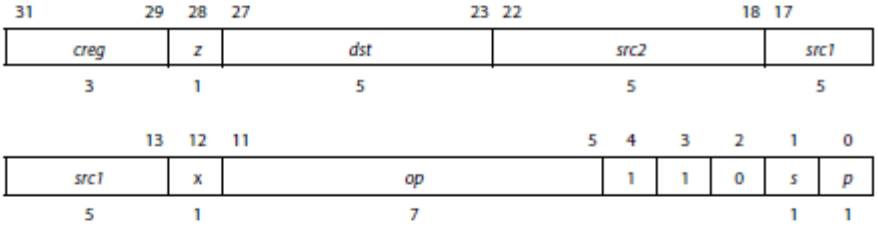
Add Two Single-Precision Floating-Point Values

***Syntax***

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

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

***Opcode***




| Opcode map field used... | For operand type... | Unit | Opfield |
|---|---|---|---|
| *src1*<br>*src2*<br>*dst* | sp<br>xsp<br>sp | .L1, .L2 | 0010000 |
| *src1*<br>*src2*<br>*dst* | sp<br>xsp<br>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

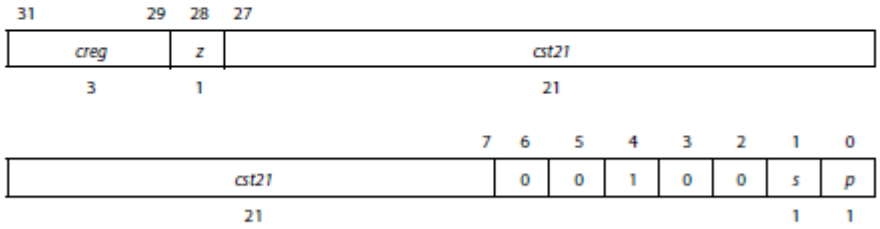
Branch Using a Displacement

***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

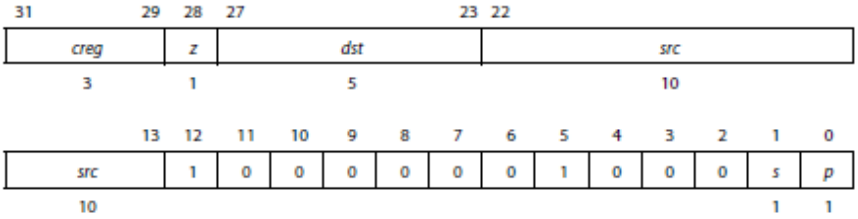
Branch and Decrement

**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

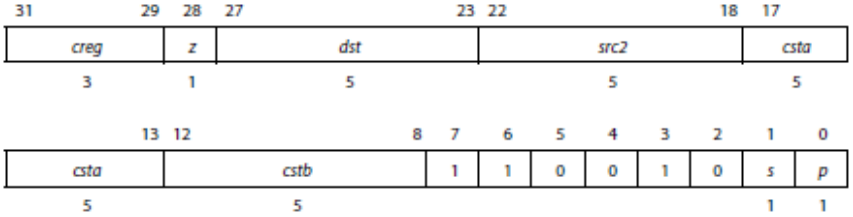
**CLR** (.unit) *src2, src1, dst*

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

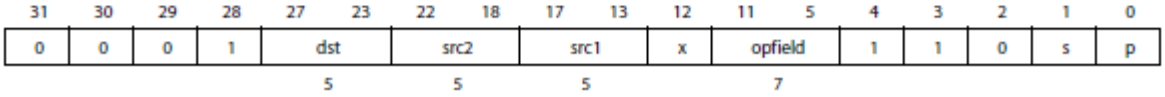
## 4.67 DADDSP

2-Way SIMD Single Precision Floating Point Addition

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

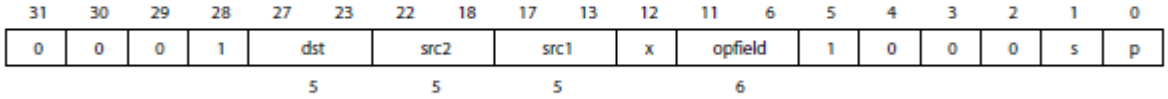
unit = .L1, .L2, .S1, or .S2

***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

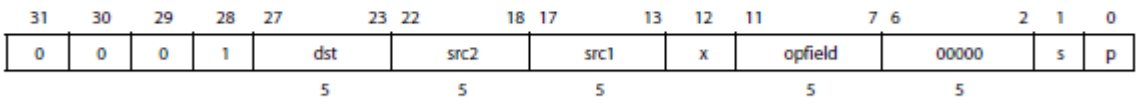
## 4.101 DMPYSP

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

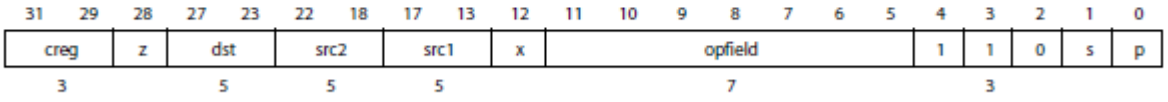
## 4.152 FADDSP

Fast Single-Precision Floating Point Add

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

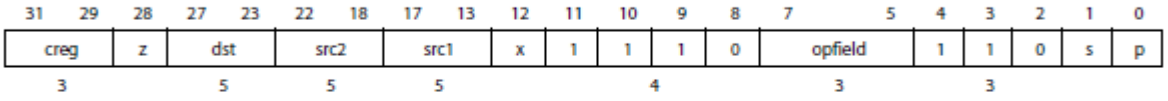
unit = .L1, .L2, .S1, or .S2

***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

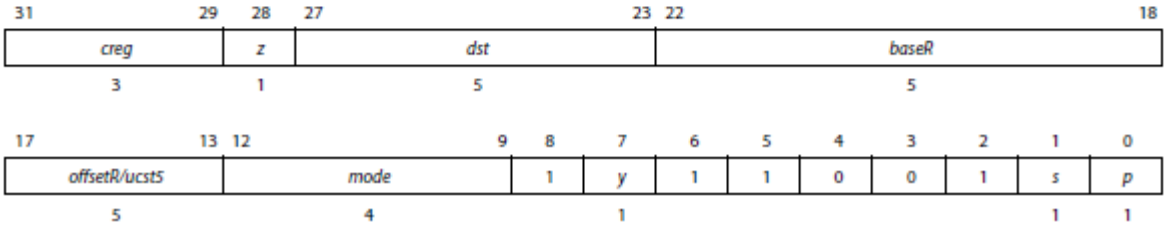
***Syntax***

```
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*<br>*offsetR*<br>*dst* | uint<br>uint<br>ullong | .D1, .D2 |
| *baseR*<br>*offsetR*<br>*dst* | uint<br>ucst5<br>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.

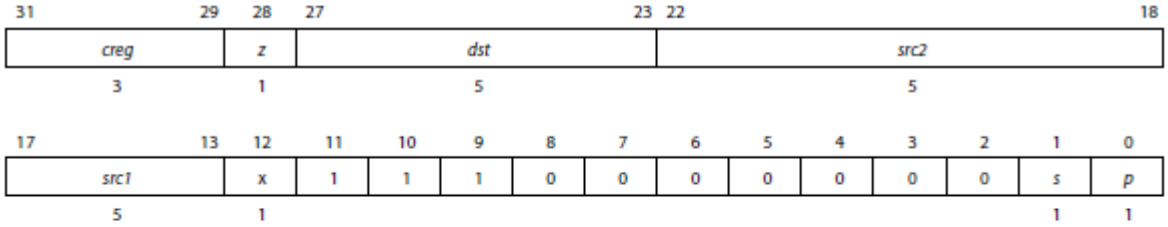
## 4.212 MPYSP

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

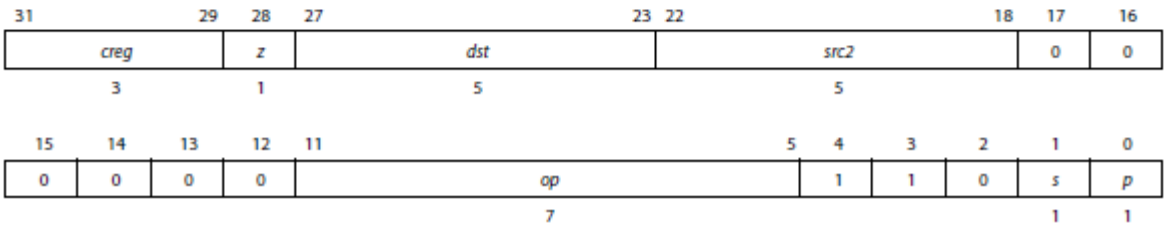
**Syntax** **MV** (.unit) *src2, dst*

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

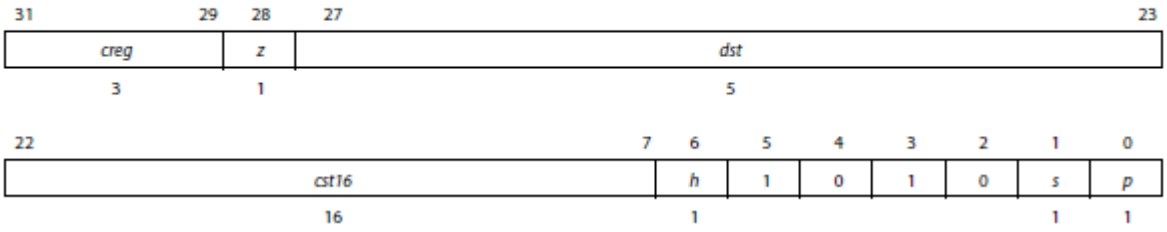
***Syntax*** **MVKH** (.unit) *cst, dst*

or

**MVKLH** (.unit) *cst, dst*

unit = .S1 or .S2

***Opcode***




***Instruction Type*** Single-cycle

***Delay Slots*** 0

***Examples*** **Example 1**

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

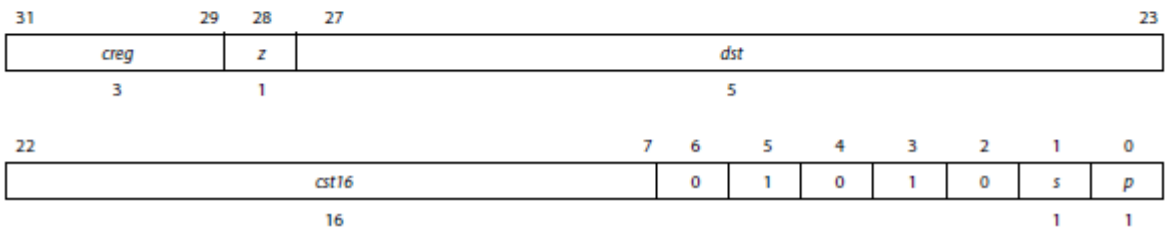
## 4.227 MVKL

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

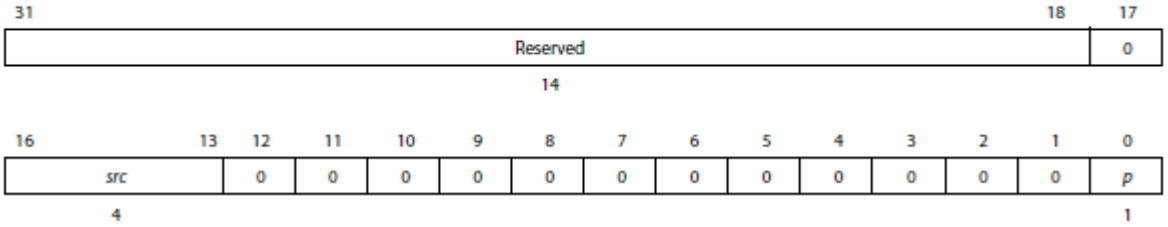
***Syntax*** **NOP** [*count*]

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
```

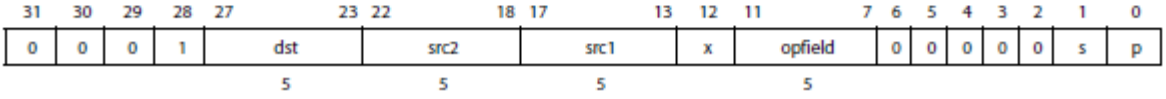
## 4.240 QMPYSP

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

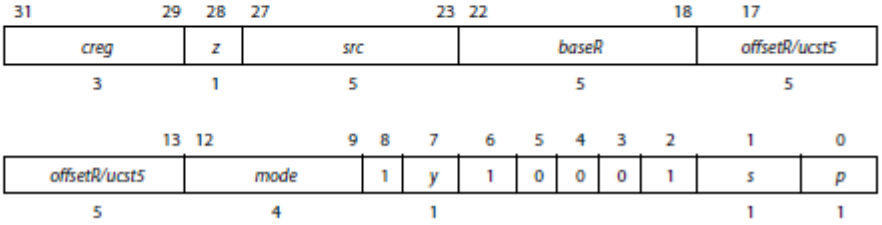
**Unsigned Constant Offset**

**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

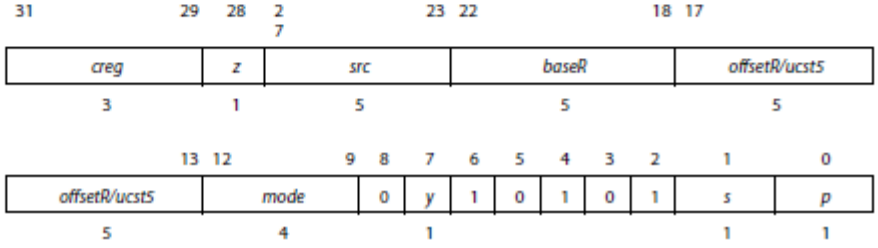
**Unsigned Constant Offset**

**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

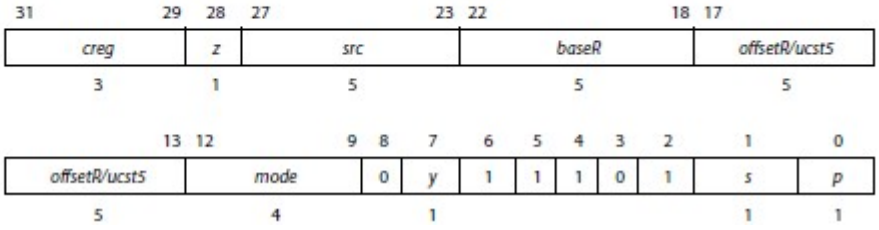
**Unsigned Constant Offset**

**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 |