Application Report
SPRAAN9A–July 2009

TMS320C28x FPU Primer

Lori Heustess *AEC DCS 2000*

ABSTRACT

This primer provides an overview of the floating-point unit (FPU) in the C2000™ Delfino microcontroller devices.

Contents

1 Introduction 2
2 Architecture 4
3 Instruction Set Overview 9
4 Interrupt Context Save and Restore 20
5 Development Tools and Documentation 24
6 Comparing the C28x+FPU to the Control Law Accelerator 26
7 References 26

List of Figures

1 C28x + FPU Functional Block Diagram 3
2 IEEE Single-Precision Floating-Point Format 4
3 C28x With Floating-Point Registers 6
4 FPU Pipeline Relationship 8
5 Floating-Point Unit Status Register (STF) 16
6 Repeat Block Register (RB) 17

List of Tables

1 IEEE Single-Precision Floating-Point Numbers 4
2 Effects of Different Rounding Modes 5
3 C28x Plus Floating-Point CPU Register Summary 7
4 Delay Slot Guidelines 9
5 Floating-Point Instruction Types - Single Cycle 10
6 Floating-Point Instruction Types - 2 Pipeline Cycles 10
7 Floating-Point Parallel Instruction Types 10
8 Types of Parallel Instructions 12
9 Floating-Point Flags 16
10 FPU Compared to CLA 26

1 Introduction

This section explains what the floating-point unit is and why it was added to the C28x™ generation of devices.

1.1 Nomenclature

The following nomenclature is used throughout this document:

- CPU is the central processing unit.
- The TMS320C28x fixed-point central-processing-unit is referred to as C28x or C28x CPU.
- The TMS320C28x plus floating-point CPU is refereed to as C28x plus floating-point or C28x+FPU.
- A floating-point instruction refers to all instructions added for floating-point support. This does not include the fixed-point CPU instruction set. These are documented in *TMS320C28x Floating Point Unit and Instruction Set Reference Guide* (SPRUEO2).
- A C28x standard instruction refers to all instructions on the fixed-point C28x CPU. These are documented in *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430).

Note: The C28x fixed-point CPU is documented in *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430). This document also applies to the C28x+FPU.

The extensions to the C28x to support floating-point are documented in *TMS320C28x Floating Point Unit and Instruction Set Reference Guide* (SPRUEO2). This document should be considered as a supplement to *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430).

1.2 What is the C28x Plus Floating Point Unit?

The C28x+FPU offers the best of two worlds:

- Everything the fixed-point C28x CPU has
- IEEE 32-bit floating-point format support

The C28x+FPU is a 32-bit fixed-point processing unit with IEEE 32-bit floating-point format support. This CPU draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The C28x devices features include a modified Harvard architecture and circular addressing. The RISC features are single-cycle instruction execution, register-to-register operations, and modified Harvard architecture. The microcontroller features include ease of use through an intuitive instruction set, byte packing and unpacking, and bit manipulation. The modified Harvard architecture of the CPU enables instruction and data fetches to be performed in parallel. The CPU can read instructions and data while it writes data simultaneously to maintain the single-cycle instruction operation across the pipeline. The CPU does this over six separate address/data buses.

Figure 1 shows the functional block diagram of the C28x+FPU. Those familiar with the C28x will find that no changes have been made to existing:

- C28x Instructions
- C28x Pipeline
- C28x Emulation
- Memory Bus Architecture

New instructions to support floating-point operations have been added as an extension to the standard C28x instruction set. This means code written for the C28x fixed-point CPU is 100% compatible with the C28x+FPU. This allows for mixing and matching fixed-point and floating-point in your application. Devices with the C28x+FPU connect the floating-point overflow and underflow (LVF, LUF) flags to the peripheral interrupt expansion (PIE) block. This makes debug overflow and underflow issues within your application much easier.

The first devices to include the C28x+FPU are the TMS320F2833x and TMS320C2834x family of microcontrollers. Look for more information on the Texas Instruments website at www.ti.com/delfino.

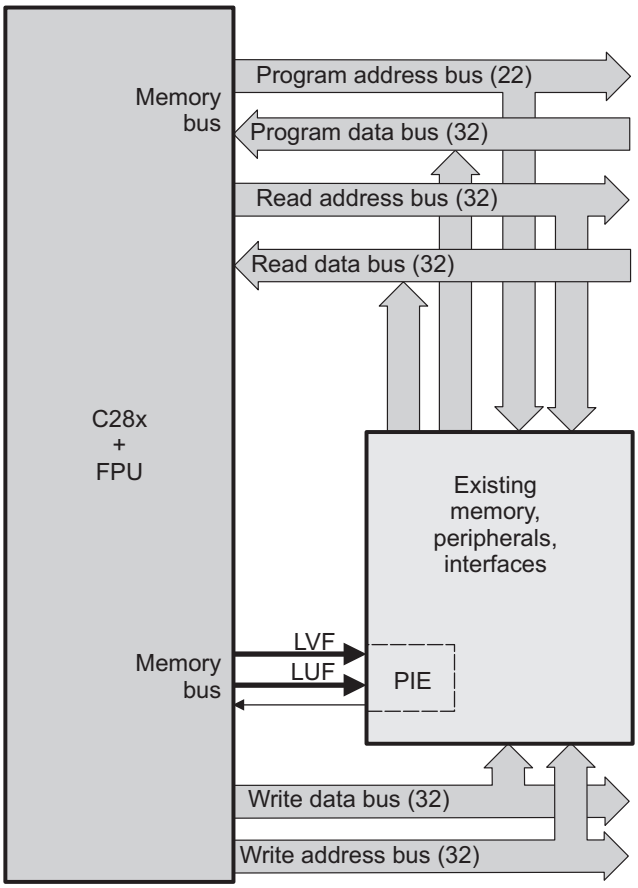


Figure 1. C28x + FPU Functional Block Diagram

1.3 Why Floating-Point for C2000?

The addition of floating-point brings performance improvements to control type algorithms for the C2000 platform. Since the C28x+FPU is backward compatible with the C28x, you are free to choose the appropriate data type for your application. Floating-point has the following advantages over fixed-point:

- Developer Requests
  Developers like floating-point. It is easier to code in than fixed-point and the resulting code is inherently more robust.
- Performance Boost
  Many algorithms used in control applications see a performance boost from native floating-point. For example, division, square root, sin, cos, FFT and IIR all benefit from floating-point.
- Simplified software development common to floating-point processors.
  Coding in floating-point is more C/C++ friendly than coding in fixed-point.

- Robustness

  Software developers typically begin creating algorithms in a floating-point environment for validation, and then convert the code to run on fixed-point devices. Now, however, you can eliminate time spent contending with scaling, saturation and adjustment of numerical resolution required in fixed-point implementations. The scaling and saturation burden seen in fixed-point is removed. In fixed-point, if a value overflows it can result in inversion of the data but in floating-point the value will automatically saturate and does not cause inversion.

1.4 Support for Long Double (64-Bit Floating Point)

All C28x devices support the long double (64-bit float) data type through the standard runtime support library that comes with the compiler. It should be noted that the Delfino microcontrollers only support 32-bit float in hardware. 64-bit float is performed in software. Consider looking at 64-bit integer math (long long) if this level of accuracy is required. It may be much faster then 64-bit float-point support in software.

2 Architecture

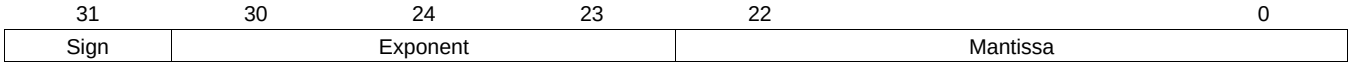
This section describes the FPU format in terms of IEEE standards. It also gives a brief overview of the register set, pipeline, and working with delay slots.

2.1 IEEE Single-Precision Floating-Point Format

The C28x+FPU follows the Institute of Electrical and Electronics Engineers, Inc. (IEEE) 754 format standard for single-precision floating-point. The single-precision (32-bit) number format includes:

- 1 sign bit: 0 means the value is positive and 1 means the value is negative.
- 8-bit exponent: This exponent is biased to allow for both positive and negative exponents.
- 23-bit mantissa: The mantissa includes an implicit leading 1 plus the fractional bits.

Figure 2. IEEE Single-Precision Floating-Point Format



The types of numbers defined by the standard are shown in Table 1.

Table 1. IEEE Single-Precision Floating-Point Numbers

Sign	Exponent	Mantissa	Value
0	0	0	Positive zero
1	0	0	Negative zero
0 or 1	0	non-zero	Denormalized number (1)
0	1-254	0x00000 - 0x7FFFF	Normal range of positive numbers (2)
1	1-254	0x00000 - 0x7FFFF	Normal range of negative numbers (2)
0	255 (max)	0	Positive infinity
1	255 (max)	0	Negative infinity
0 or 1	255 (max)	non-zero	Not a number (NaN)

(1) Denormalized values are very small. They are calculated using the formula $(-1)^{S} \times 2^{(E-126)} \times 0.M$
(2) Normalized values are calculated using the formula: $(-1)^{S} \times 2^{(E-127)} \times 1.M$

The normal range of positive and negative numbers are calculated using the formula:

$(-1)^{S} \times 2^{(E-127)} \times 1.M.$

This results in numbers in the range $\pm$ ~1.7×10^{-38} to $\pm$ ~$3.4 \times 10^{+38}$. Notice that these numbers are normalized and have a hidden 1. Thus the equivalent signed integer resolution is the number of mantissa bits + sign + 1.

The IEEE 754 standard is the most widely used standard for floating-point numbers. This standard includes:

- Standard number formats and special values such as not-a-number and infinity
- Standard rounding modes and floating point operations
- A standard used on many platforms including the Texas Instruments C67x devices

Some simplifications to the standard have been made for the C28x+FPU. These are common ways of handing these numbers:

- The status flags and compare operations treat negative zero as positive zero.
- Denormalized numbers are very small and therefore are treated as zero.
- Not-a-number is very large and looks similar to infinity. On the C28x+FPU, NaN is treated as infinity.
- The IEEE 754 standard supports five rounding modes. The C28x+FPU supports two of these modes:
  - Truncate
  - Round to nearest, tie to even. In this mode the value is rounded to the nearest value. If a value falls half way between two values, then it is rounded to the even value.

Table 2 shows how different rounding modes can affect an average calculation. To simplify the example, small numbers have been used. Each column shows the value that results from that particular rounding mode. When the values are averaged, the round-to-nearest result converges to the same result as the actual values without rounding. This is why round to nearest is the most commonly used rounding mode. Code generated by the C28x+FPU compiler by default configures the processor to use round to nearest. In your assembly code you can set this rounding mode with the SET STF instruction.

Table 2. Effects of Different Rounding Modes

Actual Value (Binary)	Actual Value	Round to Nearest (tie to Even) (1)	Truncate (1)	Round to +Infinity (1)	Round to -Infinity (1)	+.5 LSB (2)
010.111	2.875	3.000	2.000	3.000	2.000	3.000
010.110	2.750	3.000	2.000	3.000	2.000	3.000
010.101	2.625	3.000	2.000	3.000	2.000	3.000
010.100	2.5	2.000	2.000	3.000	2.000	3.000
010.011	2.375	2.000	2.000	3.000	2.000	2.000
010.010	2.250	2.000	2.000	3.000	2.000	2.000
010.001	2.125	2.000	2.000	3.000	2.000	2.000
010.000	2.00	2.000	2.000	2.000	2.000	2.000
001.111	1.875	2.000	1.000	2.000	1.000	2.000
001.110	1.750	2.000	1.000	2.000	1.000	2.000
001.101	1.625	2.000	1.000	2.000	1.000	2.000
001.100	1.500	2.000	1.000	2.000	1.000	2.000
001.011	1.375	1.000	1.000	2.000	1.000	1.000
001.001	1.125	1.000	1.000	2.000	1.000	1.000
001.000	1.00	1.000	1.000	1.000	1.000	1.000
Average:	1.9375	1.9375	1.5000	2.3570	1.5000	2.067
Delta:	0.0000	0.0000	-0.4275	+0.4375	-0.4275	+0.1292

(1) IEEE rounding mode. Only round to nearest (tie to even) and truncate are supported by the C28x+FPU.
(2) +.5 LSB rounding is typically used in fixed-point math.

2.2 C28x+FPU Register Set

The C28x+FPU register set is shown in Figure 3. The register set consists of:

- The standard C28x fixed-point register set
  - 32-bit accumulator, product and temporary register
  - 8, 32-bit extended auxiliary registers
  - 22-bit program counter and return program counter
  - 16-bit data page and stack pointers
  - 16-bit status registers and interrupt control registers
- Additional registers to support floating-point.
  - 8, 32-bit result registers (R0H-R7H)
  - 32-bit floating-point status register
  - 32-bit repeat block register

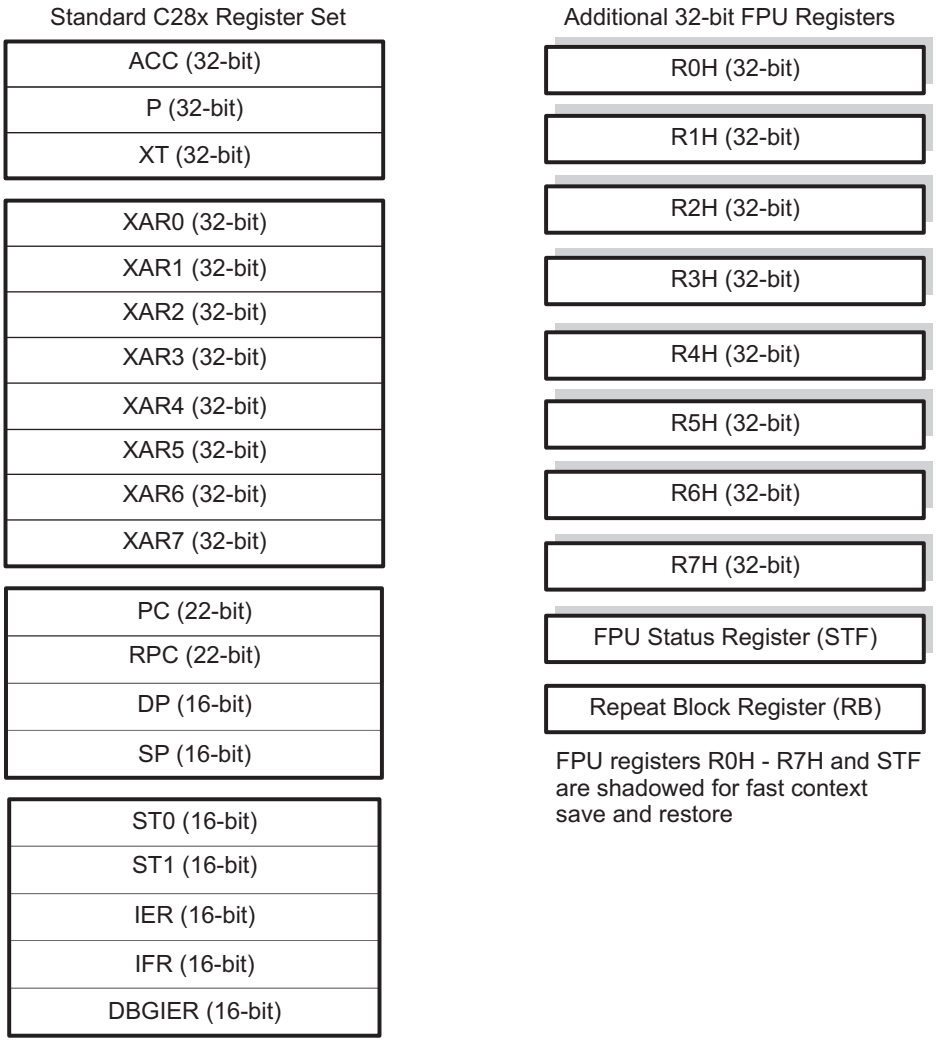


Figure 3. C28x With Floating-Point Registers

The floating-point result and status registers are all shadowed. If nested interrupts are not allowed within an interupt service routine, then the registers can be copied to their shadow registers instead of the stack. This document refers to this as a high-priority interrupt. This results in a very efficient interrupt response. See Section 4 for more information.

Table 3. C28x Plus Floating-Point CPU Register Summary

Register	C28x CPU	C28x+FPU	Size	Description
ACC	Yes	Yes	32 bits	Accumulator
XAR0	Yes	Yes	16 bits	Auxiliary register 0
XAR1	Yes	Yes	32 bits	Auxiliary register 1
XAR2	Yes	Yes	32 bits	Auxiliary register 2
XAR3	Yes	Yes	32 bits	Auxiliary register 3
XAR4	Yes	Yes	32 bits	Auxiliary register 4
XAR5	Yes	Yes	32 bits	Auxiliary register 5
XAR6	Yes	Yes	32 bits	Auxiliary register 6
XAR7	Yes	Yes	32 bits	Auxiliary register 7
DP	Yes	Yes	16 bits	Data-page pointer
IFR	Yes	Yes	16 bits	Interrupt flag register
IER	Yes	Yes	16 bits	Interrupt enable register
DBGIER	Yes	Yes	16 bits	Debug interrupt enable register
P	Yes	Yes	32 bits	Product register
PC	Yes	Yes	22 bits	Program counter
RPC	Yes	Yes	22 bits	Return program counter
SP	Yes	Yes	16 bits	Stack pointer
ST0	Yes	Yes	16 bits	Status register 0
ST1	Yes	Yes	16 bits	Status register 1
XT	Yes	Yes	32 bits	Multiplicand register
R0H	No	Yes	32 bits	Floating-point result register 0
R1H	No	Yes	32 bits	Floating-point result register 1
R2H	No	Yes	32 bits	Floating-point result register 2
R3H	No	Yes	32 bits	Floating-point result register 3
R4H	No	Yes	32 bits	Floating-point result register 4
R5H	No	Yes	32 bits	Floating-point result register 5
R6H	No	Yes	32 bits	Floating-point result register 6
R7H	No	Yes	32 bits	Floating-point result register 7
STF	No	Yes	32 bits	Floating-point status register
RB	No	Yes	32 bits	Repeat block register

2.3 C28x+FPU Pipeline

The C28x has an 8-stage pipeline:

- Two fetch stages (F1 and F2)
- Two decode stages (D2 and D2)
- Two read stages (R1 and R2)
- One execute stage (E)
- One write stage (W)

On C28x+FPU, the C28x standard instructions follow the same 8-stage pipeline. That is, no changes have been made to the C28x pipeline or instruction behavior. The 8-state pipeline is described in *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430).

In the decode 2 (D2) stage, it is determined whether an instruction is a floating-point instruction. If it is, then the pipeline behavior changes. The instruction will go through an additional decode stage (D). After the decode stage any data to be read will be fetched. The data read aligns with the C28x R2 stage where data is read for standard C28x instructions. Wait states will stall the floating-point instruction. The floating-point pipeline has an execute (E1) and write stage (W) that both align with the corresponding C28x pipeline. Wait states for write accesses will stall the floating-point instruction just as on the C28x.

Everything described so far has the C28x and FPU pipelines in lock step. This is not the case when a floating-point instruction requires an additional execute phase (E2). This is the case for math and conversion instructions. This results in an additional cycle before the instruction completes. Such operations are not pipeline protected. Instructions that require an additional execution phase require a software delay slot for the operation to complete. This delay slot is any instruction that does not use the result register of the instruction that requires the delay slot. When needed, you will Insert a NOP or any other non-conflicting instructions between operations. Examples of this are shown in Section 3.

Figure 4 shows the pipeline for C28x standard instructions and floating-point instructions.

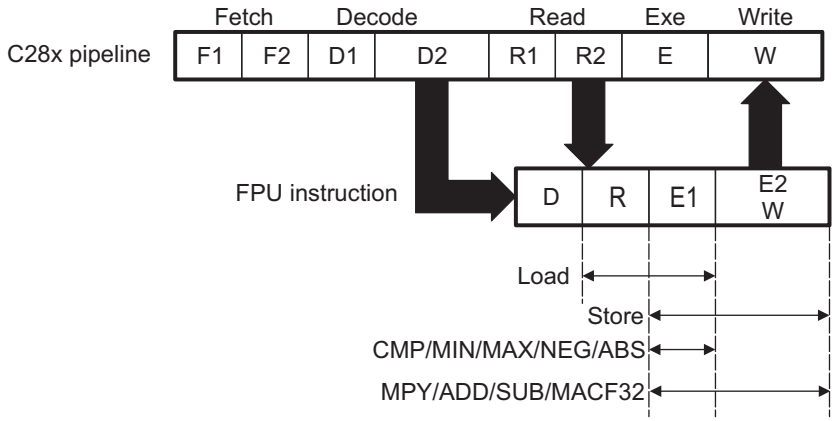


Figure 4. FPU Pipeline Relationship

2.4 Working with Delay Slots

Instructions that require a delay are indicated in the documentation with a ‘p’ after their cycle count. For example ‘2p cycles’ stands for 2 pipelined cycles. This means that a new instruction can be started every cycle, but the result from that operation will not be available until one cycle later. More examples of this are given in Section 3.

As a first pass, you can simply fill delay slots with a NOP (no operation) instruction. This will use a single cycle, but does not do any useful work. Later, to improve performance, you can remove the NOPs and instead insert non-conflicting instructions into the delay slots.

There are three general guidelines for when a delay cycle is required. These guidelines are simple and easy to recall. They are given to help you write your code, but keep in mind the C28x+FPU assembler will issue an error if you have improperly filled a delay slot and created a pipeline conflict.

The three guidelines are shown in Table 4:

Table 4. Delay Slot Guidelines

Instruction Type	Examples	Cycles	Delay Slot Requirements
Floating Point Math (1)	ADDF32, SUBF32, MPYF32, MACF32	2p	1 Delay Slot
Conversion	I16TOF32, F32TOI16, F32TOI16R, etc…	2p	1 Delay Slot
Everything Else (2)	Load, Store, Compare, Min, Max, Absolute and Negative value	1	No Delay Slot Required

(1) The single-repeatable MACF32 instruction does not require alignment cycles. Other versions of MACF32 are, however, 2p and require a delay slot.
(2) Moves between CPU and FPU registers have special pipeline alignment requirements.

3 Instruction Set Overview

This section provides information on the following:

- Instruction set
- Types of floating-point instructions
- 2 pipeline-cycle (2p) instruction examples
- Parallel instructions
- Taking advantage of delay slots
- Using floating-point flags
- The repeat block instruction (RPTB)
- Moves between C28x and FPU registers

3.1 Instruction Format

The floating-point instructions follow the same format as the C28x instructions. That is, the destination operands are always on the left and the source operands are on the right.

Example 1 shows two multiply operations. The first multiply is a fixed-point multiply. The source comes from the T register and the 16-bit location indicated by loc16. The result of the multiply is stored in the accumulator (ACC).

Example 1. Example Instructions

```
MPY     ACC, T, loc16    ; Fixed-point multiply
MPYF32  R0H, R1H, R2H    ; Floating-point multiply
```

The second multiply is a floating-point operation. In this case, the source operands are R1H and R2H. The result of the multiply is stored in R0H.

To enable the compiler to generate floating-point instructions, you must tell it that you have a C28x device that supports floating-point. This is done by using the compiler switch: - -float_support = fpu32 which is available in C28x codegen tools V5.0 and later.

3.2 Types of Floating-Point Instructions

The following tables show a summary of the types of instructions that have been added to the C28x+FPU. Note that these instructions are in addition to the standard C28x instruction set. Table 5 shows the type of single cycle instructions that are available. Notice each of these operations takes a single cycle and therefore do not require a delay slot.

Table 5. Floating-Point Instruction Types - Single Cycle

Register	Example	Cycles
Load (Conditional)	MOV32 R0H, mem32	1
Store	MOV32 mem32, R1H	1
Load with Data Move	MOVD32 R3H, mem32	1
FPU Register to C28x Register	MOV32 XAR6, R2H	1 (1)
C28x Register to FPU Register	MOV32 R3H, XAR7	1 (1)
Compare, Min, Max	CMPF32 R2H, R3H	1
Absolute Value, Negative Value	ABSF32 R2H, R3H	1
Context Save FPU Registers	SAVE	1
Context Restore FPU Registers	RESTORE	1

(1) Moves between FPU and C28x registers require additional pipeline alignment. See Section 3.8 for details.

Table 6 shows an overview of 2p, or 2-pipelined cycle, instructions. These are the math type and conversion type instructions and require 1 cycle delay for the result to be updated.

Table 6. Floating-Point Instruction Types - 2 Pipeline Cycles

Register	Example	Cycles
Unsigned Integer To Float	UI16TOF32 R1H, mem32	2p
Integer to Float	I32TOF32	2p
Float to Integer and Round	F32TOI16R	2p
Float to Integer	F32TOI32	2p
Multiply, Add, Subtract	MPYF32 R2H, R1H, R0H	2p
MAC		2P
1/x Estimate	EINVF32 R2H, R1H	2p
1/√x Estimate	EISQRTF32 R3H, R0H	2p
Repeat MAC	RPT (#N-1) || MACF32 R7H,R3H,mem32,*XAR7++	3 + N

The floating-point unit also supports some parallel operations. These are shown in and are discussed in more detail in Section 3.4. Notice that the cycle information is given for both of the operations within the instruction.

Table 7. Floating-Point Parallel Instruction Types

Register	Example	Cycles
Min or Max and Parallel Move	MINF32 R0H, R4H || MOV32 R1H, R2H	1/1
Multiply and Parallel Add or Subtract	MPYF32 R0H, R1H, R2H || ADDF32 R4H, R4H, R2H	2p/2p
Multiply, Add or Subtract, MAC and Parallel Load	MPYF32 R0H, R1H, R2H || MOV32 R4H, mem32	2p/1
Multiply, Add or Subtract, MAC and Parallel Store	MPYF32 R0H, R1H, R2H || MOV32 mem32, R0H	2p/1

3.3 2 Pipeline-Cycle (2p) Instruction Examples

Consider the following floating-point instruction shown in Example 2:

Example 2. Floating-Point Multiply

```
    MPYF32 R2H, R1H, R0H
```

The instruction is a floating-point multiply. Recall from the guidelines given in Table 4 that any floating-point math instruction is a 2p instruction. This means it requires 1-cycle delay for the result (in R2H) to be available. As a first step, you can simply add a NOP after the multiply to take care of this requirement. The NOP will take up a cycle, but does no useful work.

Example 3. Using a NOP in a Delay Slot

```
MPYF32 R2H, R1H, R0H  ; 2p instruction
NOP                   ; 1 cycle delay
                      ; <- MPYF32 completes, R2H valid
<any instruction>     ; Can use R2H
                      ;
```

To improve performance, you can replace the NOP with any non-conflicting instruction. In this case, this means any instruction that does not use R2H, which is the result register for the multiply operation. In Example 4 the NOP has been replaced with a floating-point addition instruction.

Example 4. Using a Non-Conflicting 2p Instruction in a Delay Slot

```
MPYF32 R2H, R1H, R0H  ; 2p instruction
ADDF32 R3H, R3H, R1H  ; 1 cycle delay for MPYF32
                      ; <- MPYF32 completes, R2H valid
NOP                   ; 1 cycle delay for ADDF32
                      ; <- ADDF32 complete, R3H valid
<any instruction>     ; Can use R3H
```

The addition instruction, like all math instructions, takes 2 pipeline cycles and therefore requires 1 delay cycle for its result to be valid. A NOP has been added after the addition to allow it to complete. As with the previous case, this NOP can be replaced by any non-conflicting instruction that does not use R3H. In Example 5, the NOP has been replaced by a store (MOV32) instruction.

Example 5. Using a Non-Conflicting Single Cycle Instruction in a Delay Slot

```
MPYF32 R2H, R1H, R0H  ; 2p instruction
ADDF32 R3H, R3H, R1H  ; 1 cycle delay for MPYF32
                      ; <- MPYF32 completes, R2H valid
MOV32 *XAR7, R2H      ; 1 cycle delay for ADDF32
                      ; <- ADDF32 complete, R3H valid
<any instruction>     ; Can use R3H
```

The MOV32 copies the contents of R2H, the result from the multiply, into the location pointed to by the XAR7 register. This does not cause a pipeline conflict since the multiply has had time to complete. The MOV32 instruction is single cycle and it does not require a delay slot.

3.4 Parallel Instructions

In the previous section, all of the operations were individual instructions. That is, each had its own opcode and each started sequentially. The C28x+FPU also includes special parallel instructions. Parallel instructions are a single instruction, single opcode, that performs two operations. Parallel bars before the 2nd operation indicate that the two operations belong to a parallel instruction. The following example shows an addition with parallel store instruction.

Example 6. Addition with Parallel Store

```
    ADDF32 R3H, R3H, R1H
||  MOV32 *XAR7, R3H
```

There are three types of parallel instructions available. These are summarized in Table 8.

Table 8. Types of Parallel Instructions

Register	Example	Cycles
Min or Max and Parallel Move	MINF32 R0H, R4H || MOV32 R1H, R2H	1/1
Multiply and Parallel Add or Subtract	MPYF32 R0H, R1H, R2H || ADDF32 R4H, R4H, R2H	2p/2p
Multiply, Add or Subtract, MAC and Parallel Load or Store	MPYF32 R0H, R1H, R2H || MOV32 R4H, mem32	2p/1

For a parallel instruction the cycle count includes two numbers; one for each operation. The guidelines for delay slots are the same as described in Section 2.4. Math operations take 2p cycles while load, store, min and max are all single cycle. One thing that is key to understanding parallel instructions is the knowledge that both operations are part of a single instruction. The 2nd operation is not in the delay slot of the first. For example, consider a multiply with parallel addition shown in Example 7. This is a 2p/2p instruction. One delay cycle must be added after the instruction to allow both the multiply and subtract to complete. Since both operations are started at the same time, additional delays are not required.

Example 7. 2p/2p Parallel Instruction Requires 1 Delay Slot

```
   MPYF32 R2H, R1H, R0H ; 2p/2p instruction
|| ADDF32 R5H, R4H, R1H
   NOP                  ; Delay for MPYF32 and ADDF32
                        ; <- R2H and R5H updated
   <any instruction>    ; Can use R2H
```

Consider the following 2p/1 parallel instruction which is a multiply with parallel store instruction:

Example 8. Multiply with Parallel Store

```
   MPYF32 R2H, R1H, R0H
|| MOV32  *XAR3, R2H
```

The multiply, like all math operations, takes 2 pipeline cycles and the load operation is single cycle. That means the load will finish immediately after the instruction and no delay is required. A single NOP has been added after the instruction to allow the multiply to complete.

Example 9. Using a NOP to Fill the Delay Slot of a Parallel Instruction

```
   MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32  *XAR3, R2H
                        ; <- MOV32 complete
   NOP                  ; Delay for MPYF32
                        ; <- R2H updated
   <any instruction>    ; Can use R2H
```

Notice that the MOV32 operation uses the R2H register and the R2H register is also the destination for the multiply. Remember the MOV32 operation is not in the delay slot of the MPYF32; it is part of the same instruction. Because the two operations are part of a parallel instruction MOV32's use of R2H does not result in a pipeline conflict. Both operations are started at the same time and MOV32 uses the value in R2H before the multiply. Without the parallel bars, however, MOV32 would instead be in the delay slot for the MPYF32. In this case, they would result in a pipeline conflict and the assembler would issue an error.

Example 10. Using the Result Register in a Parallel Operation

```
   ; Before: R0H = 2.0, R1H = 3.0, R2H = 10.0
; MOV32 uses the value of R2H before the multiply

   MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32  *XAR3, R2H
                        ; <- MOV32 complete
   NOP                  ; Delay for MPYF32
                        ; <- R2H updated
   <any instruction>    ; Can use R2H

; After: R2H = R1H * R0H = 3.0 * 2.0
;        *XAR3 = 10.0
```

As before, the NOP can be replaced by any non-conflicting instruction. In this case, any instruction that does not use R2H will work as shown in example Example 11. Do not worry, though, the assembler will report an error if you create a pipeline conflict.

Example 11. Filling a Delay Slot With a Non-Conflicting Instruction

```
   MPYF32 R2H, R1H, R0H ; 2p/1 instruction
|| MOV32  *XAR3, R2H
                        ; <- MOV32 complete
   MOV32  R1H, *XAR4    ; Delay for MPYF32
                        ; <- R2H updated, R1H updated
   ADDF32 R2H, R2H, R1H ; Can use R2H
```

3.5 Taking Advantage of Delay Slots

In the previous sections you saw how performance can be improved by using non-conflicting instructions within the delay slots. The good news is most instructions can be used within a delay slot location. There are really only three things that will create a pipeline conflict. In each of these cases, the C28x+FPU assembler will issue a warning if you have created a conflict.

- Destination and source register conflicts with the instruction requiring the delay slot.
  The delay slot is needed in order for the result from the operation to update. Therefore, the instruction within the delay slot should not use this register as either a source or a destination.
- Instructions that read or modify the floating-point status register flags.
  The instruction that requires the delay may access the STF register. Therefore, placing an instruction in the delay slot that reads or modifies STF will cause a conflict. These instructions are SAVE, SETFLG, RESTORE and MOVEST0.
- Moves between the C28x CPU and FPU registers have special pipeline alignment requirements.
  These requirements are shown in detail in Section 3.8. These types of move operations are infrequent.

To see how using delay slots can considerably improve performance, consider the code shown in Example 12. The code shown is hand-coded, 32-bit, fixed-point code for two calculations: Y1 = m1 * X1+B1 and Y2 = m2 * X2+B2.

Example 12. 32-Bit Fixed-Point Code: Y=mX+B

```
   ; C28x 32-bit fixed-point
; Y1=(M1*X1)>> Q + B1

  MOVL    XT,@M1
  IMPYL   P,XT,@X1
  QMPYL   ACC,XT,@X1
  ASR64   ACC:P,#Q
  ADDL    ACC,@B1
  MOVL    @Y1,ACC

; Y2=(M2*X2)>> Q + B2

  MOVL    XT,@M2
  IMPYL   P,XT,@X2
  QMPYL   ACC,XT,@X2
  ASR64   ACC:P,#Q
  ADDL    ACC,@B2
  MOVL    @Y2,ACC

; 14 cycles
```

If the fixed-point code in Example 12 is recoded into floating-point, the result may look like the code shown in Example 13. In this case, the coding was done by hand and NOPs were used to fill the delay slots required by the FPU math instructions. The performance of the floating-point code and the fixed-point code is equal; both take 14 cycles.

Example 13. 32-Bit Floating-Point Code: Y=mX+B

```
   ; C28x+FPU 32-bit floating-point
; Y1 = M1*X1 + B1

   MOV32  R0H,@M1
   MOV32  R1H,@X1
   MPYF32 R1H,R1H,R0H
|| MOV32  R0H,@B1
   NOP                    ; delay for MPYF32
   ADDF32 R1H,R1H,R0H
   NOP                    ; delay for ADDF32
   MOV32  @Y1,R1H

; Y2 = M2*X2 + B2

   MOV32  R0H,@M2
   MOV32  R1H,@X2
   MPYF32 R1H,R1H,R0H
|| MOV32  R0H,@B2
   NOP                    ; delay for MPYF32
   ADDF32 R1H,R1H,R0H
   NOP                    ; delay for ADDF32
   MOV32  @Y2,R1H

; 14 cycles
```

The performance of the floating-point code can be improved by taking advantage of the delay slots. Example 14 shows the floating-point code written such that the delay slots are used by productive non-conflicting instructions. In this case both calculations take only 2 cycles more than a single calculation shown in Example 12 and Example 13. Optimizations such as this are possible using the compiler, not just for hand-coded assembly.

Example 14. Optimized 32-Bit Floating-Point Code: Y=mX+B

```
   ; C28x+FPU 32-bit optimized floating-point
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2

   MOV32   R2H,@X1
   MOV32   R1H,@M1
   MPYF32  R3H,R2H,R1H
|| MOV32   R0H,@M2
   MOV32   R1H,@X2
   MPYF32  R0H,R1H,R0H
|| MOV32   R4H,@B1
   ADDF32  R1H,R4H,R3H
|| MOV32   R2H,@B2
   ADDF32  R0H,R2H,R0H
   MOV32   @Y1,R1H
   MOV32   @Y2,R0H

; 9 cycles
```

3.6 Using Floating-Point Flags

The C28x+FPU has three status registers: ST0, ST1, and STF. ST0 and ST1 are identical to the C28x status registers. STF has been added and responds to floating-point instructions. The bit fields of the STF register are shown in Figure 5.

Figure 5. Floating-Point Unit Status Register (STF)

31	30											16
SHDWS	Reserved											
R/W-0	R-0											

15	10	9	8	7	6	5	4	3	2	1	0
Reserved		RND32	Reserved		TF	ZI	NI	ZF	NF	LUF	LVF
R-0		R/W-0	R-0		R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

The latched overflow and underflow flags are connected to the peripheral interrupt expansion (PIE) block. This can be useful for debugging any overflow or underflow problems in an application. The LUV and LVF bits will stay set until you take an action that will clear them.

Table 9. Floating-Point Flags

Flag	Name	Used By
LVF	Latched overflow	LVF and LUF flags are set by math instructions. This includes multiply, addition, subtraction, 1/x, 1/sqrt(x)
LUF	Latched underflow	
NF	Negative floating-point	Negative and zero flags are set on: • Moves to floating-point registers • The result of compare, min, max, absolute value and negative value operations
ZF	Zero floating-point	
NI	Negative integer	
ZI	Zero integer	
TF	Test	TESTTF instruction
RND32	Rounding mode	SETFLG instruction, SAVE instruction.
SHDWS	Shadow bit	Set when the SAVE instruction is used to copy the FPU registers to their shadow. Cleared with the RESTORE instruction is used.

Instructions that control program flow are standard C28x instructions. The ST0 register flags determine what these conditional operations do. To instead take action based on floating-point results, you must first copy the relevant flags to the ST0 register. Example 15 shows how this is done. The CMPF32 instruction compares the contents of R1H and R2H and sets the NF and ZF flags in STF appropriately. MOVST0 then copies the specified flags to the ST0 register. The branch fast (BF) instruction then checks flags in ST0 to determine if the loop is repeated or not. Notice that only the flags specified by the MOVST0 are copied to ST0. If this list includes the latched overflow or underflow flags, then the flag is automatically cleared as part of the MOVST0 instruction.

Example 15. Copying FPU Flags to the ST0 Register

```
Loop:
 MOV32  R0H,*XAR4++
 MOV32  R1H,*XAR3++
 CMPF32 R1H, R0H
 MOVST0 ZF, NF
 BF     Loop, GT ; Loop if (R1H > R0H)
```

3.7 The Repeat Block Instruction

The repeat block instruction (RPTB) is new for the C28x FPU. RPTB allows you to repeat a block of code. Any mix of instructions can make up the block of code as long as they are not discontinuities such as branch, call or TRAP type instructions.

The two forms of RPTB are shown in Example 16. An important thing to understand is after the first iteration RPTB does not have any cycle overhead. That is, RPTB takes zero cycles after the first iteration. On the C28x, similar loops were implemented using the BANZ instruction. The BANZ instruction has an overhead of 4 cycles each iteration. Performance of block type algorithms such as the FFT and IIR are greatly improved by the use of the RPTB instruction.

Example 16. Two Forms of the Repeat Block Instruction

```
   ; Repeat block with immediate count
;
  RPTB #label, #RC    ; 1+0 Cycles

; Repeat block with count stored in a register
;
  RPTB #label, loc16  ; 4+0 Cycles
```

The RPTB instruction has its own support register called RB. This register is automatically managed by the hardware when RPTB is executed. You do not need to read or modify this register. The RB register are shown in Figure 6.

Figure 6. Repeat Block Register (RB)

31	30	29 23	22 16
RAS	RA	RSIZE	RE
R/W-0	R/W-0	R/W-0	R/W-0

15 0
RC
R/W-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Example 17 shows a RPTB block. The assembler calculates the size of the block using the label in the RPTB instruction. In this case, VECTOR_MAX_END is the label indicating the end of the block. When the RPTB starts the repeat block active bit (RA) is set to 1 and the repeat block size, end address and count (RSIZE, RE, and RC) are automatically populated. The count specified is the number of times the block is repeated. If RC is zero, then the block will execute once. When the block completes, the RA bit is automatically cleared.

Notes:

- The maximum block size is 127 x 16-bit words.
- The minimum block size depends on the address alignment of the block:
  - For odd alignment, the minimum is 8 words.
  - For even alignment, the minimum is 9 words.
- The assembler will check the size and alignment of your repeat blocks.
- Interrupts are the only discontinuity allowed in a repeat block.

The block size is limited to 127 16-bit words. The minimum block size depends on the address alignment of the RPTB instruction. For odd alignment the minimum is 8 words and for even alignment the minimum is 9 words. The assembler checks the size and alignment of your RPTB blocks.

Example 17. Repeat Block

```
  ; find the largest element and put its address in XAR6
  .align 2
   NOP
   RPTB    VECTOR_MAX_END, AR7   ; RA (repeat active) = 1. AR7 contains RC
   MOVL    ACC,XAR0
   MOV32   R1H,*XAR0++           ; RSIZE (block size)
   MAXF32  R0H,R1H               ; Max: 127 x 16 words
   MOVST0  NF,ZF                 ; Min:   8 words (odd aligned)
   MOVL    XAR6,ACC,LT           ;        9 words (even aligned)
VECTOR_MAX_END:                  ; RE (end address)
                                 ; RA = 0
```

A repeat block is interruptible and allows for nested interrupts. When an interrupt is taken, RA is automatically copied into a shadow (RAS) bit. Likewise, when the interrupt returns, RA is loaded from RAS. Interrupts are the only discontinuities allowed within a repeat block. Instructions such as branch, call, and trap are not allowed. Single repeat instructions like RPT || MACF32 can be used in a repeat block.

In some cases, the RB register must be saved to and restored from the stack in an interrupt service routine. The rules are follows:

- **High-Priority Interrupt Service Routine**

  A high-priority interrupt is defined as one that cannot itself be interrupted. That is nesting interrupts within this routine is not allowed. In a high-priority interrupt, the RB register only needs to be saved to and restored from the stack if the interrupt service routine contains a repeat block. If the interrupt does not have a repeat block, then there is no need to save and restore RB.
- **Low-Priority Interrupt Service Routine**

  A low priority interrupt is any interrupt that can itself be interrupted. That is it allows nested interrupts. In a low-priority interrupt, the RB register must always be saved to and restored from the stack. The save and restore of RB must be done while interrupts are disabled. Failure to save and restore RB will result in corruption of the RAS bit.

Example 18. Handling RB Within an Low-Priority Interrupt

```
  _ISR:                          ; RAS = RA, RA = 0
   ...
   PUSH RB                   ; Save RB
   ...
   CLRC INTM                 ; Enable interrupts after saving RB
   ...
   RPTB  VECTOR_MAX_END, AR7
   MOVL  ACC,XAR0
   ...
   ...
   MOVL    XAR6,ACC,LT
VECTOR_MAX_END:
    ...
   SETC INTM                 ; Disable interrupts
   ...
   POP  RB                   ; Restore RB
   ...
   IRET                      ; RA = RAS, RAS = 0
```

3.8 Moves Between C28x and FPU Registers

Moves between the C28x registers and FPU registers is a fairly infrequent operation. Due to the pipeline, additional alignment is required for these operations. As with any pipeline conflict, the assembler will issue an error if the alignment is not implemented properly.

3.8.1 Copy From C28x Register to FPU Register

The following instructions copy the contents of a C28x register to an FPU register.

- MOV32 RaH,@XARn
- MOV32 RaH,@ACC
- MOV32 RaH,@T
- MOV32 RaH,@P

When one of these instructions is used, 4 delay slots must be inserted before the destination register can be used. This is shown in Example 19

Example 19. Copy From C28x Register to FPU Register

```
MOV32   R0H, @ACC     ; Copy ACC into R0H
NOP
NOP                   ; Wait 4 instructions
NOP
NOP
ADDF32  R2H,R1H,R0H   ; Can use R0H
```

The delay slots in Example 19 cannot be filled with any conversion instruction or the FRAC32 (fractional) instruction. Otherwise any other non-conflicting instruction can be used.

3.8.2 Copy From FPU Register to C28x Register

The following instructions copy the contents of an FPU register into a C28x register:

- MOV32 @XARn,RaH
- MOV32 @ACC,RaH
- MOV32 @XT,RaH
- MOV32 @P,RaH

These instructions require an additional alignment cycle between when RaH is updated and when it can be copied to the C28x register. This delay is in addition to any normal pipeline delay. That is a single cycle instruction requires 1 alignment as shown in Example 20while a 2p instruction will require 2 delays as shown in Example 21

Example 20. Copy From FPU Register to C28x Register Following a Single -Cycle Instruction

```
MINF32 R0H,R1H         ; Single cycle instruction, updates R0H
NOP                    ; 1 Alignment cycle required before copy
MOV32  @ACC,R0H        ; Copy R0H to ACC
```

Example 21. Copy From FPU Register to C28x Register Following a 2p Instruction

```
ADDF32  R0H,R1H,R2H    ; 2p instruction
NOP                    ; Delay for ADDF32 to complete
NOP                    ; R0H Updated. Required alignment cycle before copy
MOV32  @ACC,R0H        ; Copy R0H to ACC
```

4 Interrupt Context Save and Restore

This section describes high- and low-priority interrupt context save and restore and specifying priority in C/C++.

4.1 High-Priority Interrupt Context Save and Restore

The fast interrupt context save and restore for high-priority interrupts on the C28x has been retained for the C28x+FPU. Recall that a high-priority interrupt is defined as an interrupt service routine that does not allow nested interrupts. Example 22 shows a full high-priority interrupt context save for the C28x+FPU. As on the C28x, the following critical registers are automatically saved on an interrupt: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1, PC. The remaining C28x registers are pushed onto the stack if used within the ISR. There are two new instructions in Example 22: RB must be saved if it is used in the interrupt and the FPU registers are copied to their shadow registers using the SAVE instruction. SAVE and RESTORE should only be used in high-priority interrupts. The PUSH RB and SAVE instructions are both single cycle and add little overhead to the C28x context save and restore. You can also set flags with the SAVE instruction.

The context save shown in Example 22 takes 22 cycles. This is the worst case and assumes you save all of the registers.

Example 22. High-Priority Interrupt Service Routine Context Save

```
  _HighestPriorityISR:
   ASP              ; Align stack
   PUSH  RB         ; Save RB if used     <-- New for FPU
   PUSH  AR1H:AR0H  ; Save if used
   PUSH  XAR2
   PUSH  XAR3
   PUSH  XAR4
   PUSH  XAR5
   PUSH  XAR6
   PUSH  XAR7
   PUSH  XT
   SPM   0          ; Set C28 modes
   CLRC  AMODE
   CLRC  PAGE0,OVM
   SAVE  RNDF32=1   ; FPU registers       <-- New for FPU
                    ; set FPU mode
   ...
   ...
;
; 22 Cycles worst case (all registers saved)
```

The high-priority restore code is shown in Example 23. The RESTORE instruction is used to copy the FPU registers from the shadow back to the active register set and POP RB is used to restore the RB register from the stack. The context restore takes 19 cycles worst case.

Example 23. High-Priority Interrupt Service Routine Context Restore

```
   ...
   ...
   RESTORE       ; FPU registers           ; <-- new for FPU
   POP   XT      ; Restore registers
   POP   XAR7
   POP   XAR6
   POP   XAR5
   POP   XAR4
   POP   XAR3
   POP   XAR2
   POP   AR1H:AR0H
   POP   RB       ; Restore RB             ; <-- new for FPU
   NASP           ; Un-align stack
   IRET           ; Return
;
;
; 19 Cycles worst case (all registers restored)
```

4.2 Low-Priority Interrupt Context Save and Restore

Low-priority interrupts are those interrupt routines that allow nested interrupts. In low-priority interrupts, where time is not as critical, save and restore the FPU result and STF registers using the stack as shown in Example 24. This keeps the FPU shadow registers available for your time-critical high-priority interrupts as shown in Section 4.1. In a low-priority interrupt you must also save the RB register. Failure to save RB results in corruption of the RAS (repeat active shadow) bit. The time from when the interrupt is acknowledged by the CPU to the end of the context save is 42 cycles. This is a worst case value and assumes you need to save all of the registers. Interrupts are disabled for 21 cycles of these cycles.

The corresponding context restore is shown in Example 25. The context restore takes 32 cycles worst case and interrupts are disabled for 14 cycles.

Example 24. Low-Priority Interrupt Service Routine Context Save

```
_LowerPriorityISR:
  MOVW   DP,#PIE      ; Set PIE Interrupt Priority
  MOV    AL,@PIEIERn
  OR     IER,#INTn_PRIORITY_MASK
  AND    IER,#IER_PRIORITY_MASK
  MOV    @PIEIERn,#PIEIERn_PRIORITY_MASK
  MOV    @PIEACK,#0xFFFF
  MOV    *SP++,AL
  ASP                ; Align Stack Pointer
  PUSH   RB          ; Save RB                 <-- New for FPU
  CLRC   INTM        ; Enable Interrupts
  PUSH   AR1H:AR0H   ; Save XAR0 to XAR7
  PUSH   XAR2
  PUSH   XAR3
  PUSH   XAR4
  PUSH   XAR5
  PUSH   XAR6
  PUSH   XAR7
  PUSH   XT          ; Save XT
  MOV32  *SP++,STF   ; Save STF                <-- New for FPU
  MOV32  *SP++,R0H   ; Save R0H                <-- New for FPU
  MOV32  *SP++,R1H   ; Save R1H                <-- New for FPU
  MOV32  *SP++,R2H   ; Save R2H                <-- New for FPU
  MOV32  *SP++,R3H   ; Save R3H                <-- New for FPU
  MOV32  *SP++,R4H   ; Save R4H                <-- New for FPU
  MOV32  *SP++,R5H   ; Save R5H                <-- New for FPU
  MOV32  *SP++,R6H   ; Save R6H                <-- New for FPU
  MOV32  *SP++,R7H   ; Save R7H                <-- New for FPU
  SPM    0           ; Set default C28 modes
  CLRC   AMODE
  CLRC   PAGE0,OVM
  SETFLG RNDF32=1    ; Set default FPU modes   <-- New for FPU
  ...
  ...
```

Example 25. Low-Priority Interrupt Service Routine Context Restore

```
...
...
MOV32  R7H,*--SP    ; Restore R7H             <-- New for FPU
MOV32  R6H,*--SP    ; Restore R6H             <-- New for FPU
MOV32  R5H,*--SP    ; Restore R5H             <-- New for FPU
MOV32  R4H,*--SP    ; Restore R4H             <-- New for FPU
MOV32  R3H,*--SP    ; Restore R3H             <-- New for FPU
MOV32  R2H,*--SP    ; Restore R2H             <-- New for FPU
MOV32  R1H,*--SP    ; Restore R1H             <-- New for FPU
MOV32  R0H,*--SP    ; Restore R0H             <-- New for FPU
MOV32  STF,*--SP    ; Restore STF             <-- New for FPU
POP    XT           ; Restore XT
POP    XAR7         ; Restore XAR0 to XAR7
POP    XAR6
POP    XAR5
POP    XAR4
POP    XAR3
POP    XAR2
POP    AR1H:AR0H
MOVW   DP,#PIE
SETC   INTM        ; Disable Interrupts
POP    RB          ; Restore RB              <-- New for FPU
NASP               ; Un-align Stack Pointer
MOV    AL,*--SP
MOV    @PIEIERn,AL ; Restore PIE Interrupt Priority
IRET               ; Return From Interrupt
```

4.3 Specifying Interrupt Priority in C/C++

Two new pragma statements have been added to the compiler to allow you to specify the priority of an interrupt when writing in C or C++ code. In this context, a high priority interupt is defined as one that can not itself be interrupted. A low priority interrupt is defined as one that can be interrupted.

The pragma statements are shown in Example 26.

Example 26. Pragma Statement to Specific Interrupt Priority

```
  //
// Specify a High-Priority Interrupt:
// This interrupt cannot itself be interrupted
//

#pragma INTERRUPT (function_name, HPI)

//
// Specify a Low-Priority Interrupt:
// The user has chosen to re-enable interrupts
// in this routine.  This causes it to be a low-priority
// interrupt
//

#pragma INTERRUPT (function_name, LPI)
```

The main difference between the two is how the repeat block (RB) register and context save are handled. The compiler follows these rules when working with interrupt service routines:

- Interrupts are never enabled by the compiler. It is always up to you to enable interrupts by clearing the INTM bit.
- The compiler always saves and restores RB in a low-priority interrupt.
- The compiler only saves and restores RB in a high-priority interrupt if a repeat block instruction is used within the interrupt service routine.
- The compiler only uses the SAVE and RESTORE functions in high-priority interrupts.
- The compiler will always assume interrupts are low-priority by default. This is compatible with code written for the C28x.

5 Development Tools and Documentation

This section lists the various tools and documentation that will help you develop your floating-point application.

5.1 Instruction Set Reference Guides

- ***TMS320C28x CPU and Instruction Set Reference Guide*** (SPRU430)

This document lists all of the C28x fixed-point instructions, pipeline behavior, registers and interrupt response. Everything in this document applies to both the C28x fixed-point as well as C28x+FPU controllers

- ***TMS320C28x Floating Point Unit and Instruction Set Reference Guide*** (SPRUEO2)

The FPU instruction set reference guide lists all of the instructions that have been added to support single-precision floating-point operations. This includes all the FPU instructions such as move, math and conversion instructions as well as their pipeline behavior. This users guide should be considered a supplement to the CPU guide (SPRU430).

5.2 Development Tools and Software

- **Code Composer Studio and Codegen Tools**

Code Composer Studio v3.3 and later support the C28x+FPU. Make sure you have installed the latest service release. As of this date, SR12 is the latest version. The service release contains the flash plugin as well as gel files for the floating-point devices and bug fixes. The compiler is a seperate release. This was done so customers using older compilers would not overwrite the compiler they are using when evaluating the floating-point unit. You need to install version compiler version 5.0.0 or later to build code for the floating-point unit. The 5.0.0 compiler can build both fixed-point as well as floating-point code. To tell the compiler to build floating-point, use the compiler switches: -v28 --float_support=fpu32. In CCS this option is in the build options under compiler-> advanced: floating point support. Without the float_support flag, the tools will build fixed-point code.

- **Standard Run-time Support Libraries**

When building for floating-point you need to make sure all associated libraries have also been built for floating-point. The standard run-time support (RTS) libaries built for floating-point included with the compiler have fpu32 in their name. For example rts2800_fpu32.lib and rts2800_fpu_eh.lib have been built for the floating-point unit. The "eh" version has exception handling for C++ code. Using the fixed-point RTS libraries will result in the linker issuing an error for incompatible object files.

- *C28x FPU Fast RTS Library* (SPRC664)

The fast RTS library contains hand-coded optimized math routines such as division, square root, atan2, sin and cos. This library can be linked into your project before the standard runtime support library to give your application a performance boost. As an example, the standard RTS library uses a polynomial expansion to calculate the sin function. The fast RTS library, however, uses a math look-up table in the boot ROM of the device. Using this look-up table method results in approximately a 20 cycle savings over the standard RTS calculation.

5.3 Converting From IQmath to Floating-Point

Texas Instruments TMS320C28x IQmath Library is collection of highly optimized and high precision mathematical Function Library for C/C++ programmers to seamlessly port the floating-point algorithm into fixed point code devices. Likewise if code has been written in IQmath it can quickly be converted to floating-point. Therefore there are advantages to using the IQmath library for all projects. To convert a project that has been written in IQmath to native floating point follow these steps:

- In the IQmath header file, select FLOAT_MATH. The header file will convert all IQmath function calls to their floating-point equivalent.
- When writing a floating-point number into a device register, you need to convert the floating-point number to an integer as shown in Example 27. Likewise when reading a value from a register it will need to be converted to float. In both cases, this is done by multiplying the number by a conversion factor. For example to convert a floating-point number to IQ15, multiply by 32768.0. Likewise, to convert from an IQ15 value to a floating-point value, multiply by 1/32768.0 or 0.000030518.0. One thing to note: The integer range is restricted to 24-bits for a 32-bit floating-point value.

Example 27. Converting Register Accesses From IQmath to Floating-point

```
   //
// Example:
// Convert from float to IQ15

//
// If MATH_TYPE == IQ_MATH
// Use the IQmath conversion function
//

#if MATH_TYPE == IQ_MATH
   PwmReg = (int16)_IQtoIQ15(Var1);

//
// If MATH_TYPE == FLOAT_MATH
// Scale by 2^15 = 32768.0
//

#else // MATH_TYPE is FLOAT_MATH
   PwmReg = (int16)(32768.0L*Var1);
#endif
```

- Take advantage of the Delfino on-chip floating point unit by doing the following:
  - Use C28x codegen tools version 5.0.2 or later.
  - Tell the compiler it can generate native C28x floating-point code. To do this, use the –v28 --float_support=fpu32 compiler switches. In Code Composer Studio V3.3 the float_support switch is on the advanced tab of the compiler options.
  - Use the correct run-time support library for native 32-bit floating-point. For C code this is rts2800_fpu32.lib. For C++ code with exception handling, use rts2800_fpu32_eh.lib.
  - Use the C28x FPU Fast RTS library (SPRC664) to get a performance boost from math functions such as sin, cos, div, sqrt, and atan. The Fast RTS library should be linked in before the normal run-time support library as described in the documentation.

6 Comparing the C28x+FPU to the Control Law Accelerator

The C2000 Control Law Accelerator (CLA) was introduced on C2000 Piccolo microcontrollers. The CLA is a 32-bit floating-point math accelerator that runs in parallel with the main CPU. The CLA is completely programmable in assembly. The differences include, but are not limited to, those shown in Table 10.

The biggest thing to keep in mind is the CLA is independent from the main CPU where as the FPU unit is a superset on top of the C28x fixed-point CPU and can not be thought of as independent. The CLA has its own bus structure, register set, pipeline and processing unit. In addition the CLA is interrupt driven and can access specific peripheral registers directly. This makes it ideal for handling time-critical control loops but it can also be used for filtering or math algorithms.

Table 10. FPU Compared to CLA

	CLA	C28x + FPU
Execution	In parallel with main CPU.	FPU instructions do not execute in parallel with fixed-point.
Result Registers	4 (MR0 - MR3)	8 (R0H - R7H)
Pipeline	Independent 8 stage	Shares fetch and decode with fixed-point
Memory Access	Specific memory blocks and message RAM	Entire device
Register Access	Specific registers	Entire device
Repeat Instructions	None	Repeat MACF32 & repeat block (RPTB)
Flow Control	Native call/return/branch	Uses C28x fixed-point flow control
Math and Conversion	Single cycle	2p cycles
Nested Interrupts	No, no stack pointer	Supported
Addressing Modes	Direct and indirect	All C28x addressing modes
Programming	Assembly	C, C++, Assembly

7 References

- *TMS320C28x Floating Point Unit and Instruction Set Reference Guide* (SPRUEO2)
- *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430)
- C28x FPU Fast RTS Library (SPRC664)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265