TMS320C28x Floating Point Unit and Instruction Set

Reference Guide

Literature Number: SPRUEO2A
June 2007–Revised August 2008

Contents

Preface **5**

1 Introduction **7**

1.1 Introduction to the Central Processing Unit (CPU) 8

1.2 Compatibility with the C28x Fixed-Point CPU 8

1.2.1 Floating-Point Code Development 9

1.3 Components of the C28x plus Floating-Point CPU 9

1.3.1 Emulation Logic 10

1.3.2 Memory Map 10

1.3.3 On-Chip Program and Data 10

1.3.4 CPU Interrupt Vectors 10

1.4 Memory Interface 10

1.4.1 Address and Data Buses 11

1.4.2 Alignment of 32-Bit Accesses to Even Addresses 11

2 CPU Register Set **13**

2.1 CPU Registers 14

2.1.1 Floating-Point Status Register (STF) 16

2.1.2 Repeat Block Register (RB) 18

3 Pipeline **21**

3.1 Pipeline Overview 22

3.2 General Guidelines for Floating-Point Pipeline Alignment 22

3.3 Moves from FPU Registers to C28x Registers 23

3.4 Moves from C28x Registers to FPU Registers 23

3.5 Parallel Instructions 24

3.6 Invalid Delay Instructions 24

3.7 Optimizing the Pipeline 27

4 Instruction Set **29**

4.1 Instruction Descriptions 30

4.2 Instructions 32

A Revision History **137**

A.1 Changes 137

List of Figures

1-1 FPU Functional Block Diagram 8
2-1 C28x With Floating-Point Registers 14
2-2 Floating-point Unit Status Register (STF) 16
2-3 Repeat Block Register (RB) 18
3-1 FPU Pipeline 22

List of Tables

2-1 28x Plus Floating-Point CPU Register Summary 15
2-2 Floating-point Unit Status (STF) Register Field Descriptions 16
2-3 Repeat Block (RB) Register Field Descriptions 18
4-1 Operand Nomenclature 30
4-2 Summary of Instructions 32
A-1 Technical Changes Made in This Revision 137

Read This First

This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C28x floating-point DSP.

About This Manual

The TMS320C2000™ digital signal processor (DSP) platform is part of the TMS320™ DSP family.

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h or with a leading 0x. For example, the following number is 40 hexadecimal (decimal 64): 40h or 0x40.
- Registers in this document are shown in figures and described in tables.
  - Each register figure shows a rectangle divided into fields that represent the fields of the register. Each field is labeled with its bit name, its beginning and ending bit numbers above, and its read/write properties below. A legend explains the notation used for the properties.
  - Reserved bits in a register figure designate a bit that is used for future device expansion.

Related Documentation

The following books describe the TMS320x28x and related support tools that are available on the TI website:

Data Manual and Errata—

SPRS439— TMS320F28335, TMS320F28334, TMS320F28332, TMS320F28235, TMS320F28234, TMS320F28232 Digital Signal Controllers (DSCs) Data Manual contains the pinout, signal descriptions, as well as electrical and timing specifications for the F2833x/2823x devices.

SPRZ272— TMS320F28335, F28334, F28332, TMS320F28235, F28234, F28232 Digital Signal Controllers (DSCs) Silicon Errata describes the advisories and usage notes for different versions of silicon.

CPU User's Guides—

SPRU430— TMS320C28x DSP CPU and Instruction Set Reference Guide describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

SPRUEO2— TMS320C28x Floating Point Unit and Instruction Set Reference Guide describes the floating-point unit and includes the instructions for the FPU.

Peripheral Guides—

SPRU566— TMS320x28xx, 28xxx Peripheral Reference Guide describes the peripheral reference guides of the 28x digital signal processors (DSPs).

SPRUFB0— TMS320x2833x, 2823x System Control and Interrupts Reference Guide describes the various interrupts and system control features of the 2833x digital signal controllers (DSCs).

SPRU812— TMS320x2833x, 2823x Analog-to-Digital Converter (ADC) Reference Guide describes how to configure and use the on-chip ADC module, which is a 12-bit pipelined ADC.

SPRU949— TMS320x2833x, 2823x External Interface (XINTF) User's Guide describes the XINTF, which is a nonmultiplexed asynchronous bus, as it is used on the 2833x devices.

SPRU963— TMS320x2833x, TMS320x2823x Boot ROM User's Guide describes the purpose and features of the bootloader (factory-programmed boot-loading software) and provides examples of code. It also describes other contents of the device on-chip boot ROM and identifies where all of the information is located within that memory.

SPRUFB7— TMS320x2833x, 2823x Multichannel Buffered Serial Port (McBSP) User's Guide describes the McBSP available on the F2833x devices. The McBSPs allow direct interface between a DSP and other devices in a system.

SPRUFB8— TMS320x2833x, 2823x Direct Memory Access (DMA) Reference Guide describes the DMA on the 2833x devices.

SPRUG04— TMS320x2833x, 2823x Enhanced Pulse Width Modulator (ePWM) Module Reference Guide describes the main areas of the enhanced pulse width modulator that include digital motor control, switch mode power supply control, UPS (uninterruptible power supplies), and other forms of power conversion.

SPRUG02— TMS320x2833x, 2823x High-Resolution Pulse Width Modulator (HRPWM) describes the operation of the high-resolution extension to the pulse width modulator (HRPWM).

SPRUFG4— TMS320x2833x, 2823x Enhanced Capture (eCAP) Module Reference Guide describes the enhanced capture module. It includes the module description and registers.

SPRUG05— TMS320x2833x, 2823x Enhanced Quadrature Encoder Pulse (eQEP) Reference Guide describes the eQEP module, which is used for interfacing with a linear or rotary incremental encoder to get position, direction, and speed information from a rotating machine in high performance motion and position control systems. It includes the module description and registers.

SPRUEU1— TMS320x2833x, 2823x Enhanced Controller Area Network (eCAN) Reference Guide describes the eCAN that uses established protocol to communicate serially with other controllers in electrically noisy environments.

SPRUFZ5— TMS320F2833x, 2823x Serial Communication Interface (SCI) Reference Guide describes the SCI, which is a two-wire asynchronous serial port, commonly known as a UART. The SCI modules support digital communications between the CPU and other asynchronous peripherals that use the standard non-return-to-zero (NRZ) format.

SPRUEU3— TMS320x2833x, 2823x Serial Peripheral Interface (SPI) Reference Guide describes the SPI - a high-speed synchronous serial input/output (I/O) port - that allows a serial bit stream of programmed length (one to sixteen bits) to be shifted into and out of the device at a programmed bit-transfer rate.

SPRUG03— TMS320x2833x, 2823x Inter-Integrated Circuit (I2C) Reference Guide describes the features and operation of the inter-integrated circuit (I2C) module.

Tools Guides—

SPRU513— TMS320C28x Assembly Language Tools User's Guide describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C28x device.

SPRU514— TMS320C28x Optimizing C Compiler User's Guide describes the TMS320C28x™ C/C++ compiler. This compiler accepts ANSI standard C/C++ source code and produces TMS320 DSP assembly language source code for the TMS320C28x device.

SPRU608— The TMS320C28x Instruction Set Simulator Technical Overview describes the simulator, available within the Code Composer Studio for TMS320C2000 IDE, that simulates the instruction set of the C28x™ core.

SPRU625— TMS320C28x DSP/BIOS Application Programming Interface (API) Reference Guide describes development using DSP/BIOS.

Introduction

The TMS320C2000™ DSP family consists of fixed-point and floating-point digital signal controllers (DSCs). TMS320C2000™ Digital Signal Controllers combine control peripheral integration and ease of use of a microcontroller (MCU) with the processing power and C efficiency of TI's leading DSP technology. This chapter provides an overview of the architectural structure and components of the C28x plus floating-point unit CPU.

Topic **Page**

1.1 Introduction to the Central Processing Unit (CPU) 8
1.2 Compatibility with the C28x Fixed-Point CPU 8
1.3 Components of the C28x plus Floating-Point CPU 9
1.4 Memory Interface 10

1.1 Introduction to the Central Processing Unit (CPU)

The C28x plus floating-point (C28x+FPU) processor extends the capabilities of the C28x fixed-point CPU by adding registers and instructions to support IEEE single-precision floating point operations. This device draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The DSC 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 (usable in Von Neumann mode). 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.

Throughout this document the following notations are used:

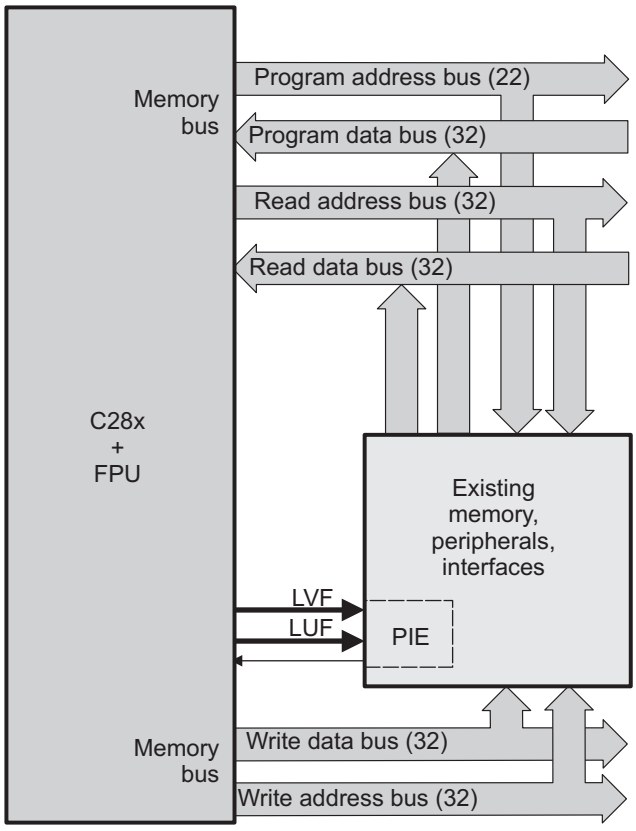
- C28x refers to the C28x fixed-point CPU.
- C28x plus Floating-Point and C28x+FPU both refer to the C28x CPU with enhancements to support IEEE single-precision floating-point operations.

1.2 Compatibility with the C28x Fixed-Point CPU

No changes have been made to the C28x base set of instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x CPU are completely compatible with the C28x+FPU and all of the features of the C28x documented in *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430) apply to the C28x+FPU.

Figure 1-1 shows basic functions of the FPU.

Figure 1-1. FPU Functional Block Diagram



1.2.1 Floating-Point Code Development

When developing C28x floating-point code use Code Composer Studio 3.3, or later, with at least service release 8. The C28x compiler V5.0, or later, is also required to generate C28x native floating-point opcodes. This compiler is available via Code Composer Studio update advisor as a seperate download. V5.0 can generate both fixed-point as well as floating-point code. To build floating-point code use the compiler switches:-v28 and - -float_support = fpu32. In Code Composer Studio 3.3 the float_support option is in the build options under compiler-> advanced: floating point support. Without the float_support flag, or with float_support = none, the compiler will generate fixed-point code.

When building for C28x floating-point 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 in a floating-point project will result in the linker issuing an error for incompatible object files.

To improve performance of native floating-point projects, consider using the *C28x FPU Fast RTS Library* (SPRC664). This 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.

1.3 Components of the C28x plus Floating-Point CPU

The C28x+FPU contains:

- A central processing unit for generating data and program-memory addresses; decoding and executing instructions; performing arithmetic, logical, and shift operations; and controlling data transfers among CPU registers, data memory, and program memory
- A floating-point unit for IEEE single-precision floating point operations.
- Emulation logic for monitoring and controlling various parts and functions of the device and for testing device operation. This logic is identical to that on the C28x fixed-point CPU.
- Signals for interfacing with memory and peripherals, clocking and controlling the CPU and the emulation logic, showing the status of the CPU and the emulation logic, and using interrupts. This logic is identical to the C28x fixed-point CPU.

Some features of the C28x+FPU central processing unit are:

- Fixed-Point instructions are pipeline protected. This pipeline for fixed-point instructions is identical to that on the C28x fixed-point CPU. The CPU implements an 8-phase pipeline that prevents a write to and a read from the same location from occurring out of order. See Figure 3-1
- Some floating-point instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots.
- Independent register space. These registers function as system-control registers, math registers, and data pointers. The system-control registers are accessed by special instructions.
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Floating point unit (FPU). The 32-bit FPU performs IEEE single-precision floating-point operations.
- Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.
- Barrel shifter. This shifter performs all left and right shifts of fixed-point data. It can shift data to the left by up to 16 bits and to the right by up to 16 bits.
- Fixed-Point Multiplier. The multiplier performs 32-bit $\times$ 32-bit 2s-complement multiplication with a 64-bit result. The multiplication can be performed with two signed numbers, two unsigned numbers, or one signed number and one unsigned number.

1.3.1 Emulation Logic

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features. For more details about these features, refer to the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430):

- Debug-and-test direct memory access (DT-DMA). A debug host can gain direct access to the content of registers and memory by taking control of the memory interface during unused cycles of the instruction pipeline.
- A counter for performance benchmarking.
- Multiple debug events. Any of the following debug events can cause a break in program execution:
  - A breakpoint initiated by the ESTOP0 or ESTOP1 instruction.
  - An access to a specified program-space or data-space location.

  When a debug event causes the C28x to enter the debug-halt state, the event is called a break event.
- Real-time mode of operation.

1.3.2 Memory Map

Like the C28x, the C28x+FPU uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all C28x+FPU designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the data sheet for your device.

1.3.3 On-Chip Program and Data

All C28x+FPU based devices contain at least two blocks of single access on-chip memory referred to as M0 and M1. Each of these blocks is 1K words in size. M0 is mapped at addresses 0x0000 – 0x03FF and M1 is mapped at addresses 0x0400 – 0x07FF. Like all other memory blocks on the C28x+FPU devices, M0 and M1 are mapped to both program and data space. Therefore, you can use M0 and M1 to execute code or for data variables. At reset, the stack pointer is set to the top of block M1. Depending on the device, it may also have additional random-access memory (RAM), read-only memory (ROM), external interface zones, or flash memory.

1.3.4 CPU Interrupt Vectors

The C28x+FPU interrupt vectors are identical to those on the C28x CPU. Sixty-four addresses in program space are set aside for a table of 32 CPU interrupt vectors. The CPU vectors can be mapped to the top or bottom of program space by way of the VMAP bit. For more information about the CPU vectors, see *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430). For devices with a peripheral interrupt expansion (PIE) block, the interrupt vectors will reside in the PIE vector table and this memory can be used as program memory.

1.4 Memory Interface

The C28x+FPU memory interface is identical to that on the C28x. The C28x+FPU memory map is accessible outside the CPU by the memory interface, which connects the CPU logic to memories, peripherals, or other interfaces. The memory interface includes separate buses for program space and data space. This means an instruction can be fetched from program memory while data memory is being accessed. The interface also includes signals that indicate the type of read or write being requested by the CPU. These signals can select a specified memory block or peripheral for a given bus transaction. In addition to 16-bit and 32-bit accesses, the C28x+FPU supports special byte-access instructions that can access the least significant byte (LSByte) or most significant byte (MSByte) of an addressed word. Strobe signals indicate when such an access is occurring on a data bus.

1.4.1 Address and Data Buses

Like the C28x, the memory interface has three address buses:

- **PAB: Program address bus**
  The PAB carries addresses for reads and writes from program space. PAB is a 22-bit bus.
- **DRAB: Data-read address bus**
  The 32-bit DRAB carries addresses for reads from data space.
- **DWAB: Data-write address bus**
  The 32-bit DWAB carries addresses for writes to data space.

The memory interface also has three data buses:

- **PRDB: Program-read data bus**
  The PRDB carries instructions during reads from program space. PRDB is a 32-bit bus.
- **DRDB: Data-read data bus**
  The DRDB carries data during reads from data space. DRDB is a 32-bit bus.
- **DWDB: Data-/Program-write data bus**
  The 32-bit DWDB carries data during writes to data space or program space.

A program-space read and a program-space write cannot happen simultaneously because both use the PAB. Similarly, a program-space write and a data-space write cannot happen simultaneously because both use the DWDB. Transactions that use different buses can happen simultaneously. For example, the CPU can read from program space (using PAB and PRDB), read from data space (using DRAB and DRDB), and write to data space (using DWAB and DWDB) at the same time. This behavior is identical to the C28x CPU.

1.4.2 Alignment of 32-Bit Accesses to Even Addresses

The C28x+FPU CPU expects memory wrappers or peripheral-interface logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CPU will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

Most instruction fetches from program space are performed as 32-bit read operations and are aligned accordingly. However, alignment of instruction fetches are effectively invisible to a programmer. When instructions are stored to program space, they do not have to be aligned to even addresses. Instruction boundaries are decoded within the CPU.

You need to be concerned with alignment when using instructions that perform 32-bit reads from or writes to data space.

CPU Register Set

The C28x+FPU architecture is the same as the C28x CPU with an extended register and instruction set to support IEEE single-precision floating point operations. This section describes the extensions to the C28x architecture.

Topic **Page**

2.1 CPU Registers 14

2.1 CPU Registers

Devices with the C28x+FPU include the standard C28x register set plus an additional set of floating-point unit registers. The additional floating-point unit registers are the following:

- Eight floating-point result registers, RnH (where n = 0 - 7)
- Floating-point Status Register (STF)
- Repeat Block Register (RB)

All of the floating-point registers except the repeat block register are shadowed. This shadowing can be used in high priority interrupts for fast context save and restore of the floating-point registers.

Figure 2-1 shows a diagram of both register sets and Table 2-1 shows a register summary. For information on the standard C28x register set, see the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430).

Figure 2-1. C28x With Floating-Point Registers

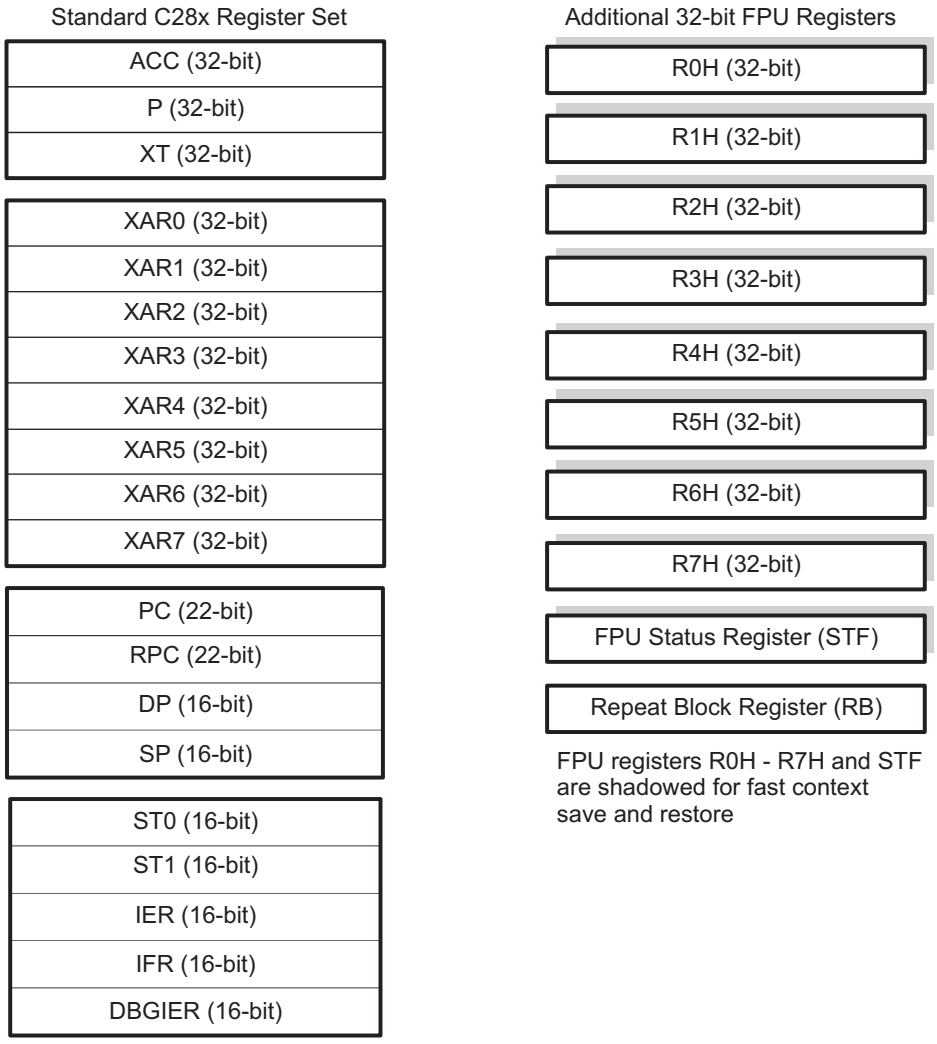


Table 2-1. 28x Plus Floating-Point CPU Register Summary

Register	C28x CPU	C28x+FPU	Size	Description	Value After Reset
ACC	Yes	Yes	32 bits	Accumulator	0x00000000
AH	Yes	Yes	16 bits	High half of ACC	0x0000
AL	Yes	Yes	16 bits	Low half of ACC	0x0000
XAR0	Yes	Yes	16 bits	Auxiliary register 0	0x00000000
XAR1	Yes	Yes	32 bits	Auxiliary register 1	0x00000000
XAR2	Yes	Yes	32 bits	Auxiliary register 2	0x00000000
XAR3	Yes	Yes	32 bits	Auxiliary register 3	0x00000000
XAR4	Yes	Yes	32 bits	Auxiliary register 4	0x00000000
XAR5	Yes	Yes	32 bits	Auxiliary register 5	0x00000000
XAR6	Yes	Yes	32 bits	Auxiliary register 6	0x00000000
XAR7	Yes	Yes	32 bits	Auxiliary register 7	0x00000000
AR0	Yes	Yes	16 bits	Low half of XAR0	0x0000
AR1	Yes	Yes	16 bits	Low half of XAR1	0x0000
AR2	Yes	Yes	16 bits	Low half of XAR2	0x0000
AR3	Yes	Yes	16 bits	Low half of XAR3	0x0000
AR4	Yes	Yes	16 bits	Low half of XAR4	0x0000
AR5	Yes	Yes	16 bits	Low half of XAR5	0x0000
AR6	Yes	Yes	16 bits	Low half of XAR6	0x0000
AR7	Yes	Yes	16 bits	Low half of XAR7	0x0000
DP	Yes	Yes	16 bits	Data-page pointer	0x0000
IFR	Yes	Yes	16 bits	Interrupt flag register	0x0000
IER	Yes	Yes	16 bits	Interrupt enable register	0x0000
DBGIER	Yes	Yes	16 bits	Debug interrupt enable register	0x0000
P	Yes	Yes	32 bits	Product register	0x00000000
PH	Yes	Yes	16 bits	High half of P	0x0000
PL	Yes	Yes	16 bits	Low half of P	0x0000
PC	Yes	Yes	22 bits	Program counter	0x3FFFC0
RPC	Yes	Yes	22 bits	Return program counter	0x00000000
SP	Yes	Yes	16 bits	Stack pointer	0x0400
ST0	Yes	Yes	16 bits	Status register 0	0x0000
ST1	Yes	Yes	16 bits	Status register 1	0x080B(1)
XT	Yes	Yes	32 bits	Multiplicand register	0x00000000
T	Yes	Yes	16 bits	High half of XT	0x0000
TL	Yes	Yes	16 bits	Low half of XT	0x0000
ROH	No	Yes	32 bits	Floating-point result register 0	0.0
R1H	No	Yes	32 bits	Floating-point result register 1	0.0
R2H	No	Yes	32 bits	Floating-point result register 2	0.0
R3H	No	Yes	32 bits	Floating-point result register 3	0.0
R4H	No	Yes	32 bits	Floating-point result register 4	0.0
R5H	No	Yes	32 bits	Floating-point result register 5	0.0
R6H	No	Yes	32 bits	Floating-point result register 6	0.0
R7H	No	Yes	32 bits	Floating-point result register 7	0.0
STF	No	Yes	32 bits	Floating-point status register	0x00000000
RB	No	Yes	32 bits	Repeat block register	0x00000000

(1) Reset value shown is for devices without the VMAP signal and MOM1MAP signal pinned out. On these devices both of these signals are tied high internal to the device.

2.1.1 Floating-Point Status Register (STF)

The floating-point status register (STF) reflects the results of floating-point operations. There are three basic rules for floating point operation flags:

1. Zero and negative flags are set based on moves to registers.
2. Zero and negative flags are set based on the result of compare, minimum, maximum, negative and absolute value operations.
3. Overflow and underflow flags are set by math instructions such as multiply, add, subtract and 1/x. These flags may also be connected to the peripheral interrupt expansion (PIE) block on your device. This can be useful for debugging underflow and overflow conditions within an application.

As on the C28x, program flow is controlled by C28x instructions that read status flags in the status register 0 (ST0) . If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into ST0 flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVST0 FLAG instruction is used to load the current value of specified STF flags into the respective bits of ST0. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

Example 2-1. Moving STF Flags to the ST0 Register

```
Loop:
   MOV32  R0H,*XAR4++
   MOV32  R1H,*XAR3++
   CMPF32 R1H, R0H
   MOVST0 ZF, NF         ; Move ZF and NF to ST0
   BF     Loop, GT       ; Loop if (R1H > R0H)
```

Figure 2-2. 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

Table 2-2. Floating-point Unit Status (STF) Register Field Descriptions

Bits	Field	Value	Description
31	SHDWS		Shadow Mode Status Bit
		0	This bit is forced to 0 by the RESTORE instruction.
		1	This bit is set to 1 by the SAVE instruction.
			This bit is not affected by loading the status register either from memory or from the shadow values.
30 - 10	Reserved	0	Reserved for future use
9	RND32		Round 32-bit Floating-Point Mode
		0	If this bit is zero, the MPYF32, ADDF32 and SUBF32 instructions will round to zero (truncate).
		1	If this bit is one, the MPYF32, ADDF32 and SUBF32 instructions will round to the nearest even value.
8 - 7	Reserved	0	Reserved for future use

Table 2-2. Floating-point Unit Status (STF) Register Field Descriptions (continued)

Bits	Field	Value	Description
6	TF		Test Flag
			The TESTTF instruction can modify this flag based on the condition tested. The SETFLG and SAVE instructions can also be used to modify this flag.
		0	The condition tested with the TESTTF instruction is false.
		1	The condition tested with the TESTTF instruction is true.
5	ZI		Zero Integer Flag
			The following instructions modify this flag based on the integer value stored in the destination register: MOV32, MOVD32, MOVDD32 The SETFLG and SAVE instructions can also be used to modify this flag.
		0	The integer value is not zero.
		1	The integer value is zero.
4	NI		Negative Integer Flag
			The following instructions modify this flag based on the integer value stored in the destination register: MOV32, MOVD32, MOVDD32 The SETFLG and SAVE instructions can also be used to modify this flag.
		0	The integer value is not negative.
		1	The integer value is negative.
3	ZF		Zero Floating-Point Flag (1)(2)
			The following instructions modify this flag based on the floating-point value stored in the destination register: MOV32, MOVD32, MOVDD32, ABSF32, NEGF32 The CMPF32, MAXF32, and MINF32 instructions modify this flag based on the result of the operation. The SETFLG and SAVE instructions can also be used to modify this flag
		0	The floating-point value is not zero.
		1	The floating-point value is zero.
2	NF		Negative Floating-Point Flag (1)(2)
			The following instructions modify this flag based on the floating-point value stored in the destination register: MOV32, MOVD32, MOVDD32, ABSF32, NEGF32 The CMPF32, MAXF32, and MINF32 instructions modify this flag based on the result of the operation. The SETFLG and SAVE instructions can also be used to modify this flag.
		0	The floating-point value is not negative.
		1	The floating-point value is negative.
1	LUF		Latched Underflow Floating-Point Flag
			The following instructions will set this flag to 1 if an underflow occurs: MPYF32, ADDF32, SUBF32, MACF32, EINVF32, EISQRTF32
		0	An underflow condition has not been latched. If the MOVST0 instruction is used to copy this bit to ST0, then LUF will be cleared.
		1	An underflow condition has been latched.
0	LVF		Latched Overflow Floating-Point Flag
			The following instructions will set this flag to 1 if an overflow occurs: MPYF32, ADDF32, SUBF32, MACF32, EINVF32, EISQRTF32
		0	An overflow condition has not been latched. If the MOVST0 instruction is used to copy this bit to ST0, then LVF will be cleared.
		1	An overflow condition has been latched.

(1) A negative zero floating-point value is treated as a positive zero value when configuring the ZF and NF flags.
(2) A DeNorm floating-point value is treated as a positive zero value when configuring the ZF and NF flags.

2.1.2 Repeat Block Register (RB)

The repeat block instruction (RPTB) is a new instruction for C28x+FPU. This instruction allows you to repeat a block of code as shown in Example 2-2.

Example 2-2. The Repeat Block (RPTB) Instruction uses the RB Register

```
; find the largest element and put its address in XAR6
   MOV32   R0H, *XAR0++;
  .align 2                      ; Aligns the next instruction to an even address

   NOP                          ; Makes RPTB odd aligned - required for a block size of 8
   RPTB    VECTOR_MAX_END, AR7  ; RA is set to 1
   MOVL    ACC,XAR0
   MOV32   R1H,*XAR0++          ; RSIZE reflects the size of the RPTB block
   MAXF32  R0H,R1H              ; in this case the block size is 8
   MOVST0  NF,ZF
   MOVL    XAR6,ACC,LT
VECTOR_MAX_END:                 ; RE indicates the end address. RA is cleared
```

The C28x_FPU hardware automatically populates the RB register based on the execution of a RPTB instruction. This register is not normally read by the application and does not accept debugger writes.

Figure 2-3. Repeat Block Register (RB)

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

15 0
RC
R-0

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

Table 2-3. Repeat Block (RB) Register Field Descriptions

Bits	Field	Value	Description
31	RAS		Repeat Block Active Shadow Bit
			When an interrupt occurs the repeat active, RA, bit is copied to the RAS bit and the RA bit is cleared. When an interrupt return instruction occurs, the RAS bit is copied to the RA bit and RAS is cleared.
		0	A repeat block was not active when the interrupt was taken.
		1	A repeat block was active when the interrupt was taken.
30	RA		Repeat Block Active Bit
		0	This bit is cleared when the repeat counter, RC, reaches zero.
			When an interrupt occurs the RA bit is copied to the repeat active shadow, RAS, bit and RA is cleared. When an interrupt return, IRET, instruction is executed, the RAS bit is copied to the RA bit and RAS is cleared.
		1	This bit is set when the RPTB instruction is executed to indicate that a RPTB is currently active.
29-23	RSIZE		Repeat Block Size
			This 7-bit value specifies the number of 16-bit words within the repeat block. This field is initialized when the RPTB instruction is executed. The value is calculated by the assembler and inserted into the RPTB instruction's RSIZE opcode field.
		0-7	Illegal block size.
		8/9-0x7F	A RPTB block that starts at an even address must include at least 9 16-bit words and a block that starts at an odd address must include at least 8 16-bit words. The maximum block size is 127 16-bit words. The codegen assembler will check for proper block size and alignment.

Table 2-3. Repeat Block (RB) Register Field Descriptions (continued)

Bits	Field	Value	Description
22-16	RE		Repeat Block End Address This 7-bit value specifies the end address location of the repeat block. The RE value is calculated by hardware based on the RSIZE field and the PC value when the RPTB instruction is executed. RE = lower 7 bits of (PC + 1 + RSIZE)
15-0	RC		Repeat Count
		0	The block will not be repeated; it will be executed only once. In this case the repeat active, RA, bit will not be set.
		1-0xFFFF	This 16-bit value determines how many times the block will repeat. The counter is initialized when the RPTB instruction is executed and is decremented when the PC reaches the end of the block. When the counter reaches zero, the repeat active bit is cleared and the block will be executed one more time. Therefore the total number of times the block is executed is RC+1.

Pipeline

The pipeline flow for C28x instructions is identical to that of the C28x CPU described in *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430). Some floating-point instructions, however, use additional execution phases and thus require a delay to allow the operation to complete. This pipeline alignment is achieved by inserting NOPs or non-conflicting instructions when required. Software control of delay slots allows you to improve performance of an application by taking advantage of the delay slots and filling them with non-conflicting instructions. This section describes the key characteristics of the pipeline with regards to floating-point instructions. The rules for avoiding pipeline conflicts are small in number and simple to follow and the C28x+FPU assembler will help you by issuing errors for conflicts.

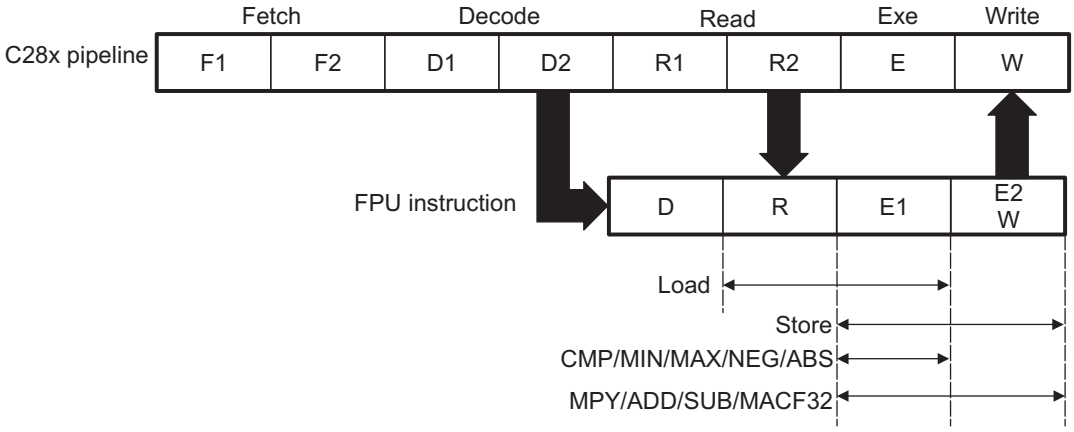
Topic **Page**

3.1 Pipeline Overview 22
3.2 General Guidelines for Floating-Point Pipeline Alignment 22
3.3 Moves from FPU Registers to C28x Registers 23
3.4 Moves from C28x Registers to FPU Registers 23
3.5 Parallel Instructions 24
3.6 Invalid Delay Instructions 24
3.7 Optimizing the Pipeline 27

3.1 Pipeline Overview

The C28x FPU pipeline is identical to the C28x pipeline for all standard C28x instructions. In the decode2 stage (D2), it is determined if an instruction is a C28x instruction or a floating-point unit instruction. The pipeline flow is shown in Figure 3-1. Notice that stalls due to normal C28x pipeline stalls (D2) and memory waitstates (R2 and W) will also stall any C28x FPU instruction. Most C28x FPU instructions are single cycle and will complete in the FPU E1 or W stage which aligns to the C28x pipeline. Some instructions will take an additional execute cycle (E2). For these instructions you must wait a cycle for the result from the instruction to be available. The rest of this section will describe when delay cycles are required. Keep in mind that the assembly tools for the C28x+FPU will issue an error if a delay slot has not been handled correctly.

Figure 3-1. FPU Pipeline



3.2 General Guidelines for Floating-Point Pipeline Alignment

While the C28x+FPU assembler will issue errors for pipeline conflicts, you may still find it useful to understand when software delays are required. This section describes three guidelines you can follow when writing C28x+FPU assembly code.

Floating-point instructions that require delay slots have a 'p' after their cycle count. For example '2p' stands for 2 pipelined cycles. This means that an instruction can be started every cycle, but the result of the instruction will only be valid one instruction later.

There are three general guidelines to determine if an instruction needs a delay slot:

1. Floating-point math operations (multiply, addition, subtraction, 1/x and MAC) require 1 delay slot.
2. Conversion instructions between integer and floating-point formats require 1 delay slot.
3. Everything else does not require a delay slot. This includes minimum, maximum, compare, load, store, negative and absolute value instructions.

There are two exceptions to these rules. First, moves between the CPU and FPU registers require special pipeline alignment that is described later in this section. These operations are typically infrequent. Second, the MACF32 R7H, R3H, mem32, *XAR7 instruction has special requirements that make it easier to use. Refer to the MACF32 instruction description for details.

An example of the 32-bit ADDF32 instruction is shown in Example 3-1. ADDF32 is a 2p instruction and therefore requires one delay slot. The destination register for the operation, R0H, will be updated one cycle after the instruction for a total of 2 cycles. Therefore, a NOP or instruction that does not use R0H must follow this instruction.

Any memory stall or pipeline stall will also stall the floating-point unit. This keeps the floating-point unit aligned with the C28x pipeline and there is no need to change the code based on the waitstates of a memory block.

Example 3-1. 2p Instruction Pipeline Alignment

```
   ADDF32 R0H, #1.5, R1H     ; 2 pipeline cycles (2p)
   NOP                       ; 1 cycle delay or non-conflicting instruction
                             ; <-- ADDF32 completes, R0H updated
   NOP                       ; Any instruction
```

3.3 Moves from FPU Registers to C28x Registers

When transferring from the floating-point unit registers to the C28x CPU registers, additional pipeline alignment is required as shown in Example 3-2 and Example 3-3.

Example 3-2. Floating-Point to C28x Register Software Pipeline Alignment

```
;  MINF32: 32-bit floating-point minimum: single-cycle operation
;  An alignment cycle is required before copying R0H to ACC
   MINF32  R0H, R1H          ; Single-cycle instruction
                             ; <-- R0H is valid
   NOP                       ; Alignment cycle
   MOV32   @ACC, R0H         ; Copy R0H to ACC
```

For 1-cycle FPU instructions, one delay slot is required between a write to the floating-point register and the transfer instruction as shown in Example 3-2. For 2p FPU instructions, two delay slots are required between a write to the floating-point register and the transfer instruction as shown in Example 3-3.

Example 3-3. Floating-Point to C28x Register Software Pipeline Alignment

```
;  ADDF32: 32-bit floating-point addition: 2p operation
;  An alignment cycle is required before copying R0H to ACC
   ADDF32  R0H, R1H, #2      ; R0H = R1H + 2, 2 pipeline cycle instruction
   NOP                       ; 1 delay cycle or non-conflicting instruction
                             ; <-- R0H is valid
   NOP                       ; Alignment cycle
   MOV32   @ACC, R0H         ; Copy R0H to ACC
```

3.4 Moves from C28x Registers to FPU Registers

Transfers from the standard C28x CPU registers to the floating-point registers require four alignment cycles. In this case the four alignment cycles can be filled with NOPs or any non-conflicting instruction except for FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32. These instructions cannot replace any of the four alignment NOPs.

Example 3-4. C28x Register to Floating-Point Register Software Pipeline Alignment

```
;  Four alignment cycles are required after copying a standard 28x CPU
;  register to a floating-point register.
;
   MOV32   R0H,@ACC          ; Copy ACC to R0H
   NOP
   NOP
   NOP
   NOP                       ; Wait 4 cycles
   ADDF32  R2H,R1H,R0H       ; R0H is valid
```

3.5 Parallel Instructions

Parallel instructions are single opcodes that perform two operations in parallel. This can be a math operation in parallel with a move operation, or two math operations in parallel. Math operations with a parallel move are referred to as 2p/1 instructions. The math portion of the operation takes 2 pipelined cycles while the move portion of the operation is single cycle. This means that NOPs or other non conflicting instructions must be inserted to align the math portion of the operation. An example of an add with parallel move instruction is shown in Example 3-5.

Example 3-5. 2p/1 Parallel Instruction Software Pipeline Alignment

```
;  ADDF32 || MOV32 instruction: 32-bit floating-point add with parallel move
;  ADDF32 is a 2p operation
;  MOV32  is a 1 cycle operation
;
   ADDF32  R0H, R1H, #2      ; R0H = R1H + 2, 2 pipeline cycle operation
|| MOV32   R1H, @Val         ; R1H gets the contents of Val, single cycle operation
                             ; <-- MOV32 completes here (R1H is valid)
   NOP                       ; 1 cycle delay or non-conflicting instruction
                             ; <-- ADDF32 completes here (R0H is valid)
   NOP                       ; Any instruction
```

Parallel math instructions are referred to as 2p/2p instructions. Both math operations take 2 cycles to complete. This means that NOPs or other non conflicting instructions must be inserted to align the both math operations. An example of a multiply with parallel add instruction is shown in Example 3-5

Example 3-6. 2p/2p Parallel Instruction Software Pipeline Alignment

```
;  MPYF32 || ADDF32 instruction: 32-bit floating-point multiply with parallel add
;  MPYF32 is a 2p operation
;  ADDF32 is a 2p cycle operation
;
   MPYF32  R0H, R1H, R3H     ; R0H = R1H * R3H, 2 pipeline cycle operation
|| ADDF32  R1H, R2H, R4H     ; R1H = R2H + R4H, 2 pipeline cycle operation
   NOP                       ; 1 cycle delay or non-conflicting instruction
                             ; <-- MPYF32 and ADDF32 complete here (R0H and R1H are valid)
   NOP                       ; Any instruction
```

3.6 Invalid Delay Instructions

Most instructions can be used in delay slots as long as source and destination register conflicts are avoided. The C28x+FPU assembler will issue an error anytime you use an conflicting instruction within a delay slot. The following guidelines can be used to avoid these conflicts.

Note: *Destination register conflicts in delay slots:*

Any operation used for pipeline alignment delay must not use the same destination register as the instruction requiring the delay. See Example 3-7.

In Example 3-7 the MPYF32 instruction uses R2H as its destination register. The next instruction should not use R2H as its destination. Since the MOV32 instruction uses the R2H register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than R2H for the MOV32 instruction as shown in Example 3-8.

Example 3-7. Destination Register Conflict

```
; Invalid delay instruction. Both instructions use the same destination register
   MPYF32 R2H, R1H, R0H        ; 2p instruction
   MOV32  R2H, mem32           ; Invalid delay instruction
```

Example 3-8. Destination Register Conflict Resolved

```
; Valid delay instruction
   MPYF32 R2H, R1H, R0H        ; 2p instruction
   MOV32  R1H, mem32           ; Valid delay
                               ; <-- MPYF32 completes, R2H valid
```

Note: *Instructions in delay slots cannot use the instruction's destination register as a source register.*

Any operation used for pipeline alignment delay must not use the destination register of the instruction requiring the delay as a source register as shown in Example 3-9. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 3-11.

In Example 3-9 the MPYF32 instruction again uses R2H as its destination register. The next instruction should not use R2H as its source since the MPYF32 will take an additional cycle to complete. Since the ADDF32 instruction uses the R2H register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than R2H or by inserting a non-conflicting instruction between the MPYF32 and ADDF32 instructions. Since the SUBF32 does not use R2H this instruction can be moved before the ADDF32 as shown in Example 3-10.

Example 3-9. Destination/Source Register Conflict

```
; Invalid delay instruction. ADDF32 should not use R2H as a source operand
   MPYF32 R2H, R1H, R0H        ; 2p instruction
   ADDF32 R3H, R3H, R2H        ; Invalid delay instruction
   SUBF32 R4H, R1H, R0H
```

Example 3-10. Destination/Source Register Conflict Resolved

```
; Valid delay instruction.
   MPYF32 R2H, R1H, R0H        ; 2p instruction
   SUBF32 R4H, R1H, R0H        ; Valid delay for MPYF32
   ADDF32 R3H, R3H, R2H        ; <-- MPYF32 completes, R2H valid
   NOP
                               ; <-- SUBF32 completes, R4H valid
```

It should be noted that a source register for the 2nd operation within a parallel instruction can be the same as the destination register of the first operation. This is because the two operations are started at the same time. The 2nd operation is not in the delay slot of the first operation. Consider Example 3-11 where the MPYF32 uses R2H as its destination register. The MOV32 is the 2nd operation in the instruction and can freely use R2H as a source register. The contents of R2H before the multiply will be used by MOV32.

Example 3-11. Parallel Instruction Destination/Source Exception

```
; Valid parallel operation.
   MPYF32 R2H, R1H, R0H        ; 2p/1 instruction
|| MOV32  mem32, R2H           ; <-- Uses R2H before the MPYF32
                               ; <-- mem32 updated
   NOP                         ; <-- Delay for MPYF32
                               ; <-- R2H updated
```

Likewise, the source register for the 2nd operation within a parallel instruction can be the same as one of the source registers of the first operation. The MPYF32 operation in Example 3-12 uses the R1H register as one of its sources. This register is also updated by the MOV32 register. The multiplication operation will use the value in R1H before the MOV32 updates it.

Example 3-12. Parallel Instruction Destination/Source Exception

```
; Valid parallel instruction
   MPYF32 R2H, R1H, R0H        ; 2p/1 instruction
|| MOV32  R1H, mem32           ; Valid
   NOP                         ; <-- MOV32 completes, R1H valid
                               ; <-- MPYF32, R2H valid
```

Note: *Operations within parallel instructions cannot use the same destination register.*

When two parallel operations have the same destination register, the result is invalid.

For example, see Example 3-13.

If both operations within a parallel instruction try to update the same destination register as shown in Example 3-13 the assembler will issue an error.

Example 3-13. Invalid Destination Within a Parallel Instruction

```
; Invalid parallel instruction. Both operations use the same destination register
   MPYF32 R2H, R1H, R0H        ; 2p/1 instruction
|| MOV32  R2H, mem32           ; Invalid
```

Some instructions access or modify the STF flags. Because the instruction requiring a delay slot will also be accessing the STF flags, these instructions should not be used in delay slots. These instructions are SAVE, SETFLG, RESTORE and MOVST0.

Note: *Do not use SAVE, SETFLG, RESTORE, or the MOVST0 instruction in a delay slot.*

3.7 Optimizing the Pipeline

The following example shows how delay slots can be used to improve the performance of an algorithm. The example performs two Y = MX+B operations. In Example 3-14 , no optimization has been done. The Y = MX+B calculations are sequential and each takes 7 cycles to complete. Notice there are NOPs in the delay slots that could be filled with non-conflicting instructions. The only requirement is these instructions must not cause a register conflict or access the STF register flags.

Example 3-14. Floating-Point Code Without Pipeline Optimization

```
; Using NOPs for alignment cycles, calculate the following:
;
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2
;

; Calculate Y1
;
   MOV32   R0H,@M1            ; Load R0H with M1 - single cycle
   MOV32   R1H,@X1            ; Load R1H with X1 - single cycle
   MPYF32  R1H,R1H,R0H        ; R1H = M1 * X1    - 2p operation
|| MOV32   R0H,@B1            ; Load R0H with B1 - single cycle
   NOP                        ; Wait for MPYF32 to complete
                              ; <-- MPYF32 completes, R1H is valid
   ADDF32  R1H,R1H,R0H        ; R1H = R1H + R0H  - 2p operation
   NOP                        ; Wait for ADDF32 to complete
                              ; <-- ADDF32 completes, R1H is valid
   MOV32   @Y1,R1H            ; Save R1H in Y1   - single cycle

; Calculate Y2

   MOV32   R0H,@M2            ; Load R0H with M2 - single cycle
   MOV32   R1H,@X2            ; Load R1H with X2 - single cycle
   MPYF32  R1H,R1H,R0H        ; R1H = M2 * X2    - 2p operation
|| MOV32   R0H,@B2            ; Load R0H with B2 - single cycle
   NOP                        ; Wait for MPYF32 to complete
                              ; <-- MPYF32 completes, R1H is valid
   ADDF32  R1H,R1H,R0H        ; R1H = R1H + R0H
   NOP                        ; Wait for ADDF32 to complete
                              ; <-- ADDF32 completes, R1H is valid
   MOV32   @Y2,R1H            ; Save R1H in Y2

; 14 cycles
; 48 bytes
```

The code shown in Example 3-15 was generated by the C28x+FPU compiler with optimization enabled. Notice that the NOPs in the first example have now been filled with other instructions. The code for the two Y = MX+B calculations are now interleaved and both calculations complete in only 9 cycles.

Example 3-15. Floating-Point Code With Pipeline Optimization

```
; Using non-conflicting instructions for alignment cycles,
; calculate the following:
;
; Y1 = M1*X1 + B1
; Y2 = M2*X2 + B2
;
   MOV32     R2H,@X1          ; Load R2H with X1 - single cycle
   MOV32     R1H,@M1          ; Load R1H with M1 - single cycle
   MPYF32    R3H,R2H,R1H      ; R3H = M1 * X1    - 2p operation
|| MOV32     R0H,@M2          ; Load R0H with M2 - single cycle
   MOV32     R1H,@X2          ; Load R1H with X2 - single cycle
                              ; <-- MPYF32 completes, R3H is valid
   MPYF32    R0H,R1H,R0H      ; R0H = M2 * X2    - 2p operation
|| MOV32     R4H,@B1          ; Load R4H with B1 - single cycle
                              ; <-- MOV32 completes, R4H is valid
   ADDF32    R1H,R4H,R3H      ; R1H = B1 + M1*X1 - 2p operation
|| MOV32     R2H,@B2          ; Load R2H with B2 - single cycle
                              ; <-- MPYF32 completes, R0H is valid
   ADDF32    R0H,R2H,R0H      ; R0H = B2 + M2*X2 - 2p operation
                              ; <-- ADDF32 completes, R1H is valid
   MOV32     @Y1,R1H          ; Store Y1
                              ; <-- ADDF32 completes, R0H is valid
   MOV32     @Y2,R0H          ; Store Y2

;  9 cycles
; 36 bytes
```

Instruction Set

This chapter describes the assembly language instructions of the TMS320C28x plus floating-point processor. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are an extension to the standard C28x instruction set. For information on standard C28x instructions, see the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (literature number SPRU430).

Topic **Page**

4.1 Instruction Descriptions **30**
4.2 Instructions **32**

4.1 Instruction Descriptions

This section gives detailed information on the instruction set. Each instruction may present the following information:

- Operands
- Opcode
- Description
- Exceptions
- Pipeline
- Examples
- See also

The example INSTRUCTION is shown to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information. On the C28x+FPU instructions, follow the same format as the C28x. The source operand(s) are always on the right and the destination operand(s) are on the left.

The explanations for the syntax of the operands used in the instruction descriptions for the TMS320C28x plus floating-point processor are given in Table 4-1. For information on the operands of standard C28x instructions, see the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430).

Table 4-1. Operand Nomenclature

Symbol	Description
#16FHi	16-bit immediate (hex or float) value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.
#16FHiHex	16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. Lower 16-bits of the mantissa are assumed to be zero.
#16FLoHex	A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value
#32Fhex	32-bit immediate value that represents an IEEE 32-bit floating-point value
#32F	Immediate float value represented in floating-point representation
#0.0	Immediate zero
#RC	16-bit immediate value for the repeat count
*(0:16bitAddr)	16-bit immediate address, zero extended
CNDF	Condition to test the flags in the STF register
FLAG	Selected flags from STF register (OR) 11 bit mask indicating which floating-point status flags to change
label	Label representing the end of the repeat block
mem16	Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location
mem32	Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location
RaH	R0H to R7H registers
RbH	R0H to R7H registers
RcH	R0H to R7H registers
RdH	R0H to R7H registers
ReH	R0H to R7H registers
RfH	R0H to R7H registers
RB	Repeat Block Register
STF	FPU Status Register
VALUE	Flag value of 0 or 1 for selected flag (OR) 11 bit mask indicating the flag value; 0 or 1

INSTRUCTION dest1, source1, source2 *Short Description*

Operands

dest1	description for the 1st operand for the instruction
source1	description for the 2nd operand for the instruction
source2	description for the 3rd operand for the instruction

Each instruction has a table that gives a list of the operands and a short description. Instructions always have their destination operand(s) first followed by the source operand(s).

Opcode This section shows the opcode for the instruction.

Description Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.

Restrictions Any constraints on the operands or use of the instruction imposed by the processor are discussed.

Pipeline This section describes the instruction in terms of pipeline cycles as described in Chapter 3.

Example Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution. All examples assume the device is running with the OBJMODE set to 1. Normally the boot ROM or the c-code initialization will set this bit.

See Also Lists related instructions.

4.2 Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 4-2. Summary of Instructions

Title	Page
ABSF32 RaH, RbH 32-bit Floating-Point Absolute Value	34
ADDF32 RaH, #16FHi, RbH 32-bit Floating-Point Addition	35
ADDF32 RaH, RbH, #16FHi 32-bit Floating-Point Addition	37
ADDF32 RaH, RbH, RcH 32-bit Floating-Point Addition	39
ADDF32 RdH, ReH, RfH ‖MOV32 mem32, RaH 32-bit Floating-Point Addition with Parallel Move	40
ADDF32 RdH, ReH, RfH ‖MOV32 RaH, mem32 32-bit Floating-Point Addition with Parallel Move	42
CMPF32 RaH, RbH 32-bit Floating-Point Compare for Equal, Less Than or Greater Than	44
CMPF32 RaH, #16FHi 32-bit Floating-Point Compare for Equal, Less Than or Greater Than	45
CMPF32 RaH, #0.0 32-bit Floating-Point Compare for Equal, Less Than or Greater Than	46
EINVF32 RaH, RbH 32-bit Floating-Point Reciprocal Approximation	47
EISQRTF32 RaH, RbH 32-bit Floating-Point Square-Root Reciprocal Approximation	49
F32TOI16 RaH, RbH Convert 32-bit Floating-Point Value to 16-bit Integer	51
F32TOI16R RaH, RbH Convert 32-bit Floating-Point Value to 16-bit Integer and Round	52
F32TOI32 RaH, RbH Convert 32-bit Floating-Point Value to 32-bit Integer	53
F32TOUI16 RaH, RbH Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer	54
F32TOUI16R RaH, RbH Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round	55
F32TOUI32 RaH, RbH Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer	56
FRACF32 RaH, RbH Fractional Portion of a 32-bit Floating-Point Value	57
I16TOF32 RaH, RbH Convert 16-bit Integer to 32-bit Floating-Point Value	58
I16TOF32 RaH, mem16 Convert 16-bit Integer to 32-bit Floating-Point Value	59
I32TOF32 RaH, mem32 Convert 32-bit Integer to 32-bit Floating-Point Value	60
I32TOF32 RaH, RbH Convert 32-bit Integer to 32-bit Floating-Point Value	61
MACF32 R3H, R2H, RdH, ReH, RfH 32-bit Floating-Point Multiply with Parallel Add	62
MACF32 R3H, R2H, RdH, ReH, RfH ‖MOV32 RaH, mem32 32-bit Floating-Point Multiply and Accumulate with Parallel Move	64
MACF32 R7H, R3H, mem32, *XAR7++ 32-bit Floating-Point Multiply and Accumulate	66
MACF32 R7H, R6H, RdH, ReH, RfH 32-bit Floating-Point Multiply with Parallel Add	68
MACF32 R7H, R6H, RdH, ReH, RfH ‖MOV32 RaH, mem32 32-bit Floating-Point Multiply and Accumulate with Parallel Move	70
MAXF32 RaH, RbH 32-bit Floating-Point Maximum	72
MAXF32 RaH, #16FHi 32-bit Floating-Point Maximum	73
MAXF32 RaH, RbH ‖MOV32 RcH, RdH 32-bit Floating-Point Maximum with Parallel Move	74
MINF32 RaH, RbH 32-bit Floating-Point Minimum	75
MINF32 RaH, #16FHi 32-bit Floating-Point Minimum	76
MINF32 RaH, RbH ‖MOV32 RcH, RdH 32-bit Floating-Point Minimum with Parallel Move	77
MOV16 mem16, RaH Move 16-bit Floating-Point Register Contents to Memory	78
MOV32 *(0:16bitAddr), loc32 Move the Contents of loc32 to Memory	79
MOV32 ACC, RaH Move 32-bit Floating-Point Register Contents to ACC	80
MOV32 loc32, *(0:16bitAddr) Move 32-bit Value from Memory to loc32	81
MOV32 mem32, RaH Move 32-bit Floating-Point Register Contents to Memory	82
MOV32 mem32, STF Move 32-bit STF Register to Memory	83
MOV32 P, RaH Move 32-bit Floating-Point Register Contents to P	84
MOV32 RaH, ACC Move the Contents of ACC to a 32-bit Floating-Point Register	85
MOV32 RaH, mem32 {, CNDF} Conditional 32-bit Move	86
MOV32 RaH, P Move the Contents of P to a 32-bit Floating-Point Register	88

Table 4-2. Summary of Instructions (continued)

MOV32 RaH, RbH {, CNDF} Conditional 32-bit Move 89
MOV32 RaH, XARn Move the Contents of XARn to a 32-bit Floating-Point Register 90
MOV32 RaH, XT Move the Contents of XT to a 32-bit Floating-Point Register 91
MOV32 STF, mem32 Move 32-bit Value from Memory to the STF Register 92
MOV32 XARn, RaH Move 32-bit Floating-Point Register Contents to XARn 93
MOV32 XT, RaH Move 32-bit Floating-Point Register Contents to XT 94
MOVD32 RaH, mem32 Move 32-bit Value from Memory with Data Copy 95
MOVF32 RaH, #32F Load the 32-bits of a 32-bit Floating-Point Register 96
MOVI32 RaH, #32FHex Load the 32-bits of a 32-bit Floating-Point Register with the immediate 97
MOVIZ RaH, #16FHiHex Load the Upper 16-bits of a 32-bit Floating-Point Register 98
MOVIZF32 RaH, #16FHi Load the Upper 16-bits of a 32-bit Floating-Point Register 99
MOVST0 FLAG Load Selected STF Flags into ST0 100
MOVXI RaH, #16FLoHex Move Immediate to the Low 16-bits of a Floating-Point Register 101
MPYF32 RaH, RbH, RcH 32-bit Floating-Point Multiply 102
MPYF32 RaH, #16FHi, RbH 32-bit Floating-Point Multiply 103
MPYF32 RaH, RbH, #16FHi 32-bit Floating-Point Multiply 104
MPYF32 RaH, RbH, RcH ||ADDF32 RdH, ReH, RfH 32-bit Floating-Point Multiply with Parallel Add 105
MPYF32 RdH, ReH, RfH ||MOV32 RaH, mem32 32-bit Floating-Point Multiply with Parallel Move 107
MPYF32 RdH, ReH, RfH ||MOV32 mem32, RaH 32-bit Floating-Point Multiply with Parallel Move 109
MPYF32 RaH, RbH, RcH ||SUBF32 RdH, ReH, RfH 32-bit Floating-Point Multiply with Parallel Subtract 110
NEGF32 RaH, RbH{, CNDF} Conditional Negation 111
POP RB Pop the RB Register from the Stack 112
PUSH RB Push the RB Register onto the Stack 113
RESTORE Restore the Floating-Point Registers 114
RPTB label, loc16 Repeat A Block of Code 116
RPTB label, #RC Repeat a Block of Code 118
SAVE FLAG, VALUE Save Register Set to Shadow Registers and Execute SETFLG 120
SETFLG FLAG, VALUE Set or clear selected floating-point status flags 122
SUBF32 RaH, RbH, RcH 32-bit Floating-Point Subtraction 123
SUBF32 RaH, #16FHi, RbH 32-bit Floating Point Subtraction 124
SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32 32-bit Floating-Point Subtraction with Parallel Move 125
SUBF32 RdH, ReH, RfH ||MOV32 mem32, RaH 32-bit Floating-Point Subtraction with Parallel Move 127
SWAPF RaH, RbH{, CNDF} Conditional Swap 128
TESTTF CNDF Test STF Register Flag Condition 129
UI16TOF32 RaH, mem16 Convert unsigned 16-bit integer to 32-bit floating-point value 130
UI16TOF32 RaH, RbH Convert unsigned 16-bit integer to 32-bit floating-point value 131
UI32TOF32 RaH, mem32 Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value 132
UI32TOF32 RaH, RbH Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value 133
ZERO RaH Zero the Floating-Point Register RaH 134
ZEROA Zero All Floating-Point Registers 135

ABSF32 RaH, RbH *32-bit Floating-Point Absolute Value*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 0101
MSW: 0000 0000  00bb baaa
```

Description

The absolute value of RbH is loaded into RaH. Only the sign bit of the operand is modified by the ABSF32 instruction.

```
if (RbH < 0) {RaH = -RbH}
else         {RaH =  RbH}
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The STF register flags are modified as follows:

```
NF = 0;
ZF = 0;
if ( RaH[30:23] == 0) ZF = 1;
```

Pipeline

This is a single-cycle instruction.

Example

```
   MOVIZF32  R1H, #-2.0   ; R1H = -2.0 (0xC0000000)
   ABSF32    R1H, R1H     ; R1H =  2.0 (0x40000000), ZF = NF = 0

   MOVIZF32  R0H, #5.0    ; R0H =  5.0 (0x40A00000)
   ABSF32    R0H, R0H     ; R0H =  5.0 (0x40A00000), ZF = NF = 0

   MOVIZF32  R0H, #0.0    ; R0H =  0.0
   ABSF32    R1H, R0H     ; R1H =  0.0  ZF = 1, NF = 0
```

See also

NEGF32 RaH, RbH{, CNDF}

ADDF32 RaH, #16FHi, RbH *32-bit Floating-Point Addition*

Operands

RaH	floating-point destination register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 1000  10II IIII
MSW: IIII IIII  IIbb baaa
```

Description

Add RbH to the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

```
RaH = RbH + #16FHi:0
```

This instruction can also be written as ADDF32 RaH, RbH, #16FHi.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

```
   ADDF32 RaH, #16FHi, RbH   ; 2 pipeline cycles (2p)
   NOP                       ; 1 cycle delay or non-conflicting instruction
                             ; <-- ADDF32 completes, RaH updated
   NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
 ; Add to R1H the value 2.0 in 32-bit floating-point format
   ADDF32 R0H, #2.0, R1H     ; R0H = 2.0 + R1H
   NOP                       ; Delay for ADDF32 to complete
                             ; <-- ADDF32 completes, R0H updated
   NOP                       ;

; Add to R3H the value -2.5 in 32-bit floating-point format
   ADDF32 R2H, #-2.5, R3H    ; R2H = -2.5 + R3H
   NOP                       ; Delay for ADDF32 to complete
                             ; <-- ADDF32 completes, R2H updated
   NOP                       ;

; Add to R5H the value 0x3FC00000 (1.5)
   ADDF32 R5H, #0x3FC0, R5H  ; R5H = 1.5 + R5H
   NOP                       ; Delay for ADDF32 to complete
                             ; <-- ADDF32 completes, R5H updated
   NOP                       ;
```

See also

ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

ADDF32 RaH, RbH, #16FHi *32-bit Floating-Point Addition*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  10II IIII
MSW: IIII IIII  IIbb baaa
```

Description

Add RbH to the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

```
RaH = RbH + #16FHi:0
```

This instruction can also be written as ADDF32 RaH, #16FHi, RbH.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

```
   ADDF32 RaH, #16FHi, RbH   ; 2 pipeline cycles (2p)
   NOP                       ; 1 cycle delay or non-conflicting instruction
                             ; <-- ADDF32 completes, RaH updated
   NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
; Add to R1H the value 2.0 in 32-bit floating-point format
   ADDF32 R0H, R1H, #2.0     ; R0H = R1H + 2.0
   NOP                       ; Delay for ADDF32 to complete
                             ; <-- ADDF32 completes, R0H updated
   NOP                       ;

; Add to R3H the value -2.5 in 32-bit floating-point format
   ADDF32 R2H, R3H, #-2.5    ; R2H = R3H + (-2.5)
   NOP                       ; Delay for ADDF32 to complete
                             ; <-- ADDF32 completes, R2H updated
   NOP                       ;

; Add to R5H the value 0x3FC00000 (1.5)
   ADDF32 R5H, R5H, #0x3FC0  ; R5H = R5H + 1.5
   NOP                       ; Delay for ADDF32 to complete
                             ; <-- ADDF32 completes, R5H updated
   NOP                       ;
```

See also

ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

ADDF32 RaH, RbH, RcH *32-bit Floating-Point Addition*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)
RcH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0111  0001 0000
MSW: 0000 000c  ccbb baaa
```

Description

Add the contents of RcH to the contents of RbH and load the result into RaH.

```
RaH = RbH + RcH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

This is a 2 pipeline-cycle instruction (2p). That is:

```
   ADDF32 RaH, RbH, RcH      ; 2 pipeline cycles (2p)
   NOP                       ; 1 cycle delay or non-conflicting instruction
                             ; <-- ADDF32 completes, RaH updated
   NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = M1*X1 + B1. This example assumes that M1, X1, B1 and Y are all on the same data page.

```
   MOVW    DP, #M1        ; Load the data page
   MOV32   R0H,@M1        ; Load R0H with M1
   MOV32   R1H,@X1        ; Load R1H with X1
   MPYF32  R1H,R1H,R0H    ; Multiply M1*X1
|| MOV32   R0H,@B1        ;     and in parallel load R0H with B1
   NOP                    ; <-- MOV32 complete
                          ; <-- MPYF32 complete
   ADDF32  R1H,R1H,R0H    ; Add M*X1 to B1 and store in R1H
   NOP
                          ; <-- ADDF32 complete
   MOV32   @Y1,R1H        ; Store the result
```

Calculate Y = A + B.

```
   MOVL   XAR4, #A
   MOV32  R0H,  *XAR4   ; Load R0H with A
   MOVL   XAR4, #B
   MOV32  R1H,  *XAR4   ; Load R1H with B
   ADDF32 R0H,R1H,R0H   ; Add A + B R0H=R0H+R1H
   MOVL   XAR4, #Y
                        ; <-- ADDF32 complete
   MOV32 *XAR4,R0H      ; Store the result
```

See also

ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, #16FHi
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

ADDF32 RdH, ReH, RfH
‖MOV32 mem32, RaH *32-bit Floating-Point Addition with Parallel Move*

Operands

RdH	floating-point destination register for the ADDF32 (R0H to R7H)
ReH	floating-point source register for the ADDF32 (R0H to R7H)
RfH	floating-point source register for the ADDF32 (R0H to R7H)
mem32	pointer to a 32-bit memory location. This will be the destination of the MOV32.
RaH	floating-point source register for the MOV32 (R0H to R7H)

Opcode

```
LSW: 1110 0000 0001 fffe
MSW: eedd daaa    mem32
```

Description

Perform an ADDF32 and a MOV32 in parallel. Add RfH to the contents of ReH and store the result in RdH. In parallel move the contents of RaH to the 32-bit location pointed to by mem32. mem32 addresses memory using any of the direct or indirect addressing modes supported by the C28x CPU.

```
RdH = ReH + RfH,
[mem32] = RaH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

Pipeline

ADDF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

```
    ADDF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
||  MOV32   mem32, RaH      ; 1 cycle
                            ; <-- MOV32 completes, mem32 updated
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- ADDF32 completes, RdH updated
    NOP
```

Any instruction in the delay slot must not use RdH as a destination register or use RdH as a source operand.

Example

```
   ADDF32      R3H, R6H, R4H          ; (A) R3H = R6H + R4H and R7H = I3
|| MOV32       R7H, *-SP[2]           ;
                                      ; <-- R7H valid
   SUBF32      R6H, R6H, R4H          ; (B) R6H = R6H - R4H
                                      ; <-- ADDF32 (A) completes, R3H valid
   SUBF32      R3H, R1H, R7H          ; (C) R3H = R1H - R7H and store R3H (A)
|| MOV32       *+XAR5[2], R3H         ;
                                      ; <-- SUBF32 (B) completes, R6H valid
                                      ; <-- MOV32 completes, (A) stored
   ADDF32      R4H, R7H, R1H          ; R4H = D = R7H + R1H and store R6H (B)
|| MOV32       *+XAR5[6], R6H         ;
                                      ; <-- SUBF32 (C) completes, R3H valid
                                      ; <-- MOV32 completes, (B) stored
   MOV32       *+XAR5[0], R3H         ; store R3H (C)
                                      ; <-- MOV32 completes, (C) stored
                                      ; <-- ADDF32 (D) completes, R4H valid
   MOV32       *+XAR5[4], R4H         ; store R4H (D)
                                      ; <-- MOV32 completes, (D) stored
```

See also

ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32

ADDF32 RdH, ReH, RfH
||MOV32 RaH, mem32 *32-bit Floating-Point Addition with Parallel Move*

Operands

RdH	floating-point destination register for the ADDF32 (R0H to R7H). RdH cannot be the same register as RaH.
ReH	floating-point source register for the ADDF32 (R0H to R7H)
RfH	floating-point source register for the ADDF32 (R0H to R7H)
RaH	floating-point destination register for the MOV32 (R0H to R7H). RaH cannot be the same register as RdH.
mem32	pointer to a 32-bit memory location. This is the source for the MOV32.

Opcode

```
LSW: 1110 0011  0001 fffe
MSW: eedd daaa    mem32
```

Description

Perform an ADDF32 and a MOV32 operation in parallel. Add RfH to the contents of ReH and store the result in RdH. In parallel move the contents of the 32-bit location pointed to by mem32 to RaH. mem32 addresses memory using any of the direct or indirect addressing modes supported by the C28x CPU.

```
RdH = ReH + RfH,
RaH = [mem32]
```

Restrictions

The destination register for the ADDF32 and the MOV32 must be unique. That is, RaH and RdH cannot be the same register.

Any instruction in the delay slot must not use RdH as a destination register or use RdH as a source operand.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if ADDF32 generates an underflow condition.
- LVF = 1 if ADDF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

```
NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;
```

Pipeline

The ADDF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

```
    ADDF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
 || MOV32   RaH, mem32      ; 1 cycle
                            ; <-- MOV32 completes, RaH updated
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- ADDF32 completes, RdH updated
    NOP
```

Example

Calculate Y = A + B - C:

```
    MOVL   XAR4, #A
    MOV32  R0H,  *XAR4  ; Load R0H with A
    MOVL   XAR4, #B
    MOV32  R1H,  *XAR4  ; Load R1H with B
    MOVL   XAR4, #C
    ADDF32 R0H,R1H,R0H  ; Add A + B and in parallel
 || MOV32  R2H, *XAR4   ; Load R2H with C
                        ; <-- MOV32 complete
    MOVL   XAR4,#Y
                        ; ADDF32 complete
    SUBF32 R0H,R0H,R2H  ; Subtract C from (A + B)
    NOP
                        ; <-- SUBF32 completes
    MOV32  *XAR4,R0H    ; Store the result
```

See also

ADDF32 RaH, #16FHi, RbH
ADDF32 RaH, RbH, #16FHi
ADDF32 RaH, RbH, RcH
ADDF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

CMPF32 RaH, RbH *32-bit Floating-Point Compare for Equal, Less Than or Greater Than*

Operands

RaH	floating-point source register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 0100
MSW: 0000 0000  00bb baaa
```

Description

Set ZF and NF flags on the result of RaH - RbH. The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE format offsetting the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- A denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The STF register flags are modified as follows:

```
If(RaH == RbH) {ZF=1, NF=0}
If(RaH >  RbH) {ZF=0, NF=0}
If(RaH <  RbH) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
; Behavior of ZF and NF flags for different comparisons

   MOVIZF32  R1H, #-2.0    ; R1H = -2.0 (0xC0000000)
   MOVIZF32  R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   CMPF32    R1H, R0H      ; ZF = 0, NF = 1
   CMPF32    R0H, R1H      ; ZF = 0, NF = 0
   CMPF32    R0H, R0H      ; ZF = 1, NF = 0

; Using the result of a compare for loop control

 Loop:
   MOV32     R0H,*XAR4++   ; Load R0H
   MOV32     R1H,*XAR3++   ; Load R1H
   CMPF32    R1H, R0H      ; Set/clear ZF and NF
   MOVST0    ZF, NF        ; Copy ZF and NF to ST0 Z and N bits
   BF        Loop, GT      ; Loop if R1H > R0H
```

See also

CMPF32 RaH, #16FHi
CMPF32 RaH, #0.0
MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, #16FHi
MINF32 RaH, RbH

CMPF32 RaH, #16FHi *32-bit Floating-Point Compare for Equal, Less Than or Greater Than*

Operands

RaH	floating-point source register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  0001 0III
MSW: IIII IIII  IIII Iaaa
```

Description

Compare the value in RaH with the floating-point value represented by the immediate operand. Set the ZF and NF flags on (RaH - #16FHi:0).

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- Denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The STF register flags are modified as follows:

```
If(RaH == #16FHi:0) {ZF=1, NF=0}
If(RaH >  #16FHi:0) {ZF=0, NF=0}
If(RaH <  #16FHi:0) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction

Example

```
; Behavior of ZF and NF flags for different comparisons

   MOVIZF32  R1H, #-2.0    ; R1H = -2.0 (0xC0000000)
   MOVIZF32  R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   CMPF32    R1H, #-2.2    ; ZF = 0, NF = 0
   CMPF32    R0H, #6.5     ; ZF = 0, NF = 1
   CMPF32    R0H, #5.0     ; ZF = 1, NF = 0

; Using the result of a compare for loop control

 Loop:
   MOV32     R1H,*XAR3++   ; Load R1H
   CMPF32    R1H, #2.0     ; Set/clear ZF and NF
   MOVST0    ZF, NF        ; Copy ZF and NF to ST0 Z and N bits
   BF        Loop, GT      ; Loop if R1H > #2.0
```

See also

CMPF32 RaH, #0.0
CMPF32 RaH, RbH
MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, #16FHi
MINF32 RaH, RbH

CMPF32 RaH, #0.0 *32-bit Floating-Point Compare for Equal, Less Than or Greater Than*

Operands

RaH	floating-point source register (R0H to R7H)
#0.0	zero

Opcode

```
LSW: 1110 0101 1010 0aaa
```

Description

Set the ZF and NF flags on (RaH - #0.0). The CMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

- Negative zero will be treated as positive zero.
- Denormalized value will be treated as positive zero.
- Not-a-Number (NaN) will be treated as infinity.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The STF register flags are modified as follows:

```
If(RaH == #0.0) {ZF=1, NF=0}
If(RaH >  #0.0) {ZF=0, NF=0}
If(RaH <  #0.0) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
; Behavior of ZF and NF flags for different comparisons

   MOVIZF32  R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   MOVIZF32  R1H, #-2.0    ; R1H = -2.0 (0xC0000000)
   MOVIZF32  R2H, #0.0     ; R2H =  0.0 (0x00000000)
   CMPF32    R0H, #0.0     ; ZF = 0, NF = 0
   CMPF32    R1H, #0.0     ; ZF = 0, NF = 1
   CMPF32    R2H, #0.0     ; ZF = 1, NF = 0

; Using the result of a compare for loop control

 Loop:
   MOV32     R1H,*XAR3++   ; Load R1H
   CMPF32    R1H, #0.0     ; Set/clear ZF and NF
   MOVST0    ZF, NF        ; Copy ZF and NF to ST0 Z and N bits
   BF        Loop, GT      ; Loop if R1H > #0.0
```

See also

CMPF32 RaH, #0.0
CMPF32 RaH, #16FHi
MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, #16FHi
MINF32 RaH, RbH

EINVF32 RaH, RbH *32-bit Floating-Point Reciprocal Approximation*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 0011
MSW: 0000 0000  00bb baaa
```

Description

This operation generates an estimate of 1/X in 32-bit floating-point format accurate to approximately 8 bits. This value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

```
Ye = Estimate(1/X);
Ye = Ye*(2.0 - Ye*X)
Ye = Ye*(2.0 - Ye*X)
```

After 2 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The EINVF32 operation will not generate a negative zero, DeNorm or NaN value.

```
RaH = Estimate of 1/RbH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if EINVF32 generates an underflow condition.
- LVF = 1 if EINVF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
EINVF32  RaH, RbH       ; 2p
NOP                     ; 1 cycle delay or non-conflicting instruction
                        ; <-- EINVF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = A/B. A fast division routine similar to that shown below can be found in the *C28x FPU Fast RTS Library* (SPRC664).

```
    MOVL  XAR4, #A
    MOV32 R0H,  *XAR4     ; Load R0H with A
    MOVL  XAR4, #B
    MOV32 R1H,  *XAR4     ; Load R1H with B
    LCR   DIV             ; Calculate R0H = R0H / R1H
    MOV32 *XAR4, R0H      ;
....

DIV:
    EINVF32     R2H, R1H            ; R2H = Ye = Estimate(1/B)
    CMPF32      R0H, #0.0           ; Check if A == 0
    MPYF32      R3H, R2H, R1H       ; R3H = Ye*B
    NOP
    SUBF32      R3H, #2.0, R3H      ; R3H = 2.0 - Ye*B
    NOP
    MPYF32      R2H, R2H, R3H       ; R2H = Ye = Ye*(2.0 - Ye*B)
    NOP
    MPYF32      R3H, R2H, R1H       ; R3H = Ye*B
    CMPF32      R1H, #0.0           ; Check if B == 0.0
    SUBF32      R3H, #2.0, R3H      ; R3H = 2.0 - Ye*B
    NEGF32      R0H, R0H, EQ        ; Fixes sign for A/0.0
    MPYF32      R2H, R2H, R3H       ; R2H = Ye = Ye*(2.0 - Ye*B)
    NOP
    MPYF32      R0H, R0H, R2H       ; R0H = Y = A*Ye = A/B
    LRETR
```

See also

EISQRTF32 RaH, RbH

EISQRTF32 RaH, RbH *32-bit Floating-Point Square-Root Reciprocal Approximation*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 0010
MSW: 0000 0000  00bb baaa
```

Description

This operation generates an estimate of 1/sqrt(X) in 32-bit floating-point format accurate to approximately 8 bits. This value can be used in a Newton-Raphson algorithm to get a more accurate answer. That is:

```
Ye = Estimate(1/sqrt(X));
Ye = Ye*(1.5 - Ye*Ye*X/2.0)
Ye = Ye*(1.5 - Ye*Ye*X/2.0)
```

After 2 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The EISQRTF32 operation will not generate a negative zero, DeNorm or NaN value.

```
RaH = Estimate of 1/sqrt (RbH)
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if EISQRTF32 generates an underflow condition.
- LVF = 1 if EISQRTF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
EINVF32  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                     ; 1 cycle delay or non-conflicting instruction
                        ; <-- EISQRTF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate the square root of X. A square-root routine similar to that shown below can be found in the *C28x FPU Fast RTS Library* (SPRC664).

```
; Y = sqrt(X)
; Ye = Estimate(1/sqrt(X));
; Ye = Ye*(1.5 - Ye*Ye*X*0.5)
; Ye = Ye*(1.5 - Ye*Ye*X*0.5)
; Y  = X*Ye
_sqrt:
                                    ; R0H = X on entry
    EISQRTF32   R1H, R0H            ; R1H = Ye = Estimate(1/sqrt(X))
    MPYF32      R2H, R0H, #0.5      ; R2H = X*0.5
    MPYF32      R3H, R1H, R1H       ; R3H = Ye*Ye
    NOP
    MPYF32      R3H, R3H, R2H       ; R3H = Ye*Ye*X*0.5
    NOP
    SUBF32      R3H, #1.5, R3H      ; R3H = 1.5 - Ye*Ye*X*0.5
    NOP
    MPYF32      R1H, R1H, R3H       ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    NOP
    MPYF32      R3H, R1H, R2H       ; R3H = Ye*X*0.5
    NOP
    MPYF32      R3H, R1H, R3H       ; R3H = Ye*Ye*X*0.5
    NOP
    SUBF32      R3H, #1.5, R3H      ; R3H = 1.5 - Ye*Ye*X*0.5
    CMPF32      R0H, #0.0           ; Check if X == 0
    MPYF32      R1H, R1H, R3H       ; R2H = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    NOP
    MOV32       R1H, R0H, EQ        ; If X is zero, change the Ye estimate to 0
    MPYF32      R0H, R0H, R1H       ; R0H = Y = X*Ye = sqrt(X)
    LRETR
```

See also

EINVF32 RaH, RbH

F32TOI16 RaH, RbH *Convert 32-bit Floating-Point Value to 16-bit Integer*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1100
MSW: 0000 0000  00bb baaa
```

Description

Convert a 32-bit floating point value in RbH to a 16-bit integer and truncate. The result will be stored in RaH.

```
RaH(15:0)  = F32TOI16(RbH)
RaH(31:16) = sign extension of RaH(15)
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOI16  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- F32TOI16 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZF32   R0H, #5.0  ; R0H = 5.0 (0x40A00000)
F32TOI16   R1H, R0H   ; R1H(15:0)  = F32TOI16(R0H)
                      ; R1H(31:16) = Sign extension of R1H(15)
MOVIZF32   R2H, #-5.0 ; R2H = -5.0 (0xC0A00000)
                      ; <-- F32TOI16 complete, R1H(15:0)  = 5 (0x0005)
                      ;                        R1H(31:16) = 0 (0x0000)
F32TOI16   R3H, R2H   ; R3H(15:0)  = F32TOI16(R2H)
                      ; R3H(31:16) = Sign extension of R3H(15)
NOP                   ; 1 Cycle delay for F32TOI16 to complete
                      ; <-- F32TOI16 complete, R3H(15:0)  = -5 (0xFFFB)
                      ;                        R3H(31:16) = (0xFFFF)
```

See also

F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

F32TOI16R RaH, RbH *Convert 32-bit Floating-Point Value to 16-bit Integer and Round*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1100
MSW: 1000 0000  00bb baaa
```

Description

Convert the 32-bit floating point value in RbH to a 16-bit integer and round to the nearest even value. The result is stored in RaH.

```
RaH(15:0)  = F32ToI16round(RbH)
RaH(31:16) = sign extension of RaH(15)
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOI16R  RaH, RbH      ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- F32TOI16R completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZ      R0H, #0x3FD9  ; R0H [31:16] = 0x3FD9
MOVXI      R0H, #0x999A  ; R0H [15:0]  = 0x999A
                         ; R0H = 1.7 (0x3FD9999A)
F32TOI16R  R1H, R0H      ; R1H(15:0)  = F32TOI16round (R0H)
                         ; R1H(31:16) = Sign extension of R1H(15)
MOVF32     R2H, #-1.7    ; R2H = -1.7 (0xBFD9999A)
                         ; <-- F32TOI16R complete, R1H(15:0)  = 2 (0x0002)
                         ;                         R1H(31:16) = 0 (0x0000)
F32TOI16R  R3H, R2H      ; R3H(15:0)  = F32TOI16round (R2H)
                         ; R3H(31:16) = Sign extension of R2H(15)
NOP                      ; 1 Cycle delay for F32TOI16R to complete
                         ; <-- F32TOI16R complete, R1H(15:0)  = -2 (0xFFFE)
                         ;                         R1H(31:16) = (0xFFFF)
```

See also

F32TOI16 RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

F32TOI32 RaH, RbH *Convert 32-bit Floating-Point Value to 32-bit Integer*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1000
MSW: 0000 0000  00bb baaa
```

Description

Convert the 32-bit floating-point value in RbH to a 32-bit integer value and truncate. Store the result in RaH.

```
RaH = F32TOI32(RbH)
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOI32  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- F32TOI32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVF32    R2H, #11204005.0  ; R2H = 11204005.0 (0x4B2AF5A5)
F32TOI32  R3H, R2H          ; R3H = F32TOI32 (R2H)
MOVF32    R4H, #-11204005.0 ; R4H = -11204005.0 (0xCB2AF5A5)
                            ; <-- F32TOI32 complete,
                            ; R3H = 11204005 (0x00AAF5A5)
F32TOI32  R5H, R4H          ; R5H = F32TOI32 (R4H)
NOP                         ; 1 Cycle delay for F32TOI32 to complete
                            ; <-- F32TOI32 complete,
                            ; R5H = -11204005 (0xFF550A5B)
```

See also

F32TOUI32 RaH, RbH
I32TOF32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32

F32TOUI16 RaH, RbH *Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1110
MSW: 0000 0000  00bb baaa
```

Description

Convert the 32-bit floating point value in RbH to an unsigned 16-bit integer value and truncate to zero. The result will be stored in RaH. To instead round the integer to the nearest even value use the F32TOUI16R instruction.

```
RaH(15:0)  = F32ToUI16(RbH)
RaH(31:16) = 0x0000
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOUI16  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                       ; 1 cycle delay or non-conflicting instruction
                          ; <-- F32TOUI16 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZF32   R4H, #9.0  ; R4H = 9.0 (0x41100000)
F32TOUI16  R5H, R4H   ; R5H (15:0) = F32TOUI16 (R4H)
                      ; R5H (31:16) = 0x0000
MOVIZF32   R6H, #-9.0 ; R6H = -9.0 (0xC1100000)
                      ; <-- F32TOUI16 complete, R5H (15:0)  = 9.0 (0x0009)
                      ;                         R5H (31:16) = 0.0 (0x0000)
F32TOUI16  R7H, R6H   ; R7H (15:0) = F32TOUI16 (R6H)
                      ; R7H (31:16) = 0x0000
NOP                   ; 1 Cycle delay for F32TOUI16 to complete
                      ; <-- F32TOUI16 complete, R7H (15:0)  = 0.0 (0x0000)
                      ;                         R7H (31:16) = 0.0 (0x0000)
```

See also

F32TOI16 RaH, RbH
F32TOUI16R RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

F32TOUI16R RaH, RbH *Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1110
MSW: 1000 0000  00bb baaa
```

Description

Convert the 32-bit floating-point value in RbH to an unsigned 16-bit integer and round to the closest even value. The result will be stored in RaH. To instead truncate the converted value, use the F32TOUI16 instruction.

```
RaH(15:0)  = F32ToUI16round(RbH)
RaH(31:16) = 0x0000
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOUI16R  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                        ; 1 cycle delay or non-conflicting instruction
                           ; <-- F32TOUI16R completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZ       R5H, #0x412C  ; R5H = 0x412C
MOVXI       R5H, #0xCCCD  ; R5H = 0xCCCD
                          ; R5H = 10.8 (0x412CCCCD)
F32TOUI16R  R6H, R5H      ; R6H (15:0)  = F32TOUI16round (R5H)
                          ; R6H (31:16) = 0x0000
MOVF32      R7H, #-10.8   ; R7H = -10.8 (0x0xC12CCCCD)
                          ; <-- F32TOUI16R complete,
                          ; R6H (15:0)  = 11.0 (0x000B)
                          ; R6H (31:16) = 0.0  (0x0000)
F32TOUI16R  R0H, R7H      ; R0H (15:0)  = F32TOUI16round (R7H)
                          ; R0H (31:16) = 0x0000
NOP                       ; 1 Cycle delay for F32TOUI16R to complete
                          ; <-- F32TOUI16R complete,
                          ; R0H (15:0)  = 0.0 (0x0000)
                          ; R0H (31:16) = 0.0 (0x0000)
```

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

F32TOUI32 RaH, RbH *Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1010
MSW: 0000 0000  00bb baaa
```

Description

Convert the 32-bit floating-point value in RbH to an unsigned 32-bit integer and store the result in RaH.

```
RaH  = F32ToUI32(RbH)
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
F32TOUI32  RaH, RbH      ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- F32TOUI32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZF32   R6H, #12.5  ; R6H = 12.5 (0x41480000)
F32TOUI32  R7H, R6H    ; R7H = F32TOUI32 (R6H)
MOVIZF32   R1H, #-6.5  ; R1H = -6.5 (0xC0D00000)
                       ; <-- F32TOUI32 complete, R7H = 12.0 (0x0000000C)
F32TOUI32  R2H, R1H    ; R2H = F32TOUI32 (R1H)
NOP                    ; 1 Cycle delay for F32TOUI32 to complete
                       ; <-- F32TOUI32 complete, R2H = 0.0 (0x00000000)
```

See also

F32TOI32 RaH, RbH
I32TOF32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32

FRACF32 RaH, RbH *Fractional Portion of a 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1111 0001
MSW: 0000 0000  00bb baaa
```

Description

Returns in RaH the fractional portion of the 32-bit floating-point value in RbH

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
FRACF32  RaH, RbH        ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- FRACF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZF32  R2H, #19.625  ; R2H = 19.625 (0x419D0000)
FRACF32   R3H, R2H      ; R3H = FRACF32 (R2H)
NOP                     ; 1 Cycle delay for FRACF32 to complete
                        ; <-- FRACF32 complete, R3H = 0.625 (0x3F200000)
```

See also

I16TOF32 RaH, RbH *Convert 16-bit Integer to 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1101
MSW: 0000 0000  00bb baaa
```

Description

Convert the 16-bit signed integer in RbH to a 32-bit floating point value and store the result in RaH.

```
RaH = I16ToF32 RbH
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
I16TOF32  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- I16TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZ     R0H, #0x0000  ; R0H[31:16] = 0.0 (0x0000)
MOVXI     R0H, #0x0004  ; R0H[15:0]  = 4.0 (0x0004)
I16TOF32  R1H, R0H      ; R1H = I16TOF32 (R0H)
MOVIZ     R2H, #0x0000  ; R2H[31:16] = 0.0 (0x0000)
                        ; <--I16TOF32 complete, R1H = 4.0 (0x40800000)
MOVXI     R2H, #0xFFFC  ; R2H[15:0] = -4.0 (0xFFFC)
I16TOF32  R3H, R2H      ; R3H = I16TOF32 (R2H)
NOP                     ; 1 Cycle delay for I16TOF32 to complete
                        ; <-- I16TOF32 complete, R3H = -4.0 (0xC0800000)
```

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

I16TOF32 RaH, mem16 *Convert 16-bit Integer to 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
mem316	16-bit source memory location to be converted

Opcode

```
LSW: 1110 0110  1100 1000
MSW: 0000 0aaa    mem16
```

Description

Convert the 16-bit signed integer indicated by the mem16 pointer to a 32-bit floating-point value and store the result in RaH.

```
RaH = I16ToF32[mem16]
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
I16TOF32  RaH, mem16     ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- I16TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVW       DP, #0x0280  ; DP = 0x0280
MOV        @0, #0x0004  ; [0x00A000] = 4.0 (0x0004)
I16TOF32   R0H, @0      ; R0H = I16TOF32 [0x00A000]
MOV        @1, #0xFFFC  ; [0x00A001] = -4.0 (0xFFFC)
                        ; <--I16TOF32 complete, R0H = 4.0 (0x40800000)
I16TOF32   R1H, @1      ; R1H = I16TOF32 [0x00A001]
NOP                     ; 1 Cycle delay for I16TOF32 to complete
                        ; <-- I16TOF32 complete, R1H = -4.0 (0xC0800000)
```

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
UI16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

I32TOF32 RaH, mem32 *Convert 32-bit Integer to 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
mem32	32-bit source for the MOV32 operation. mem32 means that the operation can only address memory using any of the direct or indirect addressing modes supported by the C28x CPU

Opcode

```
LSW: 1110 0010  1000 1000
MSW: 0000 0aaa    mem32
```

Description

Convert the 32-bit signed integer indicated by the mem32 pointer to a 32-bit floating point value and store the result in RaH.

```
RaH = I32ToF32[mem32]
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
I32TOF32  RaH, mem32     ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- I32TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVW      DP, #0x0280  ; DP = 0x0280
MOV       @0, #0x1111  ; [0x00A000] = 4369 (0x1111)
MOV       @1, #0x1111  ; [0x00A001] = 4369 (0x1111)
                       ; Value of the 32 bit signed integer present in
                       ; 0x00A001 and 0x00A000 is +286331153 (0x11111111)
I32TOF32  R1H, @0      ; R1H = I32TOF32 (0x11111111)
NOP                    ; 1 Cycle delay for I32TOF32 to complete
                       ; <-- I32TOF32 complete, R1H = 286331153 (0x4D888888)
```

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, RbH
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32

I32TOF32 RaH, RbH *Convert 32-bit Integer to 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1001
MSW: 0000 0000  00bb baaa
```

Description

Convert the signed 32-bit integer in RbH to a 32-bit floating-point value and store the result in RaH.

```
RaH = I32ToF32(RbH)
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
I32TOF32  RaH, RbH       ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- I32TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

```
MOVIZ     R2H, #0x1111  ; R2H[31:16] = 4369 (0x1111)
MOVXI     R2H, #0x1111  ; R2H[15:0]  = 4369 (0x1111)
                        ; Value of the 32 bit signed integer present
                        ; in R2H is +286331153 (0x11111111)
I32TOF32  R3H, R2H      ; R3H = I32TOF32 (R2H)
NOP                     ; 1 Cycle delay for I32TOF32 to complete
                        ; <-- I32TOF32 complete, R3H = 286331153 (0x4D888888)
```

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
UI32TOF32 RaH, RbH
UI32TOF32 RaH, mem32

MACF32 R3H, R2H, RdH, ReH, RfH *32-bit Floating-Point Multiply with Parallel Add*

Operands

This instruction is an alias for the parallel multiply and add instruction. The operands are translated by the assembler such that the instruction becomes:

```
    MPYF32  RdH, RaH, RbH
|| ADDF32  R3H, R3H, R2H
```

R3H	floating-point destination and source register for the ADDF32
R2H	floating-point source register for the ADDF32 operation (R0H to R7H)
RdH	floating-point destination register for MPYF32 operation (R0H to R7H) RdH cannot be R3H
ReH	floating-point source register for MPYF32 operation (R0H to R7H)
RfH	floating-point source register for MPYF32 operation (R0H to R7H)

Opcode

```
LSW: 1110 0111  0100 00ff
MSW: feee dddc  ccbb baaa
```

Description

This instruction is an alias for the parallel multiply and add, MACF32 || ADDF32, instruction.

```
RdH = ReH * RfH
R3H = R3H + R2H
```

Restrictions

The destination register for the MPYF32 and the ADDF32 must be unique. That is, RdH cannot be R3H.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline

Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

```
    MPYF32  RaH, RbH, RcH   ; 2 pipeline cycles (2p)
|| ADDF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
    NOP                    ; 1 cycle delay or non-conflicting instruction
                           ; <-- MPYF32, ADDF32 complete, RaH, RdH updated
    NOP
```

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.

Example

```
;   Perform 5 multiply and accumulate operations:
;
;   1st multiply: A = X0 * Y0
;   2nd multiply: B = X1 * Y1
;   3rd multiply: C = X2 * Y2
;   4th multiply: D = X3 * Y3
;   5th multiply: E = X3 * Y3
;
;   Result = A + B + C + D + E

    MOV32  R0H, *XAR4++            ; R0H = X0
    MOV32  R1H, *XAR5++            ; R1H = Y0
                                   ; R2H = A = X0 * Y0
    MPYF32 R2H, R0H, R1H           ; In parallel R0H = X1
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y1
                                   ; R3H = B = X1 * Y1
    MPYF32 R3H, R0H, R1H           ; In parallel R0H = X2
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y2
                                    ; R3H = A + B
                                   ; R2H = C = X2 * Y2
    MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y3
                                   ; R3H = (A + B) + C
                                   ; R2H = D = X3 * Y3
    MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
 || MOV32  R0H, *XAR4
    MOV32  R1H, *XAR5              ; R1H = Y4

; The next MACF32 is an alias for
; MPYF32 || ADDF32
                                   ; R2H = E = X4 * Y4
    MACF32 R3H, R2H, R2H, R0H, R1H ; in parallel R3H = (A + B + C) + D
    NOP                            ; Wait for MPYF32 || ADDF32 to complete

    ADDF32 R3H, R3H, R2H           ; R3H = (A + B + C + D) + E
    NOP                            ; Wait for ADDF32 to complete
    MOV32  @Result, R3H            ; Store the result
```

See also

MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

MACF32 R3H, R2H, RdH, ReH, RfH
||MOV32 RaH, mem32 *32-bit Floating-Point Multiply and Accumulate with Parallel Move*

Operands

R3H	floating-point destination/source register R3H for the add operation
R2H	floating-point source register R2H for the add operation
RdH	floating-point destination register (R0H to R7H) for the multiply operation RdH cannot be the same register as RaH
ReH	floating-point source register (R0H to R7H) for the multiply operation
RfH	floating-point source register (R0H to R7H) for the multiply operation
RaH	floating-point destination register for the MOV32 operation (R0H to R7H). RaH cannot be R3H or the same register as RdH.
mem32	32-bit source for the MOV32 operation

Opcode

```
LSW: 1110 0011  0011 fffe
MSW: eedd daaa    mem32
```

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF32.

```
R3H = R3H + R2H,
RdH = ReH * RfH,
RaH = [mem32]
```

Restrictions

The destination registers for the MACF32 and the MOV32 must be unique. That is, RaH cannot be R3H and RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MACF32 (add or multiply) generates an underflow condition.
- LVF = 1 if MACF32 (add or multiply) generates an overflow condition.

MOV32 sets the NF, ZF, NI and ZI flags as follows:

```
NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;
```

Pipeline

The MACF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

```
    MACF32  R3H, R2H, RdH, ReH, RfH   ; 2 pipeline cycles (2p)
 || MOV32   RaH, mem32                ; 1 cycle
                                      ; <-- MOV32 completes, RaH updated
    NOP                               ; 1 cycle delay for MACF32
                                      ; <-- MACF32 completes, R3H, RdH updated
    NOP
```

Any instruction in the delay slot for this version of MACF32 must not use R3H or RdH as a destination register or R3H or RdH as a source operand.

Example

```
;   Perform 5 multiply and accumulate operations:
;
;   1ST multiply: A = X0 * Y0
;   2nd multiply: B = X1 * Y1
;   3rd multiply: C = X2 * Y2
;   4TH multiply: D = X3 * Y3
;   5th multiply: E = X3 * Y3
;
;   Result = A + B + C + D + E

    MOV32  R0H, *XAR4++            ; R0H = X0
    MOV32  R1H, *XAR5++            ; R1H = Y0

                                   ; R2H = A = X0 * Y0
    MPYF32 R2H, R0H, R1H           ; In parallel R0H = X1
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y1

                                   ; R3H = B = X1 * Y1
    MPYF32 R3H, R0H, R1H           ; In parallel R0H = X2
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y2

                                   ; R3H = A + B
                                   ; R2H = C = X2 * Y2
    MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y3

                                   ; R3H = (A + B) + C
                                   ; R2H = D = X3 * Y3
    MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
||  MOV32  R0H, *XAR4
    MOV32  R1H, *XAR5              ; R1H = Y4

                                   ; R2H = E = X4 * Y4
    MPYF32 R2H, R0H, R1H           ; in parallel R3H = (A + B + C) + D
 || ADDF32 R3H, R3H, R2H
    NOP                            ; Wait for MPYF32 || ADDF32 to complete

    ADDF32 R3H, R3H, R2H           ; R3H = (A + B + C + D) + E
    NOP                            ; Wait for ADDF32 to complete
    MOV32  @Result, R3H            ; Store the result
```

See also

MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

MACF32 R7H, R3H, mem32, *XAR7++ *32-bit Floating-Point Multiply and Accumulate*

Operands

R7H	floating-point destination register
R3H	floating-point destination register
mem32	pointer to a 32-bit source location
*XAR7	32-bit location pointed to by auxiliary register 7

Opcode

```
LSW: 1110 0010  0101 0000
MSW: 00bb baaa    mem32
```

Description

Perform an multiply and accumulate operation. When used as a stand-alone operation, the MACF32 will perform a single multiply as shown below:

```
Cycle 1: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
```

This instruction is the only floating-point instruction that can be repeated using the single repeat instruction (RPT ||). When repeated, the destination of the accumulate will alternate between R3H and R7H on each cycle and R2H and R6H are used as temporary storage for each multiply.

```
Cycle 1: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
Cycle 2: R7H = R7H + R6H, R6H = [mem32] * [XAR7++]

Cycle 3: R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
Cycle 4: R7H = R7H + R6H, R6H = [mem32] * [XAR7++]
etc...
```

Restrictions

R2H and R6H will be used as temporary storage by this instruction.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MACF32 generates an underflow condition.
- LVF = 1 if MACF32 generates an overflow condition.

Pipeline

When repeated the MACF32 takes 3 + N cycles where N is the number of times the instruction is repeated. When repeated, this instruction has the following pipeline restrictions:

```
   <instruction1>                      ; No restriction
   <instruction2>                      ; Cannot be a 2p instruction that writes
                                       ; to R2H, R3H, R6H or R7H
   RPT #(N-1)                          ; Execute N times, where N is even
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
   <instruction3>                      ; No restrictions.
                                       ; Can read R2H, R3H, R6H and R7H
```

MACF32 can also be used standalone. In this case, the insruction takes 2 cycles and the following pipeline restrictions apply:

```
   <instruction1>                       ; No restriction
   <instruction2>                       ; Cannot be a 2p instruction that writes
                                        ; to R2H, R3H, R6H or R7H
   MACF32 R7H, R3H, *XAR6, *XAR7        ; R3H = R3H + R2H, R2H = [mem32] * [XAR7++]
                                        ; <-- R2H and R3H are valid (note: no
delay required)
   NOP
```

Example

```
   ZERO R2H                           ; Zero the accumulation registers
   ZERO R3H                           ; and temporary multiply storage registers
   ZERO R6H
   ZERO R7H
   RPT  #3                            ; Repeat MACF32 N+1 (4) times
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
   ADDF32 R7H, R7H, R3H               ; Final accumulate
   NOP
                                      ; <-- ADDF32 completes, R7H valid
   NOP
```

Cascading of RPT || MACF32 is allowed as long as the first and subsequent counts are even. Cascading is useful for creating interruptible windows so that interrupts are not delayed too long by the RPT instruction. For example:

```
   ZERO R2H                           ; Zero the accumulation registers
   ZERO R3H                           ; and temporary multiply storage registers
   ZERO R6H
   ZERO R7H
   RPT  #3                            ; Execute MACF32 N+1 (4) times
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
   RPT  #5                            ; Execute MACF32 N+1 (6) times
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
   RPT  #N                            ; Repeat MACF32 N+1 times where N+1 is even
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
   ADDF32 R7H, R7H, R3H               ; Final accumulate
   NOP
                                      ; <-- ADDF32 completes, R7H valid
```

See also

MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

MACF32 R7H, R6H, RdH, ReH, RfH *32-bit Floating-Point Multiply with Parallel Add*

Operands

This instruction is an alias for the parallel multiply and add instruction. The operands are translated by the assembler such that the instruction becomes:

```
    MPYF32  RdH, RaH, RbH
|| ADDF32  R7H, R7H, R6H
```

R7H	floating-point destination and source register for the ADDF32
R6H	floating-point source register for the ADDF32 operation (R0H to R7H)
RdH	floating-point destination register for MPYF32 operation (R0H to R7H) RdH cannot be R3H
ReH	floating-point source register for MPYF32 operation (R0H to R7H)
RfH	floating-point source register for MPYF32 operation (R0H to R7H)

Opcode

```
LSW: 1110 0111  0100 00ff
MSW: feee dddc  ccbb baaa
```

Description

This instruction is an alias for the parallel multiply and add, MACF32 || ADDF32, instruction.

```
RdH = RaH * RbH
R7H = R6H + R6H
```

Restrictions

The destination register for the MPYF32 and the ADDF32 must be unique. That is, RdH cannot be R7H.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline

Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

```
    MPYF32  RaH, RbH, RcH   ; 2 pipeline cycles (2p)
|| ADDF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
    NOP                    ; 1 cycle delay or non-conflicting instruction
                           ; <-- MPYF32, ADDF32 complete, RaH, RdH updated
    NOP
```

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.

Example

```
;   Perform 5 multiply and accumulate operations:
;
;   1st multiply: A = X0 * Y0
;   2nd multiply: B = X1 * Y1
;   3rd multiply: C = X2 * Y2
;   4th multiply: D = X3 * Y3
;   5th multiply: E = X3 * Y3
;
;   Result = A + B + C + D + E

    MOV32  R0H, *XAR4++            ; R0H = X0
    MOV32  R1H, *XAR5++            ; R1H = Y0
                                   ; R6H = A = X0 * Y0
    MPYF32 R6H, R0H, R1H           ; In parallel R0H = X1
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y1
                                   ; R7H = B = X1 * Y1
    MPYF32 R7H, R0H, R1H           ; In parallel R0H = X2
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y2
                                   ; R7H = A + B
                                   ; R6H = C = X2 * Y2
    MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y3
                                   ; R7H = (A + B) + C
                                   ; R6H = D = X3 * Y3
    MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
||  MOV32  R0H, *XAR4
    MOV32  R1H, *XAR5              ; R1H = Y4

; Next MACF32 is an alias for
; MPYF32 || ADDF32

    MACF32 R7H, R6H, R6H, R0H, R1H ; R6H = E = X4 * Y4
                                   ; in parallel R7H = (A + B + C) + D
    NOP                            ; Wait for MPYF32 || ADDF32 to complete
    ADDF32 R7H, R7H, R6H           ; R7H = (A + B + C + D) + E
    NOP                            ; Wait for ADDF32 to complete
    MOV32  @Result, R7H            ; Store the result
```

See also

MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

MACF32 R7H, R6H, RdH, ReH, RfH
||MOV32 RaH, mem32 *32-bit Floating-Point Multiply and Accumulate with Parallel Move*

Operands

R7H	floating-point destination/source register R7H for the add operation
R6H	floating-point source register R6H for the add operation
RdH	floating-point destination register (R0H to R7H) for the multiply operation. RdH cannot be the same register as RaH.
ReH	floating-point source register (R0H to R7H) for the multiply operation
RfH	floating-point source register (R0H to R7H) for the multiply operation
RaH	floating-point destination register for the MOV32 operation (R0H to R7H). RaH cannot be R3H or the same as RdH.
mem32	32-bit source for the MOV32 operation

Opcode

```
LSW: 1110 0011  1100 fffe
MSW: eedd daaa    mem32
```

Description

Multiply/accumulate the contents of floating-point registers and move from register to memory. The destination register for the MOV32 cannot be the same as the destination registers for the MACF32.

```
R7H = R7H + R6H
RdH = ReH * RfH,
RaH = [mem32]
```

Restrictions

The destination registers for the MACF32 and the MOV32 must be unique. That is, RaH cannot be R7H and RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MACF32 (add or multiply) generates an underflow condition.
- LVF = 1 if MACF32 (add or multiply) generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

```
NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) {ZF = 1; NF = 0;}
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;
```

Pipeline

The MACF32 takes 2 pipeline cycles (2p) and the MOV32 takes a single cycle. That is:

```
    MACF32  R7H, R6H, RdH, ReH, RfH   ; 2 pipeline cycles (2p)
 || MOV32   RaH, mem32                ; 1 cycle
                                      ; <-- MOV32 completes, RaH updated
    NOP                               ; 1 cycle delay
                                      ; <-- MACF32 completes, R7H, RdH updated
    NOP
```

Example

```
;   Perform 5 multiply and accumulate operations:
;
;   1st multiply: A = X0 * Y0
;   2nd multiply: B = X1 * Y1
;   3rd multiply: C = X2 * Y2
;   4th multiply: D = X3 * Y3
;   5th multiply: E = X3 * Y3
;
;   Result = A + B + C + D + E

    MOV32  R0H, *XAR4++            ; R0H = X0
    MOV32  R1H, *XAR5++            ; R1H = Y0

                                   ; R6H = A = X0 * Y0
    MPYF32 R6H, R0H, R1H           ; In parallel R0H = X1
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y1

                                   ; R7H = B = X1 * Y1
    MPYF32 R7H, R0H, R1H           ; In parallel R0H = X2
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y2

                                   ; R7H = A + B
                                   ; R6H = C = X2 * Y2
    MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y3

                                   ; R7H = (A + B) + C
                                   ; R6H = D = X3 * Y3
    MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
||  MOV32  R0H, *XAR4
    MOV32  R1H, *XAR5              ; R1H = Y4

                                   ; R6H = E = X4 * Y4
    MPYF32 R6H, R0H, R1H           ; in parallel R7H = (A + B + C) + D
 || ADDF32 R7H, R7H, R6H
    NOP                            ; Wait for MPYF32 || ADDF32 to complete

    ADDF32 R7H, R7H, R6H           ; R7H = (A + B + C + D) + E
    NOP                            ; Wait for ADDF32 to complete
    MOV32  @Result, R7H            ; Store the result
```

See also

MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

MAXF32 RaH, RbH *32-bit Floating-Point Maximum*

Operands

RaH	floating-point source/destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 0110
MSW: 0000 0000  00bb baaa
```

Description

```
if(RaH < RbH) RaH = RbH
```

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == RbH) {ZF=1, NF=0}
if(RaH >  RbH) {ZF=0, NF=0}
if(RaH <  RbH) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
   MOVIZF32     R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   MOVIZF32     R1H, #-2.0    ; R1H = -2.0 (0xC0000000)
   MOVIZF32     R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)
   MAXF32       R2H, R1H      ; R2H = -1.5, ZF = NF = 0
   MAXF32       R1H, R2H      ; R1H = -1.5, ZF = 0, NF = 1
   MAXF32       R2H, R0H      ; R2H =  5.0, ZF = 0, NF = 1
   MAXF32       R0H, R2H      ; R2H =  5.0, ZF = 1, NF = 0
```

See also

CMPF32 RaH, RbH
CMPF32 RaH, #16FHi
CMPF32 RaH, #0.0
MAXF32 RaH, RbH || MOV32 RcH, RdH
MAXF32 RaH, #16FHi
MINF32 RaH, RbH
MINF32 RaH, #16FHi

MAXF32 RaH, #16FHi *32-bit Floating-Point Maximum*

Operands

RaH	floating-point source/destination register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  0010 0III
MSW: IIII IIII  IIII Iaaa
```

Description

Compare RaH with the floating-point value represented by the immediate operand. If the immediate value is larger, then load it into RaH.

```
if(RaH < #16FHi:0) RaH = #16FHi:0
```

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == #16FHi:0) {ZF=1, NF=0}
if(RaH >  #16FHi:0) {ZF=0, NF=0}
if(RaH <  #16FHi:0) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
   MOVIZF32     R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   MOVIZF32     R1H, #4.0     ; R1H =  4.0 (0x40800000)
   MOVIZF32     R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)
   MAXF32       R0H, #5.5     ; R0H =  5.5, ZF = 0, NF = 1
   MAXF32       R1H, #2.5     ; R1H =  4.0, ZF = 0, NF = 0
   MAXF32       R2H, #-1.0    ; R2H = -1.0, ZF = 0, NF = 1
   MAXF32       R2H, #-1.0    ; R2H = -1.5, ZF = 1, NF = 0
```

See also

MAXF32 RaH, RbH
MAXF32 RaH, RbH || MOV32 RcH, RdH
MINF32 RaH, RbH
MINF32 RaH, #16FHi

MAXF32 RaH, RbH
||MOV32 RcH, RdH *32-bit Floating-Point Maximum with Parallel Move*

Operands

RaH	floating-point source/destination register for the MAXF32 operation (R0H to R7H) RaH cannot be the same register as RcH
RbH	floating-point source register for the MAXF32 operation (R0H to R7H)
RcH	floating-point destination register for the MOV32 operation (R0H to R7H) RcH cannot be the same register as RaH
RdH	floating-point source register for the MOV32 operation (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 1100
MSW: 0000 dddc  ccbb baaa
```

Description

If RaH is less than RbH, then load RaH with RbH. Thus RaH will always have the maximum value. If RaH is less than RbH, then, in parallel, also load RcH with the contents of RdH.

```
if(RaH < RbH) { RaH = RbH; RcH = RdH; }
```

The MAXF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for the output from the MAXF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Restrictions

The destination register for the MAXF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RcH.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == RbH) {ZF=1, NF=0}
if(RaH >  RbH) {ZF=0, NF=0}
if(RaH <  RbH) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
    MOVIZF32    R0H, #5.0     ; R0H =  5.0 (0x40A00000)
    MOVIZF32    R1H, #4.0     ; R1H =  4.0 (0x40800000)
    MOVIZF32    R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)
    MOVIZF32    R3H, #-2.0    ; R3H = -2.0 (0xC0000000)
    MAXF32      R0H, R1H      ; R0H = 5.0, R3H = -1.5, ZF = 0, NF = 0
 || MOV32       R3H, R2H
    MAXF32      R1H, R0H      ; R1H = 5.0, R3H = -1.5, ZF = 0, NF = 1
 || MOV32       R3H, R2H
    MAXF32      R0H, R1H      ; R0H = 5.0, R2H = -1.5, ZF = 1, NF = 0
 || MOV32       R2H, R1H
```

See also

MAXF32 RaH, RbH
MAXF32 RaH, #16FHi

MINF32 RaH, RbH *32-bit Floating-Point Minimum*

Operands

RaH	floating-point source/destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 0111
MSW: 0000 0000  00bb baaa
```

Description

```
if(RaH > RbH) RaH = RbH
```

Special cases for the output from the MINF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == RbH) {ZF=1, NF=0}
if(RaH >  RbH) {ZF=0, NF=0}
if(RaH <  RbH) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
   MOVIZF32     R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   MOVIZF32     R1H, #4.0     ; R1H =  4.0 (0x40800000)
   MOVIZF32     R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)
   MINF32       R0H, R1H      ; R0H =  4.0, ZF = 0, NF = 0
   MINF32       R1H, R2H      ; R1H = -1.5, ZF = 0, NF = 0
   MINF32       R2H, R1H      ; R2H = -1.5, ZF = 1, NF = 0
   MINF32       R1H, R0H      ; R2H = -1.5, ZF = 0, NF = 1
```

See also

MAXF32 RaH, RbH
MAXF32 RaH, #16FHi
MINF32 RaH, #16FHi
MINF32 RaH, RbH || MOV32 RcH, RdH

MINF32 RaH, #16FHi *32-bit Floating-Point Minimum*

Operands

RaH	floating-point source/destination register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  0011 0III
MSW: IIII IIII  IIII Iaaa
```

Description

Compare RaH with the floating-point value represented by the immediate operand. If the immidate value is smaller, then load it into RaH.

```
if(RaH > #16FHi:0) RaH = #16FHi:0
```

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

Special cases for the output from the MINF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == #16FHi:0) {ZF=1, NF=0}
if(RaH >  #16FHi:0) {ZF=0, NF=0}
if(RaH <  #16FHi:0) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
   MOVIZF32     R0H, #5.0     ; R0H =  5.0 (0x40A00000)
   MOVIZF32     R1H, #4.0     ; R1H =  4.0 (0x40800000)
   MOVIZF32     R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)
   MINF32       R0H, #5.5     ; R0H =  5.0, ZF = 0, NF = 1
   MINF32       R1H, #2.5     ; R1H =  2.5, ZF = 0, NF = 0
   MINF32       R2H, #-1.0    ; R2H = -1.5, ZF = 0, NF = 1
   MINF32       R2H, #-1.5    ; R2H = -1.5, ZF = 1, NF = 0
```

See also

MAXF32 RaH, #16FHi
MAXF32 RaH, RbH
MINF32 RaH, RbH
MINF32 RaH, RbH || MOV32 RcH, RdH

MINF32 RaH, RbH
||MOV32 RcH, RdH *32-bit Floating-Point Minimum with Parallel Move*

Operands

RaH	floating-point source/destination register for the MIN32 operation (R0H to R7H) RaH cannot be the same register as RcH
RbH	floating-point source register for the MIN32 operation (R0H to R7H)
RcH	floating-point destination register for the MOV32 operation (R0H to R7H) RcH cannot be the same register as RaH
RdH	floating-point source register for the MOV32 operation (R0H to R7H)

Opcode

```
LSW: 1110 0110  1001 1101
MSW: 0000 dddc  ccbb baaa
```

Description

```
if(RaH > RbH) { RaH = RbH; RcH = RdH; }
```

Special cases for the output from the MINF32 operation:

- NaN output will be converted to infinity
- A denormalized output will be converted to positive zero.

Restrictions

The destination register for the MINF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RcH.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

The ZF and NF flags are configured on the result of the operation, not the result stored in the destination register.

```
if(RaH == RbH) {ZF=1, NF=0}
if(RaH >  RbH) {ZF=0, NF=0}
if(RaH <  RbH) {ZF=0, NF=1}
```

Pipeline

This is a single-cycle instruction.

Example

```
    MOVIZF32     R0H, #5.0     ; R0H =  5.0 (0x40A00000)
    MOVIZF32     R1H, #4.0     ; R1H =  4.0 (0x40800000)
    MOVIZF32     R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)
    MOVIZF32     R3H, #-2.0    ; R3H = -2.0 (0xC0000000)
    MINF32       R0H, R1H      ; R0H = 4.0, R3H = -1.5, ZF = 0, NF = 0
 || MOV32        R3H, R2H
    MINF32       R1H, R0H      ; R1H = 4.0, R3H = -1.5, ZF = 1, NF = 0
 || MOV32        R3H, R2H
    MINF32       R2H, R1H      ; R2H = -1.5, R1H = 4.0, ZF = 1, NF = 1
 || MOV32        R1H, R3H
```

See also

MINF32 RaH, RbH
MINF32 RaH, #16FHi

MOV16 mem16, RaH *Move 16-bit Floating-Point Register Contents to Memory*

Operands

mem16	points to the 16-bit destination memory
RaH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0010  0001 0011
MSW: 0000 0aaa    mem16
```

Description

Move 16-bit value from the lower 16-bits of the floating-point register (RaH[15:0]) to the location pointed to by mem16.

```
[mem16] = RaH[15:0]
```

Flags

No flags STF flags are affected.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a single-cycle instruction.

Example

```
    MOVW    DP, #0x02C0  ; DP = 0x02C0
    MOVXI   R4H,#0x0003  ; R4H = 3.0 (0x0003)
    MOV16   @0, R4H      ; [0x00B000] = 3.0 (0x0003)
```

See also

MOVIZ RaH, #16FHiHex
MOVIZF32 RaH, #16FHi
MOVXI RaH, #16FLoHex

MOV32 *(0:16bitAddr), loc32 *Move the Contents of loc32 to Memory*

Operands

0:16bitAddr	16-bit immediate address, zero extended
loc32	32 bit source location

Opcode

```
LSW: 1011 1101    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in loc32 to the memory location addressed by 0:16bitAddr. The EALLOW bit in the ST1 register is ignored by this operation.

```
[0:16bitAddr] = [loc32]
```

Flags

This instruction does not modify any STF register flags.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a two-cycle instruction.

Example

```
    MOVIZ   R5H, #0x1234    ; R5H[31:16] = 0x1234
    MOVXI   R5H, #0xABCD    ; R5H[15:0]  = 0xABCD
    NOP                     ; 1 Alignment Cycle
    MOV32   ACC, R5H        ; ACC = 0x1234ABCD
    MOV32   *(0xA000), @ACC ; [0x00A000] = ACC
    NOP                     ; 1 Cycle delay for MOV32 to complete
                            ; <-- MOV32 *(0:16bitAddr), loc32 complete,
                            ; [0x00A000] = 0xABCD, [0x00A001] = 0x1234
```

See also

MOV32 mem32, RaH
MOV32 mem32, STF
MOV32 loc32, *(0:16bitAddr)

MOV32 ACC, RaH *Move 32-bit Floating-Point Register Contents to ACC*

Operands

ACC	28x accumulator
RaH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1011 1111    loc32
MSW: IIII IIII  IIII IIII
```

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

```
ACC = RaH
```

Flags

No STF flags are affected.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Z and N flag in status register zero (ST0) of the 28x CPU are affected.

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H        ; Single-cycle instruction
NOP                   ; 1 alignment cycle
MOV32  @ACC,R0H       ; Copy R0H to ACC
NOP                   ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H  ; 2 pipeline instruction (2p)
NOP                   ; 1 cycle delay for ADDF32 to complete
                      ; <-- ADDF32 completes, R2H is valid
NOP                   ; 1 alignment cycle
MOV32  ACC, R2H       ; copy R2H into ACC, takes 2 cycles
                      ; <-- MOV32 completes, ACC is valid
NOP                   ; Any instruction
```

Example

```
MOVIZF32  R0H, #2.5   ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP                   ; Delay for conversion instruction
                      ; <-- Conversion complete, R0H valid
NOP                   ; Alignment cycle
MOV32     P, R0H      ; P = 2 = 0x00000002
```

See also

MOV32 P, RaH
MOV32 XARn, RaH
MOV32 XT, RaH

MOV32 loc32, *(0:16bitAddr) *Move 32-bit Value from Memory to loc32*

Operands

loc32	destination location
0:16bitAddr	16-bit address of the 32-bit source value

Opcode

```
LSW: 1011 1111    loc32
MSW: IIII IIII  IIII IIII
```

Description

Copy the 32-bit value referenced by 0:16bitAddr to the location indicated by loc32.

```
[loc32] = [0:16bitAddr]
```

Flags

No STF flags are affected. If loc32 is the ACC register, then the Z and N flag in status register zero (ST0) of the 28x CPU are affected.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 cycle instruction.

Example

```
MOVW   DP, #0x0300     ; DP = 0x0300
MOV    @0, #0xFFFF     ; [0x00C000] = 0xFFFF;
MOV    @1, #0x1111     ; [0x00C001] = 0x1111;
MOV32  @ACC, *(0xC000) ; AL = [0x00C000], AH = [0x00C001]
NOP                    ; 1 Cycle delay for MOV32 to complete
                       ; <-- MOV32 complete,  AL = 0xFFFF, AH = 0x1111
```

See also

MOV32 RaH, mem32{, CNDF}
MOV32 *(0:16bitAddr), loc32
MOV32 STF, mem32
MOVD32 RaH, mem32

MOV32 mem32, RaH *Move 32-bit Floating-Point Register Contents to Memory*

Operands

RaH	floating-point register (R0H to R7H)
mem32	points to the 32-bit destination memory

Opcode

```
LSW: 1110 0010  0000 0011
MSW: 0000 0aaa    mem32
```

Description

Move from memory to STF.

```
[mem32] = RaH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

No flags affected.

Pipeline

This is a single-cycle instruction.

Example

```
;   Perform 5 multiply and accumulate operations:
;
;   1st multiply: A = X0 * Y0
;   2nd multiply: B = X1 * Y1
;   3rd multiply: C = X2 * Y2
;   4th multiply: D = X3 * Y3
;   5th multiply: E = X3 * Y3
;
;   Result = A + B + C + D + E

    MOV32  R0H, *XAR4++             ; R0H = X0
    MOV32  R1H, *XAR5++             ; R1H = Y0

                                    ; R6H = A = X0 * Y0
    MPYF32 R6H, R0H, R1H            ; In parallel R0H = X1
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++             ; R1H = Y1

                                    ; R7H = B = X1 * Y1
    MPYF32 R7H, R0H, R1H            ; In parallel R0H = X2
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++             ; R1H = Y2

                                    ; R7H = A + B
                                    ; R6H = C = X2 * Y2
    MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X3
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++             ; R1H = Y3

                                    ; R3H = (A + B) + C
                                    ; R6H = D = X3 * Y3
    MACF32 R7H, R6H, R6H, R0H, R1H ; In parallel R0H = X4
||  MOV32  R0H, *XAR4
    MOV32  R1H, *XAR5               ; R1H = Y4

                                    ; R6H = E = X4 * Y4
    MPYF32 R6H, R0H, R1H            ; in parallel R7H = (A + B + C) + D
 || ADDF32 R7H, R7H, R2H
    NOP                             ; Wait for MPYF32 || ADDF32 to complete

    ADDF32 R7H, R7H, R6H            ; R7H = (A + B + C + D) + E
    NOP                             ; Wait for ADDF32 to complete
    MOV32  @Result, R7H             ; Store the result
```

See also

MOV32 *(0:16bitAddr), loc32
MOV32 mem32, STF

MOV32 mem32, STF *Move 32-bit STF Register to Memory*

Operands

STF	floating-point status register
mem32	points to the 32-bit destination memory

Opcode

```
LSW: 1110 0010  0000 0000
MSW: 0000 0000    mem32
```

Description

Copy the floating-point status register, STF, to memory.

```
[mem32] = STF
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

No flags affected.

Pipeline

This is a single-cycle instruction.

Example 1

```
MOVW      DP, #0x0280  ; DP = 0x0280
MOVIZF32  R0H, #2.0    ; R0H = 2.0 (0x40000000)
MOVIZF32  R1H, #3.0    ; R1H = 3.0 (0x40400000)
CMPF32    R0H, R1H     ; ZF = 0, NF = 1, STF = 0x00000004
MOV32     @0, STF      ; [0x00A000] = 0x00000004
```

Example 2

```
MOV32    *SP++, STF   ; Store STF in stack
MOVF32   R2H, #3.0    ; R2H = 3.0 (0x40400000)
MOVF32   R3H, #5.0    ; R3H = 5.0 (0x40A00000)
CMPF32   R2H, R3H     ; ZF = 0, NF = 1, STF = 0x00000004
MOV32    R3H, R2H, LT ; R3H = 3.0 (0x40400000)
MOV32    STF, *--SP   ; Restore STF from stack
```

See also

MOV32 mem32, RaH
MOV32 *(0:16bitAddr), loc32
MOVST0 FLAG

MOV32 P, RaH *Move 32-bit Floating-Point Register Contents to P*

Operands

P	28x product register P
RaH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1011 1111    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in RaH to the 28x product register P.

```
P = RaH
```

Flags

No flags affected in floating-point unit.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H        ; Single-cycle instruction
NOP                   ; 1 alignment cycle
MOV32  @ACC,R0H       ; Copy R0H to ACC
NOP                   ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H  ; 2 pipeline instruction (2p)
NOP                   ; 1 cycle delay for ADDF32 to complete
                      ; <-- ADDF32 completes, R2H is valid
NOP                   ; 1 alignment cycle
MOV32  ACC, R2H       ; copy R2H into ACC, takes 1 cycle
                      ; <-- MOV32 completes, ACC is valid
NOP                   ; Any instruction
```

Example

```
MOVIZF32  R0H, #2.5   ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP                   ; Delay for conversion instruction
                      ; <-- Conversion complete, R0H valid
NOP                   ; Alignment cycle
MOV32     P, R0H      ; P = 2 = 0x00000002
```

See also

MOV32 ACC, RaH
MOV32 XARn, RaH
MOV32 XT, RaH

MOV32 RaH, ACC *Move the Contents of ACC to a 32-bit Floating-Point Register*

Operands

RaH	floating-point destination register (R0H to R7H)
ACC	accumulator

Opcode

```
LSW: 1011 1101    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in ACC to the floating-point register RaH.

```
RaH = ACC
```

Flags

This instruction does not modify any STF register flags.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

```
MOV32   R0H,@ACC          ; Copy ACC to R0H
NOP                       ; Wait 4 cycles
NOP                       ; Do not use FRACF32, UI16TOF32
NOP                       ; I16TOF32, F32TOUI32 or F32TOI32
NOP                       ;
                          ; <-- R0H is valid
```

Example

```
MOV    AH, #0x0000
MOV    AL, #0x0200        ; ACC = 512
MOV32  R0H, ACC
NOP
NOP
NOP
NOP    UI32TOF32 R0H, R0H ; R0H = 512.0 (0x44000000)
```

See also

MOV32 RaH, P
MOV32 RaH, XARn
MOV32 RaH, XT

MOV32 RaH, mem32 {, CNDF} *Conditional 32-bit Move*

Operands

RaH	floating-point destination register (R0H to R7H)
mem32	pointer to the 32-bit source memory location
CNDF	optional condition.

Opcode

```
LSW: 1110 0010  1010 CNDF
MSW: 0000 0aaa    mem32
```

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

```
if (CNDF == TRUE) RaH = [mem32]
```

CNDF is one of the following conditions:

Encode [1]	CNDF	Description	STF Flags Tested
0000	NEQ	Not equal to zero	ZF == 0
0001	EQ	Equal to zero	ZF == 1
0010	GT	Greater than zero	ZF == 0 AND NF == 0
0011	GEQ	Greater than or equal to zero	NF == 0
0100	LT	Less than zero	NF == 1
0101	LEQ	Less than or equal to zero	ZF == 1 AND NF == 1
1010	TF	Test flag set	TF == 1
1011	NTF	Test flag not set	TF == 0
1100	LU	Latched underflow	LUF == 1
1101	LV	Latched overflow	LVF == 1
1110	UNC	Unconditional	None
1111	UNCF [2]	Unconditional with flag modification	None

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	No	No

```
if(CNDF == UNCF)
{
   NF = RaH(31);
   ZF = 0;
   if(RaH[30:23] == 0) { ZF = 1; NF = 0; }
   NI = RaH[31];
   ZI = 0;
   if(RaH[31:0] == 0) ZI = 1;
}
else No flags modified;
```

Pipeline

This is a single-cycle instruction.

Example

```
MOVW      DP, #0x0300  ; DP = 0x0300
MOV       @0, #0x5555  ; [0x00C000] = 0x5555
MOV       @1, #0x5555  ; [0x00C001] = 0x5555
MOVIZF32  R3H, #7.0    ; R3H = 7.0 (0x40E00000)
MOVIZF32  R4H, #7.0    ; R4H = 7.0 (0x40E00000)
MAXF32    R3H, R4H     ; ZF = 1, NF = 0
MOV32     R1H, @0, EQ  ; R1H = 0x55555555
```

See also

MOV32 RaH, RbH{, CNDF}
MOVD32 RaH, mem32

MOV32 RaH, P — *Move the Contents of P to a 32-bit Floating-Point Register*

Operands

RaH	floating-point register (R0H to R7H)
P	product register

Opcode

```
LSW: 1011 1101    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in the product register, P, to the floating-point register RaH.

```
RaH = P
```

Flags

This instruction does not modify any STF register flags.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

```
MOV32   R0H,@P          ; Copy P to R0H
NOP                     ; Wait 4 alignment cycles
NOP                     ; Do not use FRACF32, UI16TOF32
NOP                     ; I16TOF32, F32TOUI32 or F32TOI32
NOP                     ;
                        ; <-- R0H is valid
                        ; Instruction can use R0H as a source
```

Example

```
MOV    PH, #0x0000
MOV    PL, #0x0200      ; P = 512
MOV32  R0H, P
NOP
NOP
NOP
NOP
UI32TOF32 R0H, R0H      ; R0H = 512.0 (0x44000000)
```

See also

MOV32 RaH, ACC
MOV32 RaH, XARn
MOV32 RaH, XT

MOV32 RaH, RbH {, CNDF} *Conditional 32-bit Move*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)
CNDF	optional condition.

Opcode

```
LSW: 1110 0110  1100 CNDF
MSW: 0000 0000  00bb baaa
```

Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

```
if (CNDF == TRUE) RaH = RbH
```

CNDF is one of the following conditions:

Encode [1]	CNDF	Description	STF Flags Tested
0000	NEQ	Not equal to zero	ZF == 0
0001	EQ	Equal to zero	ZF == 1
0010	GT	Greater than zero	ZF == 0 AND NF == 0
0011	GEQ	Greater than or equal to zero	NF == 0
0100	LT	Less than zero	NF == 1
0101	LEQ	Less than or equal to zero	ZF == 1 AND NF == 1
1010	TF	Test flag set	TF == 1
1011	NTF	Test flag not set	TF == 0
1100	LU	Latched underflow	LUF == 1
1101	LV	Latched overflow	LVF == 1
1110	UNC	Unconditional	None
1111	UNCF [2]	Unconditional with flag modification	None

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	No	No

```
if(CNDF == UNCF)
{
   NF = RaH(31); ZF = 0;
   if(RaH[30:23] == 0) {ZF = 1; NF = 0;}
   NI = RaH(31); ZI = 0;
   if(RaH[31:0] == 0) ZI = 1;
}
else No flags modified;
```

Pipeline

This is a single-cycle instruction.

Example

```
    MOVIZF32  R3H, #8.0     ; R3H = 8.0 (0x41000000)
    MOVIZF32  R4H, #7.0     ; R4H = 7.0 (0x40E00000)
    MAXF32    R3H, R4H      ; ZF = 0, NF = 0
    MOV32     R1H, R3H, GT  ; R1H = 8.0 (0x41000000)
```

See also

MOV32 RaH, mem32{, CNDF}

MOV32 RaH, XARn *Move the Contents of XARn to a 32-bit Floating-Point Register*

Operands

RaH	floating-point register (R0H to R7H)
XARn	auxiliary register (XAR0 - XAR7)

Opcode

```
LSW: 1011 1101    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in the auxiliary register XARn to the floating point register RaH.

```
RaH = XARn
```

Flags

This instruction does not modify any STF register flags.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

```
MOV32   R0H,@XAR7         ; Copy XAR7 to R0H
NOP                       ; Wait 4 alignment cycles
NOP                       ; Do not use FRACF32, UI16TOF32
NOP                       ; I16TOF32, F32TOUI32 or F32TOI32
NOP                       ;
                          ; <-- R0H is valid
ADDF32  R2H,R1H,R0H       ; Instruction can use R0H as a source
```

Example

```
MOVL   XAR1, #0x0200 ; XAR1 = 512
MOV32  R0H, XAR1
NOP
NOP
NOP
NOP
UI32TOF32 R0H, R0H  ; R0H = 512.0 (0x44000000)
```

See also

MOV32 RaH, ACC
MOV32 RaH, P
MOV32 RaH, XT

MOV32 RaH, XT *Move the Contents of XT to a 32-bit Floating-Point Register*

Operands

RaH	floating-point register (R0H to R7H)
XT	auxiliary register (XAR0 - XAR7)

Opcode

```
LSW: 1011 1101    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in temporary register, XT, to the floating-point register RaH.

```
RaH = XT
```

Flags

This instruction does not modify any STF register flags.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28x CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, UI16TOF32, I16TOF32, F32TOUI32, and F32TOI32.

```
MOV32   R0H, XT          ; Copy XT to R0H
NOP                      ; Wait 4 alignment cycles
NOP                      ; Do not use FRACF32, UI16TOF32
NOP                      ; I16TOF32, F32TOUI32 or F32TOI32
NOP                      ;
                         ; <-- R0H is valid
ADDF32  R2H,R1H,R0H      ; Instruction can use R0H as a source
```

Example

```
MOVIZF32  R6H, #5.0  ; R6H = 5.0 (0x40A00000)
NOP                  ; 1 Alignment cycle
MOV32     XT, R6H    ; XT = 5.0 (0x40A00000)
MOV32     R1H, XT    ; R1H = 5.0 (0x40A00000)
```

See also

MOV32 RaH, ACC
MOV32 RaH, P
MOV32 RaH, XARn

MOV32 STF, mem32 *Move 32-bit Value from Memory to the STF Register*

Operands

STF	floating-point unit status register
mem32	pointer to the 32-bit source memory location

Opcode

```
LSW: 1110 0010  1000 0000
MSW: 0000 0000    mem32
```

Description

Move from memory to the floating-point unit's status register STF.

```
STF = [mem32]
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Restoring status register will overwrite all flags.

Pipeline

This is a single-cycle instruction.

Example 1

```
MOVW   DP, #0x0300  ; DP = 0x0300
MOV    @2, #0x020C  ; [0x00C002] = 0x020C
MOV    @3, #0x0000  ; [0x00C003] = 0x0000
MOV32  STF, @2      ; STF = 0x0000020C
```

Example 2

```
MOV32    *SP++, STF   ; Store STF in stack
MOVF32   R2H, #3.0    ; R2H = 3.0 (0x40400000)
MOVF32   R3H, #5.0    ; R3H = 5.0 (0x40A00000)
CMPF32   R2H, R3H     ; ZF = 0, NF = 1, STF = 0x00000004
MOV32    R3H, R2H, LT ; R3H = 3.0 (0x40400000)
MOV32    STF, *--SP   ; Restore STF from stack
```

See also

MOV32 mem32, STF
MOVST0 FLAG

MOV32 XARn, RaH *Move 32-bit Floating-Point Register Contents to XARn*

Operands

XARn	28x auxiliary register (XAR0 - XAR7)
RaH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1011 1111    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value from the floating-point register RaH to the auxiliary register XARn.

```
XARn = RaH
```

Flags

No flags affected in floating-point unit.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H        ; Single-cycle instruction
NOP                   ; 1 alignment cycle
MOV32  @ACC,R0H       ; Copy R0H to ACC
NOP                   ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H  ; 2 pipeline instruction (2p)
NOP                   ; 1 cycle delay for ADDF32 to complete
                      ; <-- ADDF32 completes, R2H is valid
NOP                   ; 1 alignment cycle
MOV32  ACC, R2H       ; copy R2H into ACC, takes 1 cycle
                      ; <-- MOV32 completes, ACC is valid
NOP                   ; Any instruction
```

Example

```
MOVIZF32  R0H, #2.5   ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP                   ; Delay for conversion instruction
                      ; <-- Conversion complete, R0H valid
NOP                   ; Alignment cycle
MOV32     XAR0, R0H   ; XAR0 = 2 = 0x00000002
```

See also

MOV32 ACC, RaH
MOV32 P, RaH
MOV32 XT, RaH

MOV32 XT, RaH *Move 32-bit Floating-Point Register Contents to XT*

Operands

XT	temporary register
RaH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1011 1111    loc32
MSW: IIII IIII  IIII IIII
```

Description

Move the 32-bit value in RaH to the temporary register XT.

```
XT = RaH
```

Flags

No flags affected in floating-point unit.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

While this is a single-cycle instruction, additional pipeline alignment is required when copying a floating-point register to a C28x register. If the move follows a single cycle floating point instruction, a single alignment cycle must be added. For example:

```
MINF32 R0H,R1H        ; Single-cycle instruction
NOP                   ; 1 alignment cycle
MOV32  @XT,R0H        ; Copy R0H to ACC
NOP                   ; Any instruction
```

If the move follows a 2 pipeline-cycle floating point instruction, then two alignment cycles must be used. For example:

```
ADDF32 R2H, R1H, R0H  ; 2 pipeline instruction (2p)
NOP                   ; 1 cycle delay for ADDF32 to complete
                      ; <-- ADDF32 completes, R2H is valid
NOP                   ; 1 alignment cycle
MOV32  XT, R2H        ; copy R2H into ACC, takes 1 cycle
                      ; <-- MOV32 completes, ACC is valid
NOP                   ; Any instruction
```

Example

```
MOVIZF32  R0H, #2.5   ; R0H = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP                   ; Delay for conversion instruction
                      ; <-- Conversion complete, R0H valid
NOP                   ; Alignment cycle
MOV32     XT, R0H     ; XT = 2 = 0x00000002
```

See also

MOV32 ACC, RaH
MOV32 P, RaH
MOV32 XARn, RaH

MOVD32 RaH, mem32 *Move 32-bit Value from Memory with Data Copy*

Operands

RaH	floating-point register (R0H to R7H)
mem32	pointer to the 32-bit source memory location

Opcode

```
LSW: 1110 0010  0010 0011
MSW: 0000 0aaa    mem32
```

Description

Move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.

```
RaH = [mem32]
[mem32+2] = [mem32]
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	No	No

```
NF = RaH[31];
ZF = 0;
if(RaH[30:23] == 0){ ZF = 1; NF = 0; }
NI = RaH[31];
ZI = 0;
if(RaH[31:0] == 0) ZI = 1;
```

Pipeline

This is a single-cycle instruction.

Example

```
    MOVW    DP, #0x02C0  ; DP = 0x02C0
    MOV     @2, #0x0000  ; [0x00B002] = 0x0000
    MOV     @3, #0x4110  ; [0x00B003] = 0x4110
    MOVD32  R7H, @2      ; R7H = 0x41100000,
                         ; [0x00B004] = 0x0000, [0x00B005] = 0x4110
```

See also

MOV32 RaH, mem32 {,CNDF}

MOVF32 RaH, #32F *Load the 32-bits of a 32-bit Floating-Point Register*

Operands

This instruction is an alias for MOVIZ and MOVXI instructions. The second operand is translated by the assembler such that the instruction becomes:

```
MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
```

RaH	floating-point destination register (R0H to R7H)
#32F	immediate float value represented in floating-point representation

Opcode

```
LSW: 1110 1000 0000 0III (opcode of MOVIZ RaH, #16FHiHex)
MSW: IIII IIII IIII Iaaa

LSW: 1110 1000 0000 1III (opcode of MOVXI RaH, #16FLoHex)
MSW: IIII IIII IIII Iaaa
```

Description

Note: This instruction accepts the immediate operand only in floating-point representation. To specify the immediate value as a hex value (IEEE 32-bit floating-point format) use the MOVI32 RaH, #32FHex instruction.

Load the 32-bits of RaH with the immediate float value represented by #32F.

#32F is a float value represented in floating-point representation. The assembler will only accept a float value represented in floating-point representation. That is, 3.0 can only be represented as #3.0. #0x40400000 will result in an error.

```
RaH = #32F
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

Depending on #32FH, this instruction takes one or two cycles. If all of the lower 16-bits of the IEEE 32-bit floating-point format of #32F are zeros, then the assembler will convert MOVF32 into only MOVIZ instruction. If the lower 16-bits of the IEEE 32-bit floating-point format of #32F are not zeros, then the assembler will convert MOVF32 into MOVIZ and MOVXI instructions.

Example

```
MOVF32   R1H, #3.0      ; R1H = 3.0 (0x40400000)
                        ; Assembler converts this instruction as
                        ; MOVIZ R1H, #0x4040
MOVF32   R2H, #0.0      ; R2H = 0.0 (0x00000000)
                        ; Assembler converts this instruction as
                        ; MOVIZ R2H, #0x0
MOVF32   R3H, #12.265   ; R3H = 12.625 (0x41443D71)
                        ; Assembler converts this instruction as
                        ; MOVIZ R3H, #0x4144
                        ; MOVXI R3H, #0x3D71
```

See also

MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
MOVI32 RaH, #32FHex
MOVIZF32 RaH, #16FHi

MOVI32 RaH, #32FHex *Load the 32-bits of a 32-bit Floating-Point Register with the immediate*

Operands

This instruction is an alias for MOVIZ and MOVXI instructions. The second operand is translated by the assembler such that the instruction becomes:

```
MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
```

RaH	floating-point register (R0H to R7H)
#32FHex	A 32-bit immediate value that represents an IEEE 32-bit floating-point value.

Opcode

```
LSW: 1110 1000 0000 0III (opcode of MOVIZ RaH, #16FHiHex)
MSW: IIII IIII IIII Iaaa

LSW: 1110 1000 0000 1III (opcode of MOVXI RaH, #16FLoHex)
MSW: IIII IIII IIII Iaaa
```

Description

Note: This instruction only accepts a hex value as the immediate operand. To specify the immediate value with a floating-point representation use the MOVF32 RaH, #32F instruction.

Load the 32-bits of RaH with the immediate 32-bit hex value represented by #32Fhex.

#32Fhex is a 32-bit immediate hex value that represents the IEEE 32-bit floating-point value of a floating-point number. The assembler will only accept a hex immediate value. That is, 3.0 can only be represented as #0x40400000. #3.0 will result in an error.

```
RaH = #32FHex
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

Depending on #32FHex, this instruction takes one or two cycles. If all of the lower 16-bits of #32FHex are zeros, then assembler will convert MOVI32 to the MOVIZ instruction. If the lower 16-bits of #32FHex are not zeros, then assembler will convert MOVI32 to a MOVIZ and a MOVXI instruction.

Example

```
MOVI32   R1H, #0x40400000   ; R1H = 0x40400000
                            ; Assembler converts this instruction as
                            ; MOVIZ R1H, #0x4040
MOVI32   R2H, #0x00000000   ; R2H = 0x00000000
                            ; Assembler converts this instruction as
                            ; MOVIZ R2H, #0x0
MOVI32   R3H, #0x40004001   ; R3H = 0x40004001
                            ; Assembler converts this instruction as
                            ; MOVIZ R3H, #0x4000
                            ; MOVXI R3H, #0x4001
MOVI32   R4H, #0x00004040   ; R4H = 0x00004040
                            ; Assembler converts this instruction as
                            ; MOVIZ R4H, #0x0000
                            ; MOVXI R4H, #0x4040
```

See also

MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex
MOVF32 RaH, #32F
MOVIZF32 RaH, #16FHi

MOVIZ RaH, #16FHiHex *Load the Upper 16-bits of a 32-bit Floating-Point Register*

Operands

RaH	floating-point register (R0H to R7H)
#16FHiHex	A 16-bit immediate hex value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  0000 0III
MSW: IIII IIII  IIII Iaaa
```

Description

Note: This instruction only accepts a hex value as the immediate operand. To specify the immediate value with a floating-point representation use the MOVIZF32 pseudo instruction.

Load the upper 16-bits of RaH with the immediate value #16FHiHex and clear the low 16-bits of RaH.

#16FHiHex is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. The assembler will only accept a hex immediate value. That is, -1.5 can only be represented as #0xBFC0. #-1.5 will result in an error.

By itself, MOVIZ is useful for loading a floating-point register with a constant in which the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). If a constant requires all 32-bits of a floating-point register to be iniitalized, then use MOVIZ along with the MOVXI instruction.

```
RaH[31:16] = #16FHiHex
RaH[15:0] = 0
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a single-cycle instruction.

Example

```
; Load R0H with -1.5 (0xBFC00000)
    MOVIZ     R0H, #0xBFC0    ; R0H = 0xBFC00000

; Load R0H with pi = 3.141593 (0x40490FDB)
    MOVIZ     R0H, #0x4049    ; R0H = 0x40490000
    MOVXI     R0H, #0x0FDB    ; R0H = 0x40490FDB
```

See also

MOVIZF32 RaH, #16FHi
MOVXI RaH, #16FLoHex

MOVIZF32 RaH, #16FHi *Load the Upper 16-bits of a 32-bit Floating-Point Register*

Operands

RaH	floating-point register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  0000 0III
MSW: IIII IIII  IIII Iaaa
```

Description

Load the upper 16-bits of RaH with the value represented by #16FHi and clear the low 16-bits of RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. This addressing mode is most useful for constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). #16FHi can be specified in hex or float. That is, -1.5 can be represented as #-1.5 or #0xBFC0.

MOVIZF32 is an alias for the MOVIZ RaH, #16FHiHex instruction. In the case of MOVIZF32 the assembler will accept either a hex or float as the immediate value and encodes it into a MOVIZ instruction. For example, MOVIZF32 RaH, #-1.5 will be encoded as MOVIZ RaH, 0xBFC0.

```
RaH[31:16] = #16FHi
RaH[15:0] = 0
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a single-cycle instruction.

Example

```
    MOVIZF32 R0H, #3.0        ; R0H =  3.0 = 0x40400000
    MOVIZF32 R1H, #1.0        ; R1H =  1.0 = 0x3F800000
    MOVIZF32 R2H, #2.5        ; R2H =  2.5 = 0x40200000
    MOVIZF32 R3H, #-5.5       ; R3H = -5.5 = 0xC0B00000
    MOVIZF32 R4H, #0xC0B0     ; R4H = -5.5 = 0xC0B00000
;
; Load R5H with pi = 3.141593 (0x40490000)
;
    MOVIZF32 R5H, #3.141593   ; R5H =  3.140625 (0x40490000)
;
; Load R0H with a more accurate pi = 3.141593 (0x40490FDB)
;
    MOVIZF32  R0H,#0x4049     ; R0H = 0x40490000
    MOVXI     R0H,#0x0FDB     ; R0H = 0x40490FDB
```

See also

MOVIZ RaH, #16FHiHex
MOVXI RaH, #16FLoHex

MOVST0 FLAG *Load Selected STF Flags into ST0*

Operands

FLAG	Selected flag

Opcode

```
LSW: 1010 1101  FFFF FFFF
```

Description

Load selected flags from the STF register into the ST0 register of the 28x CPU where FLAG is one or more of TF, CI, ZI, ZF, NI, NF, LUF or LVF. The specified flag maps to the ST0 register as follows:

- Set OV = 1 if LVF or LUF is set. Otherwise clear OV.
- Set N = 1 if NF or NI is set. Otherwise clear N.
- Set Z = 1 if ZF or ZI is set. Otherwise clear Z.
- Set C = 1 if TF is set. Otherwise clear C.
- Set TC = 1 if TF is set. Otherwise clear TF.

If any STF flag is not specified, then the corresponding ST0 register bit is not modified.

Restrictions

Do not use the MOVST0 instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the MOVST0 operation.

```
; The following is INVALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   MOVST0  TF                ; INVALID, do not use MOVST0 in a delay slot


; The following is VALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   NOP                       ; 1 delay cycle, R2H updated after this instruction
   MOVST0  TF                ; VALID
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

When the flags are moved to the C28x ST0 register, the LUF or LVF flags are automatically cleared if selected.

Pipeline

This is a single-cycle instruction.

Example

Program flow is controlled by C28x instructions that read status flags in the status register 0 (ST0) . If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into ST0 flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVST0 FLAG instruction is used to load the current value of specified STF flags into the respective bits of ST0. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

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

See also

MOV32 mem32, STF
MOV32 STF, mem32

MOVXI RaH, #16FLoHex *Move Immediate to the Low 16-bits of a Floating-Point Register*

Operands

Ra	floating-point register (R0H to R7H)
#16FLoHex	A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits will not be modified.

Opcode

```
LSW: 1110 1000  0000 1III
MSW: IIII IIII  IIII Iaaa
```

Description

Load the low 16-bits of RaH with the immediate value #16FLoHex. #16FLoHex represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits of RaH will not be modified. MOVXI can be combined with the MOVIZ or MOVIZF32 instruction to initialize all 32-bits of a RaH register.

```
RaH[15:0] = #16FLoHex
RaH[31:16] = Unchanged
```

Flags

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a single-cycle instruction.

Example

```
; Load R0H with pi = 3.141593 (0x40490FDB)
    MOVIZ     R0H,#0x4049     ; R0H = 0x40490000
    MOVXI     R0H,#0x0FDB     ; R0H = 0x40490FDB
```

See also

MOVIZ RaH, #16FHiHex
MOVIZF32 RaH, #16FHi

MPYF32 RaH, RbH, RcH *32-bit Floating-Point Multiply*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)
RcH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0111  0000 0000
MSW: 0000 000c  ccbb baaa
```

Description

Multiply the contents of two floating-point registers.

```
RaH = RbH * RcH
```

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
MPYF32  RaH, RbH, RcH    ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- MPYF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example

Calculate Y = A * B:

```
MOVL   XAR4, #A
MOV32  R0H,  *XAR4  ; Load R0H with A
MOVL   XAR4, # B
MOV32  R1H,  *XAR4  ; Load R1H with B
MPYF32 R0H,R1H,R0H  ; Multiply A * B
MOVL   XAR4, #Y
                    ; <--MPYF32 complete
MOV32  *XAR4,R0H    ; Save the result
```

See also

MPYF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH
MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RdH, ReH, RfH || MOV32 mem32, RaH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32

MPYF32 RaH, #16FHi, RbH *32-bit Floating-Point Multiply*

Operands

RaH	floating-point destination register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.
RcH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 1000  01II IIII
MSW: IIII IIII  IIbb baaa
```

Description

Multiply RbH with the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

```
RaH = RbH * #16FHi:0
```

This instruction can also be written as MPYF32 RaH, RbH, #16FHi.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
    MPYF32  RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
    NOP                      ; 1 cycle delay or non-conflicting instruction
                             ; <-- MPYF32 completes, RaH updated
    NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example 1

```
    MOVIZF32  R3H, #2.0       ; R3H = 2.0 (0x40000000)
    MPYF32    R4H, #3.0, R3H  ; R4H = 3.0 * R3H
    MOVL      XAR1, #0xB006   ; <-- Non conflicting instruction
                              ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
    MOV32     *XAR1, R4H      ; Save the result in memory location 0xB006
```

Example 2

```
;Same as above example but #16FHi is represented in Hex
    MOVIZF32  R3H, #2.0          ; R3H = 2.0 (0x40000000)
    MPYF32    R4H, #0x4040, R3H  ; R4H = 0x4040 * R3H
                                 ; 3.0 is represented as 0x40400000 in
                                 ; IEEE 754 32-bit format
    MOVL      XAR1, #0xB006      ; <-- Non conflicting instruction
                                 ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
    MOV32     *XAR1, R4H         ; Save the result in memory location 0xB006
```

See also

MPYF32 RaH, RbH, #16FHi
MPYF32 RaH, RbH, RcH
MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH

MPYF32 RaH, RbH, #16FHi *32-bit Floating-Point Multiply*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.

Opcode

```
LSW: 1110 1000  01II IIII
MSW: IIII IIII  IIbb baaa
```

Description

Multiply RbH with the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

```
RaH = RbH * #16FHi:0
```

This instruction can also be writen as MPYF32 RaH, #16FHi, RbH.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
    MPYF32  RaH, RbH, #16FHi ; 2 pipeline cycles (2p)
    NOP                      ; 1 cycle delay or non-conflicting instruction
                             ; <-- MPYF32 completes, RaH updated
    NOP
```

Any instruction in the delay slot must not use RaH as a destination register or use RaH as a source operand.

Example 1

```
    MOVIZF32  R3H, #2.0       ; R3H = 2.0 (0x40000000)
    MPYF32    R4H, R3H, #3.0  ; R4H = R3H * 3.0
    MOVL      XAR1, #0xB008   ; <-- Non conflicting instruction
                              ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
    MOV32     *XAR1, R4H      ; Save the result in memory location 0xB008
```

Example 2

```
;Same as above example but #16FHi is represented in Hex
    MOVIZF32  R3H, #2.0          ; R3H = 2.0 (0x40000000)
    MPYF32    R4H, R3H, #0x4040  ; R4H = R3H * 0x4040
                                 ; 3.0 is represented as 0x40400000 in
                                 ; IEEE 754 32-bit format
    MOVL      XAR1, #0xB008      ; <-- Non conflicting instruction
                                 ; <-- MPYF32 complete, R4H = 6.0 (0x40C00000)
    MOV32     *XAR1, R4H         ; Save the result in memory location 0xB008
```

See also

MPYF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH

MPYF32 RaH, RbH, RcH
||ADDF32 RdH, ReH, RfH *32-bit Floating-Point Multiply with Parallel Add*

Operands

RaH	floating-point destination register for MPYF32 (R0H to R7H) RaH cannot be the same register as RdH
RbH	floating-point source register for MPYF32 (R0H to R7H)
RcH	floating-point source register for MPYF32 (R0H to R7H)
RdH	floating-point destination register for ADDF32 (R0H to R7H) RdH cannot be the same register as RaH
ReH	floating-point source register for ADDF32 (R0H to R7H)
RfH	floating-point source register for ADDF32 (R0H to R7H)

Opcode

```
LSW: 1110 0111  0100 00ff
MSW: feee dddc  ccbb baaa
```

Description

Multiply the contents of two floating-point registers with parallel addition of two registers.

```
RaH = RbH * RcH
RdH = ReH + RfH
```

This instruction can also be written as:

```
MACF32 RaH, RbH, RcH, RdH, ReH, RfH
```

Restrictions

The destination register for the MPYF32 and the ADDF32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or ADDF32 generates an underflow condition.
- LVF = 1 if MPYF32 or ADDF32 generates an overflow condition.

Pipeline

Both MPYF32 and ADDF32 take 2 pipeline cycles (2p) That is:

```
    MPYF32  RaH, RbH, RcH   ; 2 pipeline cycles (2p)
||  ADDF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- MPYF32, ADDF32 complete, RaH, RdH updated
    NOP
```

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.

Example

```
;   Perform 5 multiply and accumulate operations:
;
;   1st multiply: A = X0 * Y0
;   2nd multiply: B = X1 * Y1
;   3rd multiply: C = X2 * Y2
;   4th multiply: D = X3 * Y3
;   5th multiply: E = X3 * Y3
;
;   Result = A + B + C + D + E

    MOV32  R0H, *XAR4++            ; R0H = X0
    MOV32  R1H, *XAR5++            ; R1H = Y0

                                   ; R2H = A = X0 * Y0
    MPYF32 R2H, R0H, R1H           ; In parallel R0H = X1
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y1

                                   ; R3H = B = X1 * Y1
    MPYF32 R3H, R0H, R1H           ; In parallel R0H = X2
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y2

                                   ; R3H = A + B
                                   ; R2H = C = X2 * Y2
    MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X3
 || MOV32  R0H, *XAR4++
    MOV32  R1H, *XAR5++            ; R1H = Y3

                                   ; R3H = (A + B) + C
                                   ; R2H = D = X3 * Y3
    MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H = X4
||  MOV32  R0H, *XAR4
    MOV32  R1H, *XAR5              ; R1H = Y4

                                   ; R2H = E = X4 * Y4
    MPYF32 R2H, R0H, R1H           ; in parallel R3H = (A + B + C) + D
 || ADDF32 R3H, R3H, R2H
    NOP                            ; Wait for MPYF32 || ADDF32 to complete

    ADDF32 R3H, R3H, R2H           ; R3H = (A + B + C + D) + E
    NOP                            ; Wait for ADDF32 to complete
    MOV32  @Result, R3H            ; Store the result
```

See also

MACF32 R3H, R2H, RdH, ReH, RfH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++
MACF32 R7H, R6H, RdH, ReH, RfH
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32

MPYF32 RdH, ReH, RfH
‖MOV32 RaH, mem32 *32-bit Floating-Point Multiply with Parallel Move*

Operands

RdH	floating-point destination register for the MPYF32 (R0H to R7H) RdH cannot be the same register as RaH
ReH	floating-point source register for the MPYF32 (R0H to R7H)
RfH	floating-point source register for the MPYF32 (R0H to R7H)
RaH	floating-point destination register for the MOV32 (R0H to R7H) RaH cannot be the same register as RdH
mem32	pointer to a 32-bit memory location. This will be the source of the MOV32.

Opcode

```
LSW: 1110 0011  0000 fffe
MSW: eedd daaa    mem32
```

Description

Multiply the contents of two floating-point registers and load another.

```
RdH = ReH * RfH
RaH = [mem32]
```

Restrictions

The destination register for the MPYF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

```
NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;
```

Pipeline

MPYF32 takes 2 pipeline-cycles (2p) and MOV32 takes a single cycle. That is:

```
    MPYF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
||  MOV32   RaH, mem32      ; 1 cycle
                            ; <-- MOV32 completes, RaH updated
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- MPYF32 completes, RdH updated
    NOP
```

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

Example

Calculate Y = M1*X1 + B1. This example assumes that M1, X1, B1 and Y1 are all on the same data page.

```
   MOVW    DP, #M1       ; Load the data page
   MOV32   R0H,@M1       ; Load R0H with M1
   MOV32   R1H,@X1       ; Load R1H with X1
   MPYF32  R1H,R1H,R0H   ; Multiply M1*X1
|| MOV32   R0H,@B1       ;     and in parallel load R0H with B1
                         ; <-- MOV32 complete
   NOP                   ; Wait 1 cycle for MPYF32 to complete
                         ; <-- MPYF32 complete
   ADDF32  R1H,R1H,R0H   ; Add M*X1 to B1 and store in R1H
   NOP                   ; Wait 1 cycle for ADDF32 to complete
                         ; <-- ADDF32 complete
   MOV32   @Y1,R1H       ; Store the result
```

Calculate Y = (A * B) * C:

```
   MOVL   XAR4, #A
   MOV32  R0H,  *XAR4    ; Load ROH with A
   MOVL   XAR4, #B
   MOV32  R1H,  *XAR4   ; Load R1H with B
   MOVL   XAR4, #C
   MPYF32 R1H,R1H,R0H   ; Calculate R1H = A * B
|| MOV32  R0H,  *XAR4   ;     and in parallel load R2H with C
                        ; <-- MOV32 complete
   MOVL   XAR4, #Y
                        ; <-- MPYF32 complete
   MPYF32 R2H,R1H,R0H   ; Calculate Y = (A * B) * C
   NOP                  ; Wait 1 cycle for MPYF32 to complete
                        ; MPYF32 complete
   MOV32 *XAR4,R2H
```

See also

MPYF32 RdH, ReH, RfH || MOV32 mem32, RaH
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++

MPYF32 RdH, ReH, RfH
||MOV32 mem32, RaH *32-bit Floating-Point Multiply with Parallel Move*

Operands

RdH	floating-point destination register for the MPYF32 (R0H to R7H)
ReH	floating-point source register for the MPYF32 (R0H to R7H)
RfH	floating-point source register for the MPYF32 (R0H to R7H)
mem32	pointer to a 32-bit memory location. This will be the destination of the MOV32.
RaH	floating-point source register for the MOV32 (R0H to R7H)

Opcode

```
LSW: 1110 0000  0000 fffe
MSW: eedd daaa    mem32
```

Description

Multiply the contents of two floating-point registers and move from memory to register.

```
RdH = ReH * RfH,
[mem32] = RaH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

MPYF32 takes 2 pipeline-cycles (2p) and MOV32 takes a single cycle. That is:

```
    MPYF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
||  MOV32   mem32, RaH      ; 1 cycle
                            ; <-- MOV32 completes, mem32 updated
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- MPYF32 completes, RdH updated
    NOP
```

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

Example

```
    MOVL      XAR1, #0xC003   ; XAR1 = 0xC003
    MOVIZF32  R3H, #2.0       ; R3H = 2.0 (0x40000000)
    MPYF32    R3H, R3H, #5.0  ; R3H = R3H * 5.0
    MOVIZF32  R1H, #5.0       ; R1H = 5.0 (0x40A00000)
                              ; <-- MPYF32 complete, R3H = 10.0 (0x41200000)
    MPYF32    R3H, R1H, R3H   ; R3H = R1H * R3H
 || MOV32     *XAR1, R3H      ; and in parallel store previous R3 value
                              ; MOV32 complete, [0xC003] = 0x4120,
                              ;                 [0xC002] = 0x0000
    NOP                       ; 1 cycle delay for MPYF32 to complete
                              ; <-- MPYF32 , R3H = 50.0 (0x42480000)
```

See also

MPYF32 RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R3H, R2H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R6H, RdH, ReH, RfH || MOV32 RaH, mem32
MACF32 R7H, R3H, mem32, *XAR7++

MPYF32 RaH, RbH, RcH
‖SUBF32 RdH, ReH, RfH *32-bit Floating-Point Multiply with Parallel Subtract*

Operands

RaH	floating-point destination register for MPYF32 (R0H to R7H) RaH cannot be the same register as RdH
RbH	floating-point source register for MPYF32 (R0H to R7H)
RcH	floating-point source register for MPYF32 (R0H to R7H)
RdH	floating-point destination register for SUBF32 (R0H to R7H) RdH cannot be the same register as RaH
ReH	floating-point source register for SUBF32 (R0H to R7H)
RfH	floating-point source register for SUBF32 (R0H to R7H)

Opcode

```
LSW: 1110 0111  0101 00ff
MSW: feee dddc  ccbb baaa
```

Description

Multiply the contents of two floating-point registers with parallel subtraction of two registers.

```
RaH = RbH * RcH,
RdH = ReH - RfH
```

Restrictions

The destination register for the MPYF32 and the SUBF32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:.

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 or SUBF32 generates an underflow condition.
- LVF = 1 if MPYF32 or SUBF32 generates an overflow condition.

Pipeline

MPYF32 and SUBF32 both take 2 pipeline-cycles (2p). That is:

```
    MPYF32  RaH, RbH, RcH   ; 2 pipeline cycles (2p)
||  SUBF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- MPYF32, SUBF32 complete. RaH, RdH updated
    NOP
```

Any instruction in the delay slot must not use RaH or RdH as a destination register or as a source operand.

Example

```
    MOVIZF32  R4H, #5.0       ; R4H = 5.0 (0x40A00000)
    MOVIZF32  R5H, #3.0       ; R5H = 3.0 (0x40400000)
    MPYF32    R6H, R4H, R5H   ; R6H = R4H * R5H
 || SUBF32    R7H, R4H, R5H   ; R7H = R4H - R5H
    NOP                       ; 1 cycle delay for MPYF32 || SUBF32 to complete
                              ; <-- MPYF32 || SUBF32 complete,
                              ; R6H = 15.0 (0x41700000), R7H = 2.0 (0x40000000)
```

See also

SUBF32 RaH, RbH, RcH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH

NEGF32 RaH, RbH{, CNDF} *Conditional Negation*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)
CNDF	condition tested

Opcode

```
LSW: 1110 0110  1010 CNDF
MSW: 0000 0000  00bb baaa
```

Description

```
if (CNDF == true) {RaH = - RbH }
else              {RaH = RbH }
```

CNDF is one of the following conditions:

Encode [1]	CNDF	Description	STF Flags Tested
0000	NEQ	Not equal to zero	ZF == 0
0001	EQ	Equal to zero	ZF == 1
0010	GT	Greater than zero	ZF == 0 AND NF == 0
0011	GEQ	Greater than or equal to zero	NF == 0
0100	LT	Less than zero	NF == 1
0101	LEQ	Less than or equal to zero	ZF == 1 AND NF == 1
1010	TF	Test flag set	TF == 1
1011	NTF	Test flag not set	TF == 0
1100	LU	Latched underflow	LUF == 1
1101	LV	Latched overflow	LVF == 1
1110	UNC	Unconditional	None
1111	UNCF [2]	Unconditional with flag modification	None

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	Yes	Yes	No	No

Pipeline

This is a single-cycle instruction.

Example

```
    MOVIZF32     R0H, #5.0     ; R0H =  5.0 (0x40A00000)
    MOVIZF32     R1H, #4.0     ; R1H =  4.0 (0x40800000)
    MOVIZF32     R2H, #-1.5    ; R2H = -1.5 (0xBFC00000)

    MPYF32       R4H, R1H, R2H ; R4H = -6.0
    MPYF32       R5H, R0H, R1H ; R5H = 20.0
                               ; <-- R4H valid
    CMPF32       R4H, #0.0     ; NF = 1
                               ; <-- R5H valid
    NEGF32       R4H, R4H, LT  ; if NF = 1, R4H = 6.0
    CMPF32       R5H, #0.0     ; NF = 0
    NEGF32       R5H, R5H, GEQ ; if NF = 0, R4H = -20.0
```

See also

ABSF32 RaH, RbH

POP RB *Pop the RB Register from the Stack*

Operands

RB	repeat block register

Opcode

```
LSW: 1111 1111  1111 0001
```

Description

Restore the RB register from stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

Flags

This instruction does not affect any flags floating-point Unit:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a single-cycle instruction.

Example

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
_Interrupt:               ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Save RB register only if a RPTB block is used in the
ISR
       ...
       ...
       RPTB #BlockEnd, AL ; Execute the block AL+1 times
       ...
       ...
BlockEnd                  ; End of block to be repeated
       ...
       ...
       POP  RB            ; Restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```
; Repeat Block within a Low-Priority Interrupt (Interruptible)
_Interrupt:               ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Always save RB register
       ...
       CLRC INTM          ; Enable interrupts only after saving RB
       ...
       ...                ; ISR may or may not include a RPTB block
       ...
       SETC INTM          ; Disable interrupts before restoring RB
       ...
       POP  RB            ; Always restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

See also

PUSH RB
RPTB #RSIZE, RC
RPTB #RSIZE, loc16

PUSH RB *Push the RB Register onto the Stack*

Operands

RB	repeat block register

Opcode

```
LSW: 1111 1111  1111 0000
```

Description

Save the RB register on the stack. If a high-priority interrupt contains a RPTB instruction, then the RB register must be stored on the stack before the RPTB block and restored after the RTPB block. In a low-priority interrupt RB must always be saved and restored. This save and restore must occur when interrupts are disabled.

Flags

This instruction does not affect any flags floating-point Unit:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a single-cycle instruction for the first iteration, and zero cycles thereafter.

Example

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not have to save the RB register.

```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
_Interrupt:               ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Save RB register only if a RPTB block is used in the
ISR
       ...
       RPTB #BlockEnd, AL  ; Execute the block AL+1 times
       ...
       ...
BlockEnd                  ; End of block to be repeated
       ...
       POP  RB            ; Restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```
; Repeat Block within a Low-Priority Interrupt (Interruptible)
_Interrupt:               ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Always save RB register
       ...
       CLRC INTM          ; Enable interrupts only after saving RB
       ...
       ...                ; ISR may or may not include a RPTB block
       ...
       SETC INTM          ; Disable interrupts before restoring RB
       ...
       POP  RB            ; Always restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

See also

POP RB
RPTB #RSIZE, RC
RPTB #RSIZE, loc16

RESTORE *Restore the Floating-Point Registers*

Operands

none	This instruction does not have any operands

Opcode

```
LSW: 1110 0101  0110 0010
```

Description

Restore the floating-point register set (R0H - R7H and STF) from their shadow registers. The SAVE and RESTORE instructions should be used in high-priority interrupts. That is interrupts that cannot themselves be interrupted. In low-priority interrupt routines the floating-point registers should be pushed onto the stack.

Restrictions

The RESTORE instruction cannot be used in any delay slots for pipelined operations. Doing so will yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the RESTORE operation.

```
; The following is INVALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   RESTORE                   ; INVALID, do not use RESTORE in a delay slot

; The following is VALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   NOP                       ; 1 delay cycle, R2H updated after this instruction
   RESTORE                   ; VALID
```

Flags

Restoring the status register will overwrite all flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Pipeline

This is a single-cycle instruction.

Example

The following example shows a complete context save and restore for a high-priority interrupt. Note that the CPU automatically stores the following registers: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1 and PC. If an interrupt is low priority (that is it can be interrupted), then push the floating point registers onto the stack instead of using the SAVE and RESTORE operations.

```
; Interrupt Save
_HighestPriorityISR:    ; Uninterruptable
   ASP                  ; Align stack
   PUSH  RB             ; Save RB register if used in the ISR
   PUSH  AR1H:AR0H         ; Save other registers if used
   PUSH  XAR2
   PUSH  XAR3
   PUSH  XAR4
   PUSH  XAR5
   PUSH  XAR6
   PUSH  XAR7
   PUSH  XT
   SPM   0              ; Set default C28 modes
   CLRC  AMODE
   CLRC  PAGE0,OVM
   SAVE  RNDF32=1       ; Save all FPU registers
   ...                  ; set default FPU modes
   ...
; Interrupt Restore
   ...
   RESTORE              ; Restore all FPU registers
   POP   XT             ; restore other registers
   POP   XAR7
   POP   XAR6
   POP   XAR5
   POP   XAR4
   POP   XAR3
   POP   XAR2
   POP   AR1H:AR0H
   POP   RB             ; restore RB register
   NASP                 ; un-align stack
   IRET                 ; return from interrupt
```

See also

SAVE FLAG, VALUE

RPTB label, loc16 *Repeat A Block of Code*

Operands

label	This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block.
loc16	16-bit location for the repeat count value.

Opcode

```
LSW: 1011 0101  0bbb bbbb
MSW: 0000 0000    loc16
```

Description

Initialize repeat block loop, repeat count from [loc16]

Restrictions

- The maximum block size is ≤127 16-bit words.
- An even aligned block must be ≥ 9 16-bit words.
- An odd aligned block must be ≥ 8 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

Flags

This instruction does not affect any flags in the floating-point unit:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

```
; Repeat Block of 8 Words (Interruptible)
;
; find the largest element and put its address in XAR6
  .align 2

   NOP
   RPTB    VECTOR_MAX_END, AR7      ; Execute the block AR7+1 times
   MOVL    ACC,XAR0
   MOV32   R1H,*XAR0++              ; min size = 8, 9 words
   MAXF32  R0H,R1H                  ; max size = 127 words
   MOVST0  NF,ZF
   MOVL    XAR6,ACC,LT
VECTOR_MAX_END:                     ; label indicates the end
                                    ; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not

have to save the RB register.

```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt:              ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Save RB register only if a RPTB block is used in the
ISR
       ...
       ...
       RPTB #BlockEnd, AL ; Execute the block AL+1 times
       ...
       ...
       ...
BlockEnd                  ; End of block to be repeated
       ...
       ...
       POP  RB            ; Restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```
; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt:              ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Always save RB register
       ...
       CLRC INTM          ; Enable interrupts only after saving RB
       ...
       ...
       ...                ; ISR may or may not include a RPTB block
       ...
       ...
       SETC INTM          ; Disable interrupts before restoring RB
       ...
       POP  RB            ; Always restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

See also

POP RB
PUSH RB
RPTB label, RC

RPTB label, #RC *Repeat a Block of Code*

Operands

label	This label is used by the assembler to determine the end of the repeat block and to calculate RSIZE. This label should be placed immediately after the last instruction included in the repeat block.
#RC	16-bit immediate value for the repeat count.

Opcode

```
LSW: 1011 0101  1bbb bbbb
MSW: cccc cccc  cccc cccc
```

Description

Repeat a block of code. The repeat count is specified as a immediate value.

Restrictions

- The maximum block size is ≤127 16-bit words.
- An even aligned block must be ≥ 9 16-bit words.
- An odd aligned block must be ≥ 8 16-bit words.
- Interrupts must be disabled when saving or restoring the RB register.
- Repeat blocks cannot be nested.
- Any discontinuity type operation is not allowed inside a repeat block. This includes all call, branch or TRAP instructions. Interrupts are allowed.
- Conditional execution operations are allowed.

Flags

This instruction does not affect any flags int the floating-point unit:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example

The minimum size for the repeat block is 8 words if the block is even aligned and 9 words if the block is odd aligned. If you have a block of 8 words, as in the following example, you can make sure the block is odd aligned by proceeding it by a .align 2 directive and a NOP instruction. The .align 2 directive will make sure the NOP is even aligned. Since a NOP is a 16-bit instruction the RPTB will be odd aligned. For blocks of 9 or more words, this is not required.

```
; Repeat Block (Interruptible)
;
; find the largest element and put its address in XAR6
  .align 2

   NOP
   RPTB    VECTOR_MAX_END, #(4-1)   ; Execute the block 4 times
   MOVL    ACC,XAR0
   MOV32   R1H,*XAR0++              ; 8 or 9 words ≤ block size ≤ 127 words
   MAXF32  R0H,R1H
   MOVST0  NF,ZF
   MOVL    XAR6,ACC,LT
VECTOR_MAX_END:                     ; RE indicates the end address
                                    ; RA is cleared
```

When an interrupt is taken the repeat active (RA) bit in the RB register is automatically copied to the repeat active shadow (RAS) bit. When the interrupt exits, the RAS bit is automatically copied back to the RA bit. This allows the hardware to keep track if a repeat loop was active whenever an interrupt is taken and restore that state automatically.

A high priority interrupt is defined as an interrupt that cannot itself be interrupted. In a high priority interrupt, the RB register must be saved if a RPTB block is used within the interrupt. If the interrupt service routine does not include a RPTB block, then you do not

have to save the RB register.

```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
; Interrupt:              ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Save RB register only if a RPTB block is used in the
ISR
       ...
       ...
       RPTB #BlockEnd, #5 ; Execute the block 5+1 times
       ...
       ...
       ...
BlockEnd                  ; End of block to be repeated
       ...
       ...
       POP  RB            ; Restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

A low-priority interrupt is defined as an interrupt that allows itself to be interrupted. The RB register must always be saved and restored in a low-priority interrupt. The RB register must stored before interrupts are enabled. Likewise before restoring the RB register interrupts must first be disabled.

```
; Repeat Block within a Low-Priority Interrupt (Interruptible)
;
; Interrupt:              ; RAS = RA, RA = 0
       ...
       PUSH RB            ; Always save RB register
       ...
       CLRC INTM          ; Enable interrupts only after saving RB
       ...
       ...
       ...                ; ISR may or may not include a RPTB block
       ...
       ...
       SETC INTM          ; Disable interrupts before restoring RB
       ...
       POP  RB            ; Always restore RB register
       ...
       IRET               ; RA = RAS, RAS = 0
```

See also

POP RB
PUSH RB
RPTB #RSIZE, loc16

SAVE FLAG, VALUE *Save Register Set to Shadow Registers and Execute SETFLG*

Operands

FLAG	11 bit mask indicating which floating-point status flags to change.
VALUE	11 bit mask indicating the flag value; 0 or 1.

Opcode

```
LSW: 1110 0110  01FF FFFF
MSW: FFFF FVVV  VVVV VVVV
```

Description

This operation copies the current working floating-point register set (R0H to R7H and STF) to the shadow register set and combines the SETFLG FLAG, VALUE operation in a single cycle. The status register is copied to the shadow register before the flag values are changed. The STF[SHDWM] flag is set to 1 when the SAVE command has been executed. The SAVE and RESTORE instructions should be used in high-priority interrupts. That is interrupts that cannot themselves be interrupted. In low-priority interrupt routines the floating-point registers should be pushed onto the stack.

Restrictions

Do not use the SAVE instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the SAVE operation.

```
; The following is INVALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   SAVE    RNDF32=1          ; INVALID, do not use SAVE in a delay slot


; The following is VALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   NOP                       ; 1 delay cycle, R2H updated after this instruction
   SAVE  RNDF32=1            ; VALID
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Any flag can be modified by this instruction.

Pipeline

This is a single-cycle instruction.

Example

To make it easier and more legible, the assembler will accept a FLAG=VALUE syntax for the STFLG operation as shown below:

```
   SAVE    RNDF32=0, TF=1, ZF=0  ; FLAG = 01001000100, VALUE = X0XX0XXX1XX
   MOVST0  TF, ZF, LUF           ; Copy the indicated flags to ST0
                                 ; Note: X means this flag will not be modified.
                                 ; The assembler will set these X values to 0.
```

The following example shows a complete context save and restore for a high priority interrupt. Note that the CPU automatically stores the following registers: ACC, P, XT, ST0, ST1, IER, DP, AR0, AR1 and PC.

```
_HighestPriorityISR:
   ASP                  ; Align stack
   PUSH  RB             ; Save RB register if used in the ISR
   PUSH  AR1H:AR0H        ; Save other registers if used
   PUSH  XAR2
   PUSH  XAR3
   PUSH  XAR4
   PUSH  XAR5
   PUSH  XAR6
   PUSH  XAR7
   PUSH  XT
   SPM   0              ; Set default C28 modes
   CLRC  AMODE
   CLRC  PAGE0,OVM
   SAVE  RNDF32=0       ; Save all FPU registers
   ...                  ; set default FPU modes
   ...
   ...
   ...
   RESTORE              ; Restore all FPU registers
   POP   XT             ; restore other registers
   POP   XAR7
   POP   XAR6
   POP   XAR5
   POP   XAR4
   POP   XAR3
   POP   XAR2
   POP   AR1H:AR0H
   POP   RB             ; restore RB register
   NASP                 ; un-align stack
   IRET                 ; return from interrupt
```

See also

RESTORE
SETFLG FLAG, VALUE

SETFLG FLAG, VALUE *Set or clear selected floating-point status flags*

Operands

FLAG	11 bit mask indicating which floating-point status flags to change.
VALUE	11 bit mask indicating the flag value; 0 or 1.

Opcode

```
LSW: 1110 0110  00FF FFFF
MSW: FFFF FVVV  VVVV VVVV
```

Description

The SETFLG instruction is used to set or clear selected floating-point status flags in the STF register. The FLAG field is an 11-bit value that indicates which flags will be changed. That is, if a FLAG bit is set to 1 it indicates that flag will be changed; all other flags will not be modified. The bit mapping of the FLAG field is shown below:

10	9	8	7	6	5	4	3	2	1	0
reserved	RNDF32	reserved	reserved	TF	ZI	NI	ZF	NF	LUF	LVF

The VALUE field indicates the value the flag should be set to; 0 or 1.

Restrictions

Do not use the SETFLG instruction in the delay slots for pipelined operations. Doing so can yield invalid results. To avoid this, the proper number of NOPs or non-pipelined instructions must be inserted before the SETFLG operation.

```
; The following is INVALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   SETFLG  RNDF32=1          ; INVALID, do not use SETFLG in a delay slot


; The following is VALID
   MPYF32  R2H, R1H, R0H     ; 2 pipeline-cycle instruction (2p)
   NOP                       ; 1 delay cycle, R2H updated after this instruction
   SETFLG  RNDF32=1          ; VALID
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Any flag can be modified by this instruction.

Pipeline

This is a single-cycle instruction.

Example

To make it easier and legible, the assembler will accept a FLAG=VALUE syntax for the STFLG operation as shown below:

```
   SETFLG    RNDF32=0, TF=1, ZF=0      ; FLAG = 01001001000, VALUE = X0XX1XX0XXX
   MOVST0    TF, ZF, LUF               ; Copy the indicated flags to ST0
                                       ; X means this flag is not modified.
                                       ; The assembler will set X values to 0
```

See also

SAVE FLAG, VALUE

SUBF32 RaH, RbH, RcH *32-bit Floating-Point Subtraction*

Operands

RaH	floating-point destination register (R0H to R1)
RbH	floating-point source register (R0H to R1)
RcH	floating-point source register (R0H to R1)

Opcode

```
LSW: 1110 0111  0010 0000
MSW: 0000 000c  ccbb baaa
```

Description

Subtract the contents of two floating-point registers

```
RaH = RbH - RcH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
    SUBF32  RaH, RbH, RcH    ; 2 pipeline cycles (2p)
    NOP                      ; 1 cycle delay or non-conflicting instruction
                             ; <-- SUBF32 completes, RaH updated
    NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

Calculate Y - A + B - C:

```
   MOVL   XAR4, #A
   MOV32  R0H,  *XAR4   ; Load R0H with A
   MOVL   XAR4, #B
   MOV32  R1H,  *XAR4   ; Load R1H with B
   MOVL   XAR4, #C
   ADDF32 R0H,R1H,R0H   ; Add A + B and in parallel
|| MOV32  R2H,*XAR4     ; Load R2H with C
                        ; <-- MOV32 complete
   MOVL   XAR4,#_xt
                        ; <-- ADDF32 complete
   SUBF32 R0H,R0H,R2H   ; Subtract C from (A + B)
   NOP
                        ; <-- SUBF32 completes
   MOV32  *XAR4,R0H     ; Store the result
```

See also

SUBF32 RaH, #16FHi, RbH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH

SUBF32 RaH, #16FHi, RbH *32-bit Floating Point Subtraction*

Operands

RaH	floating-point destination register (R0H to R1)
#16FHi	A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0.
RbH	floating-point source register (R0H to R1)

Opcode

```
LSW: 1110 1000  11II IIII
MSW: IIII IIII  IIbb baaa
```

Description

Subtract RbH from the floating-point value represented by the immediate operand. Store the result of the addition in RaH.

#16FHi is a 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. #16FHi is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 (0x40000000), 4.0 (0x40800000), 0.5 (0x3F000000), and -1.5 (0xBFC00000). The assembler will accept either a hex or float as the immediate value. That is, the value -1.5 can be represented as #-1.5 or #0xBFC0.

```
RaH = #16FHi:0 - RbH
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
SUBF32  RaH, #16FHi, RbH ; 2 pipeline cycles (2p)
NOP                      ; 1 cycle delay or non-conflicting instruction
                         ; <-- SUBF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

Calculate Y = 2.0 - (A + B):

```
MOVL   XAR4, #A
MOV32  R0H,  *XAR4   ; Load R0H with A
MOVL   XAR4, #B
MOV32  R1H,  *XAR4   ; Load R1H with B
ADDF32 R0H,R1H,R0H   ; Add A + B and in parallel
NOP
                     ; <-- ADDF32 complete
SUBF32 R0H,#2.0,R2H  ; Subtract (A + B) from 2.0
NOP
                     ; <-- SUBF32 completes
MOV32  *XAR4,R0H     ; Store the result
```

See also

SUBF32 RaH, RbH, RcH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH

SUBF32 RdH, ReH, RfH
||MOV32 RaH, mem32 *32-bit Floating-Point Subtraction with Parallel Move*

Operands

RdH	floating-point destination register (R0H to R7H) for the SUBF32 operation RdH cannot be the same register as RaH
ReH	floating-point source register (R0H to R7H) for the SUBF32 operation
RfH	floating-point source register (R0H to R7H) for the SUBF32 operation
RaH	floating-point destination register (R0H to R7H) for the MOV32 operation RaH cannot be the same register as RdH
mem32	pointer to 32-bit source memory location for the MOV32 operation

Opcode

```
LSW: 1110 0011  0010 fffe
MSW: eedd daaa    mem32
```

Description

Subtract the contents of two floating-point registers and move from memory to a floating-point register.

```
RdH = ReH - RfH,
RaH = [mem32]
```

Restrictions

The destination register for the SUBF32 and the MOV32 must be unique. That is, RaH cannot be the same register as RdH.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	Yes	Yes	Yes	Yes	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if SUBF32 generates an underflow condition.
- LVF = 1 if SUBF32 generates an overflow condition.

The MOV32 Instruction will set the NF, ZF, NI and ZI flags as follows:

```
NF = RaH(31);
ZF = 0;
if(RaH(30:23) == 0) { ZF = 1; NF = 0; }
NI = RaH(31);
ZI = 0;
if(RaH(31:0) == 0) ZI = 1;
```

Pipeline

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

```
    SUBF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
||  MOV32   RaH, mem32      ; 1 cycle
                            ; <-- MOV32 completes, RaH updated
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- SUBF32 completes, RdH updated
    NOP
```

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

Example

```
    MOVL    XAR1, #0xC000  ; XAR1 = 0xC000
    SUBF32  R0H, R1H, R2H  ; (A) R0H = R1H - R2H
||  MOV32   R3H, *XAR1     ;
                           ; <-- R3H valid
    MOV32   R4H, *+XAR1[2] ;
                           ; <-- (A) completes, R0H valid, R4H valid
    ADDF32  R5H, R4H, R3H  ; (B) R5H = R4H + R3H
||  MOV32   *+XAR1[4], R0H ;
                           ; <-- R0H stored
    MOVL    XAR2, #0xE000  ;
                           ; <-- (B) completes, R5H valid
    MOV32   *XAR2, R5H     ;
                           ; <-- R5H stored
```

See also

SUBF32 RaH, RbH, RcH
SUBF32 RaH, #16FHi, RbH
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH

SUBF32 RdH, ReH, RfH
||MOV32 mem32, RaH *32-bit Floating-Point Subtraction with Parallel Move*

Operands

RdH	floating-point destination register (R0H to R7H) for the SUBF32 operation
ReH	floating-point source register (R0H to R7H) for the SUBF32 operation
RfH	floating-point source register (R0H to R7H) for the SUBF32 operation
mem32	pointer to 32-bit destination memory location for the MOV32 operation
RaH	floating-point source register (R0H to R7H) for the MOV32 operation

Opcode

```
LSW: 1110 0000  0010 fffe
MSW: eedd daaa    mem32
```

Description

Subtract the contents of two floating-point registers and move from a floating-point register to memory.

```
RdH = ReH - RfH,
[mem32] = RaH
```

Flags

This instruction modifies the following flags in the STF register:SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	Yes	Yes

The STF register flags are modified as follows:

- LUF = 1 if SUBF32 generates an underflow condition.
- LVF = 1 if SUBF32 generates an overflow condition.

Pipeline

SUBF32 is a 2 pipeline-cycle instruction (2p) and MOV32 takes a single cycle. That is:

```
    SUBF32  RdH, ReH, RfH   ; 2 pipeline cycles (2p)
||  MOV32   mem32, RaH      ; 1 cycle
                            ; <-- MOV32 completes, mem32 updated
    NOP                     ; 1 cycle delay or non-conflicting instruction
                            ; <-- ADDF32 completes, RdH updated
    NOP
```

Any instruction in the delay slot must not use RdH as a destination register or as a source operand.

Example

```
   ADDF32      R3H, R6H, R4H          ; (A) R3H = R6H + R4H and R7H = I3
|| MOV32       R7H, *-SP[2]           ;
                                       ; <-- R7H valid
   SUBF32      R6H, R6H, R4H          ; (B) R6H = R6H - R4H
                                       ; <-- ADDF32 (A) completes, R3H valid
   SUBF32      R3H, R1H, R7H          ; (C) R3H = R1H - R7H and store R3H (A)
|| MOV32       *+XAR5[2], R3H         ;
                                       ; <-- SUBF32 (B) completes, R6H valid
                                       ; <-- MOV32 completes, (A) stored
   ADDF32      R4H, R7H, R1H          ; R4H = D = R7H + R1H and store R6H (B)
|| MOV32       *+XAR5[6], R6H         ;
                                       ; <-- SUBF32 (C) completes, R3H valid
                                       ; <-- MOV32 completes, (B) stored
   MOV32       *+XAR5[0], R3H         ; store R3H (C)
                                       ; <-- MOV32 completes, (C) stored
                                       ; <-- ADDF32 (D) completes, R4H valid
   MOV32       *+XAR5[4], R4H         ; store R4H (D)
                                       ; <-- MOV32 completes, (D) stored
```

See also

SUBF32 RaH, RbH, RcH
SUBF32 RaH, #16FHi, RbH
SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH

SWAPF RaH, RbH{, CNDF} *Conditional Swap*

Operands

RaH	floating-point register (R0H to R7H)
RbH	floating-point register (R0H to R7H)
CNDF	condition tested

Opcode

```
LSW: 1110 0110  1110 CNDF
MSW: 0000 0000  00bb baaa
```

Description

Conditional swap of RaH and RbH.

```
if (CNDF == true) swap RaH and RbH
```

CNDF is one of the following conditions:

Encode (1)	CNDF	Description	STF Flags Tested
0000	NEQ	Not equal to zero	ZF == 0
0001	EQ	Equal to zero	ZF == 1
0010	GT	Greater than zero	ZF == 0 AND NF == 0
0011	GEQ	Greater than or equal to zero	NF == 0
0100	LT	Less than zero	NF == 1
0101	LEQ	Less than or equal to zero	ZF == 1 AND NF == 1
1010	TF	Test flag set	TF == 1
1011	NTF	Test flag not set	TF == 0
1100	LU	Latched underflow	LUF == 1
1101	LV	Latched overflow	LVF == 1
1110	UNC	Unconditional	None
1111	UNCF (2)	Unconditional with flag modification	None

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

No flags affected

Pipeline

This is a single-cycle instruction.

Example

```
;find the largest element and put it in R1H

    MOVL      XAR1, #0xB000    ;
    MOV32     R1H, *XAR1       ; Initialize R1H
    .align 2

    NOP
    RPTB      LOOP_END, #(10-1); Execute the block 10 times
    MOV32     R2H, *XAR1++     ; Update R2H with next element
    CMPF32    R2H, R1H         ; Compare R2H with R1H
    SWAPF     R1H, R2H, GT     ; Swap R1H and R2H if R2 > R1
    NOP                        ; For minimum repeat block size
    NOP                        ; For minimum repeat block size
LOOP_END:
```

See also

TESTTF CNDF *Test STF Register Flag Condition*

Operands

CNDF	condition to test

Opcode

```
LSW: 1110 0101  1000 CNDF
```

Description

Test the floating-point condition and if true, set the TF flag. If the condition is false, clear the TF flag. This is useful for temporarily storing a condition for later use.

```
if (CNDF == true) TF = 1; else TF = 0;
```

CNDF is one of the following conditions:

Encode [1]	CNDF	Description	STF Flags Tested
0000	NEQ	Not equal to zero	ZF == 0
0001	EQ	Equal to zero	ZF == 1
0010	GT	Greater than zero	ZF == 0 AND NF == 0
0011	GEQ	Greater than or equal to zero	NF == 0
0100	LT	Less than zero	NF == 1
0101	LEQ	Less than or equal to zero	ZF == 1 AND NF == 1
1010	TF	Test flag set	TF == 1
1011	NTF	Test flag not set	TF == 0
1100	LU	Latched underflow	LUF == 1
1101	LV	Latched overflow	LVF == 1
1110	UNC	Unconditional	None
1111	UNCF [2]	Unconditional with flag modification	None

(1) Values not shown are reserved.
(2) This is the default operation if no CNDF field is specified. This condition will allow the ZF, NF, ZI, and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	Yes	No	No	No	No	No	No

```
TF = 0; if (CNDF == true) TF = 1;
```

Note: If (CNDF == UNC or UNCF), the TF flag will be set to 1.

Pipeline

This is a single-cycle instruction.

Example

```
   CMPF32    R0H, #0.0   ; Compare R0H against 0
   TESTTF    LT          ; Set TF if R0H less than 0 (NF == 0)
   ABS       R0H, R0H    ; Get the absolute value of R0H

   ; Perform calculations based on ABS R0H

   MOVST0    TF          ; Copy TF to TC in ST0
   SBF       End, NTC    ; Branch to end if TF was not set
   NEGF32    R0H, R0H
End
```

See also

UI16TOF32 RaH, mem16 *Convert unsigned 16-bit integer to 32-bit floating-point value*

Operands

RaH	floating-point destination register (R0H to R7H)
mem16	pointer to 16-bit source memory location

Opcode

```
LSW: 1110 0010  1100 0100
MSW: 0000 0aaa    mem16
```

Description

When converting F32 to I16/UI16 data format, the F32TOI16/UI16 operation truncates to zero while the F32TOI16R/UI16R operation will round to nearest (even) value.

```
RaH = UI16ToF32[mem16]
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
    UI16TOF32  RaH, mem16      ; 2 pipeline cycles (2p)
    NOP                        ; 1 cycle delay or non-conflicting instruction
                               ; <-- UI16TOF32 completes, RaH updated
    NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

```
; float32 y,m,b;
; AdcRegs.RESULT0 is an unsigned int
; Calculate: y = (float)AdcRegs.ADCRESULT0 * m + b;
;
   MOVW       DP   @0x01C4
   UI16TOF32  R0H, @8          ; R0H = (float)AdcRegs.RESULT0
   MOV32      R1H, *-SP[6]     ; R1H = M
                               ; <-- Conversion complete, R0H valid
   MPYF32     R0H, R1H, R0H    ; R0H = (float)X * M
   MOV32      R1H, *-SP[8]     ; R1H = B
                               ; <-- MPYF32 complete, R0H valid
   ADDF32     R0H, R0H, R1H    ; R0H = Y = (float)X * M + B
   NOP
                               ; <-- ADDF32 complete, R0H valid
   MOV32      *-[SP], R0H      ; Store Y
```

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, RbH

UI16TOF32 RaH, RbH *Convert unsigned 16-bit integer to 32-bit floating-point value*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1111
MSW: 0000 0000  00bb baaa
```

Description

When converting F32 to I16/UI16 data format, the F32TOI16/UI16 operation truncates to zero while the F32TOI16R/UI16R operation will round to nearest (even) value.

```
RaH = UI16ToF32[RbH]
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
UI16TOF32  RaH, RbH        ; 2 pipeline cycles (2p)
NOP                        ; 1 cycle delay or non-conflicting instruction
                           ; <-- UI16TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

```
MOVXI      R5H, #0x800F  ; R5H[15:0] = 32783 (0x800F)
UI16TOF32  R6H, R5H      ; R6H = UI16TOF32 (R5H[15:0])
NOP                      ; 1 cycle delay for UI16TOF32 to complete
                         ; R6H = 32783.0 (0x47000F00)
```

See also

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
UI16TOF32 RaH, mem16

UI32TOF32 RaH, mem32 *Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
mem32	pointer to 32-bit source memory location

Opcode

```
LSW: 1110 0010  1000 0100
MSW: 0000 0aaa    mem32
```

Description

```
RaH = UI32ToF32[mem32]
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
    UI32TOF32  RaH, mem32      ; 2 pipeline cycles (2p)
    NOP                        ; 1 cycle delay  non-conflicting instruction
                               ; <-- UI32TOF32 completes, RaH updated
    NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

```
; unsigned long X
; float Y, M, B
; ...
; Calculate Y = (float)X * M + B
;
   UI32TOF32  R0H, *-SP[2]     ; R0H = (float)X
   MOV32      R1H, *-SP[6]     ; R1H = M
                               ; <-- Conversion complete, R0H valid
   MPYF32     R0H, R1H, R0H    ; R0H = (float)X * M
   MOV32      R1H, *-SP[8]     ; R1H = B
                               ; <-- MPYF32 complete, R0H valid
   ADDF32     R0H, R0H, R1H    ; R0H = Y = (float)X * M + B
   NOP
                               ; <-- ADDF32 complete, R0H valid
   MOV32      *-[SP], R0H      ; Store Y
```

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
I32TOF32 RaH, RbH
UI32TOF32 RaH, RbH

UI32TOF32 RaH, RbH *Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value*

Operands

RaH	floating-point destination register (R0H to R7H)
RbH	floating-point source register (R0H to R7H)

Opcode

```
LSW: 1110 0110  1000 1011
MSW: 0000 0000  00bb baaa
```

Description

```
RaH = UI32ToF32 RbH
```

Flags

This instruction does not affect any flags:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

Pipeline

This is a 2 pipeline cycle (2p) instruction. That is:

```
UI32TOF32  RaH, RbH        ; 2 pipeline cycles (2p)
NOP                        ; 1 cycle delay or non-conflicting instruction
                           ; <-- UI32TOF32 completes, RaH updated
NOP
```

Any instruction in the delay slot must not use RaH as a destination register or as a source operand.

Example

```
MOVIZ      R3H, #0x8000  ; R3H[31:16] = 0x8000
MOVXI      R3H, #0x1111  ; R3H[15:0]  = 0x1111
                         ; R3H = 2147488017
UI32TOF32  R4H, R3H      ; R4H = UI32TOF32 (R3H)
NOP                      ; 1 cycle delay for UI32TOF32 to complete
                         ; R4H = 2147488017.0 (0x4F000011)
```

See also

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
I32TOF32 RaH, RbH
UI32TOF32 RaH, mem32

ZERO RaH — *Zero the Floating-Point Register RaH*

Operands

RaH	floating-point register (R0H to R7H)

Opcode

```
LSW: 1110 0101  1001 0aaa
```

Description

Zero the indicated floating-point register:

```
RaH = 0
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

No flags affected.

Pipeline

This is a single-cycle instruction.

Example

```
;for(i = 0; i < n; i++)
;{
;   real += (x[2*i] * y[2*i]) - (x[2*i+1] * y[2*i+1]);
;   imag += (x[2*i] * y[2*i+1]) + (x[2*i+1] * y[2*i]);
;}
;Assume AR7 = n-1

    ZERO    R4H                 ; R4H = real = 0
    ZERO    R5H                 ; R5H = imag = 0
LOOP
    MOV     AL, AR7
    MOV     ACC, AL << 2
    MOV     AR0, ACC
    MOV32   R0H, *+XAR4[AR0]    ; R0H =  x[2*i]
    MOV32   R1H, *+XAR5[AR0]    ; R1H =  y[2*i]
    ADD     AR0, #2
    MPYF32  R6H, R0H, R1H;      ; R6H =  x[2*i] * y[2*i]
 || MOV32   R2H, *+XAR4[AR0]    ; R2H =  x[2*i+1]
    MPYF32  R1H, R1H, R2H       ; R1H =  y[2*i] * x[2*i+2]
 || MOV32   R3H, *+XAR5[AR0]    ; R3H =  y[2*i+1]
    MPYF32  R2H, R2H, R3H       ; R2H =  x[2*i+1] * y[2*i+1]
 || ADDF32  R4H, R4H, R6H       ; R4H += x[2*i] * y[2*i]
    MPYF32  R0H, R0H, R3H       ; R0H =  x[2*i] * y[2*i+1]
 || ADDF32  R5H, R5H, R1H       ; R5H += y[2*i] * x[2*i+2]
    SUBF32  R4H, R4H, R2H       ; R4H -= x[2*i+1] * y[2*i+1]
    ADDF32  R5H, R5H,R0H        ; R5H += x[2*i] * y[2*i+1]
    BANZ    LOOP , AR7--
```

See also

ZEROA

ZEROA *Zero All Floating-Point Registers*

Operands

none

Opcode

```
LSW: 1110 0101  0110 0011
```

Description

Zero all floating-point registers:

```
R0H = 0
R1H = 0
R2H = 0
R3H = 0
R4H = 0
R5H = 0
R6H = 0
R7H = 0
```

Flags

This instruction modifies the following flags in the STF register:

Flag	TF	ZI	NI	ZF	NF	LUF	LVF
Modified	No	No	No	No	No	No	No

No flags affected.

Pipeline

This is a single-cycle instruction.

Example

```
;for(i = 0; i < n; i++)
;{
;   real += (x[2*i] * y[2*i]) - (x[2*i+1] * y[2*i+1]);
;   imag += (x[2*i] * y[2*i+1]) + (x[2*i+1] * y[2*i]);
;}
;Assume AR7 = n-1

    ZER0A                       ; Clear all RaH registers
LOOP
    MOV     AL, AR7
    MOV     ACC, AL << 2
    MOV     AR0, ACC
    MOV32   R0H, *+XAR4[AR0]    ; R0H =  x[2*i]
    MOV32   R1H, *+XAR5[AR0]    ; R1H =  y[2*i]
    ADD     AR0,#2
    MPYF32  R6H, R0H, R1H;      ; R6H =  x[2*i] * y[2*i]
 || MOV32   R2H, *+XAR4[AR0]    ; R2H =  x[2*i+1]
    MPYF32  R1H, R1H, R2H       ; R1H =  y[2*i] * x[2*i+2]
 || MOV32   R3H, *+XAR5[AR0]    ; R3H =  y[2*i+1]
    MPYF32  R2H, R2H, R3H       ; R2H =  x[2*i+1] * y[2*i+1]
 || ADDF32  R4H, R4H, R6H       ; R4H += x[2*i] * y[2*i]
    MPYF32  R0H, R0H, R3H       ; R0H =  x[2*i] * y[2*i+1]
 || ADDF32  R5H, R5H, R1H       ; R5H += y[2*i] * x[2*i+2]
    SUBF32  R4H, R4H, R2H       ; R4H -= x[2*i+1] * y[2*i+1]
    ADDF32  R5H, R5H,R0H        ; R5H += x[2*i] * y[2*i+1]
    BANZ    LOOP , AR7--
```

See also

ZERO RaH

Revision History

A.1 Changes

This revision history lists the technical changes made in the most recent revision.

Table A-1. Technical Changes Made in This Revision

Location	Additions, Deletions, Modifications
Figure 1-1	Modified the functional block diagram
Section 1.2.1	Added this section.
Section 1.3.1	Deleted part of the last bullet in Emulation Logic section
Section 1.4.1	Modified bullets in Address and Data Buses section
Example 2-2	Modified code in The Repeat Block Instruction example
Example 3-8	Modified text following Destination Register Conflict Resolved example
ADDF32 RaH, #16FHi, RbH ADDF32 RaH, RbH, #16FHi	Modified operand for instruction ADDF32 RaH, #16FHi, RbH. Updated the description.
ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32	Modified the instruction ADDF32 RdH, ReH, RfH || MOV32 RaH, mem32
CMPF32 RaH, #16FHi	Modified the CMPF32 RaH, #16FHi instruction
CMPF32 RaH, #0.0	Modified the CMPF32 RaH, #0.0 instruction
F32TOI32 RaH, RbH	Modified the F32TO132 RaH, RbH instruction
F32TOUI32 RaH, RbH	Modified the F32TOUI32 RaH, RbH instruction
I16TOF32 RaH, RbH	Modified the I16TOF32 RaH, RbH instruction
MACF32 R3H, R2H, RdH, ReH	Modified the MACF32 R3H, R2H, RdH, ReH, RfH instruction
MAXF32 RaH, #16FHi	Modifed the syntax of the immediate operand. Modified the desciption.
MINF32 RaH, #16FHi	Modified the syntax of the immediate operand. Modified the description.
MINF32 RaH, RbH	Modified the MINF32 RaH, RbH instruction
MOV16 mem16, RaH	Modified the MOV16 mem16, RaH instruction
MOV32 loc32, *(0:16bitAddr)	Modified the MOV32 loc32, #(0:16bitAddr) instruction
MOV32 mem32, RaH	Modified the MOV32 mem32, RaH instruction
MOV32 mem32, STF	Modified the MOV32 mem32, STF instruction
MOV32 RaH, XT	Modified the MOV32 RaH, XT instruction
MOVF32 RaH, #32F	Added the MOV32 RaH, #32 instruction
MOVI32 RaH, #32FHex	Added the MOVI32 RaH, #32FHex instruction
MOVIZ RaH, #16FHiHex	Modified the syntax for the immediate operand. Modified the description. Modified the example.
MOVIZF32 RaH, #16FHi	Modified the syntax for the immediate operand. Modified the description.
MOVXI RaH, #16FLo	Modified the MOVXI RaH, #16FLo instruction. Modified the syntax of the immediate operand. Modified the description.
MPYF32 RaH, #16FHi, RbH MPYF32 RaH, RbH, #16FHi	Modified the syntax of the immediate operand. Modified the instruction description.
MPYF32 RdH, ReH, RfH	Modified the MPYF32 RdH, ReH, RfH ||MOV32 RaH, mem32 instruction
SAVE FLAG, VALUE	Modified the SAVE FLAG, VALUE instruction
SUBF32 RaH, #16FHi, RbH	Modified the syntax for the immediate operand. Modified the description.

Table A-1. Technical Changes Made in This Revision (continued)

Location	Additions, Deletions, Modifications
SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH	Modified the SUBF32, RdH, ReH, RfH ||MOV32 mem32, RaH instruction
UI16TOF32 RaH, RbH	Modified the UI16TOF32 RaH, RbH instruction
UI32TOF32 RaH, RbH	Modified the UI32TOF32 RaH, RbH instruction
Globally	The syntax sections of the #16F, #16I and #immF32 immediate addressing modes modes were changed to #16FHi, #16FHiHex, and #16FLoHex to be more descriptive and consistent. The descriptions for instructions using these modes were updated for clarity.
Example 2-2	Changed first instruction in example
Table 4-1	Updated the operand nomenclature table
Section 2.1.2	Modified the register figure introduction and register figure
EINVF32 RaH, RbH	Modified the example
EISQRTF32 RaH, RbH	Modified the example
Chapter 4	Added instructions to the See Also area of various instructions

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
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2008, Texas Instruments Incorporated