## TMS320C28x Extended Instruction Sets

## Technical Reference Manual

4 TEXAS<br>INSTRUMENTS

## Contents

Preface ..... 5
1 Floating Point Unit (FPU) ..... 9
1.1 Overview. ..... 10
1.1.1 Compatibility with the C28x Fixed-Point CPU ..... 10
1.2 Components of the C28x plus Floating-Point CPU ..... 11
1.2.1 Emulation Logic ..... 12
1.2.2 Memory Map ..... 12
1.2.3 On-Chip Program and Data ..... 12
1.2.4 CPU Interrupt Vectors ..... 12
1.2.5 Memory Interface ..... 12
1.3 CPU Register Set ..... 13
1.3.1 CPU Registers ..... 13
1.4 Pipeline ..... 19
1.4.1 Pipeline Overview ..... 19
1.4.2 General Guidelines for Floating-Point Pipeline Alignment ..... 20
1.4.3 Moves from FPU Registers to C28x Registers ..... 20
1.4.4 Moves from C28x Registers to FPU Registers ..... 21
1.4.5 Parallel Instructions ..... 21
1.4.6 Invalid Delay Instructions ..... 22
1.4.7 Optimizing the Pipeline ..... 25
1.5 Floating Point Unit Instruction Set ..... 26
1.5.1 Instruction Descriptions ..... 26
1.5.2 Instructions ..... 29
2 C28 Viterbi, Complex Math and CRC Unit-II (VCU-II) ..... 140
2.1 Overview ..... 141
2.2 Components of the C28x Plus VCU ..... 142
2.2.1 Emulation Logic ..... 144
2.2.2 Memory Map ..... 144
2.2.3 CPU Interrupt Vectors ..... 144
2.2.4 Memory Interface ..... 144
2.2.5 Address and Data Buses ..... 144
2.2.6 Alignment of 32-Bit Accesses to Even Addresses ..... 145
2.3 Register Set ..... 146
2.3.1 VCU Register Set ..... 147
2.3.2 VCU Status Register (VSTATUS) ..... 149
2.3.3 Repeat Block Register (RB) ..... 152
2.4 Pipeline ..... 154
2.4.1 Pipeline Overview ..... 154
2.4.2 General Guidelines for VCU Pipeline Alignment ..... 154
2.4.3 Parallel Instructions ..... 155
2.4.4 Invalid Delay Instructions ..... 156
2.5 Instruction Set ..... 159
2.5.1 Instruction Descriptions ..... 159
2.5.2 General Instructions ..... 161
2.5.3 Arithmetic Math Instructions ..... 205

INSTRUMENTS
www.ti.com
2.5.4 Complex Math Instructions ..... 212
2.5.5 Cyclic Redundancy Check (CRC) Instructions ..... 271
2.5.6 Deinterleaver Instructions ..... 287
2.5.7 FFT Instructions ..... 303
2.5.8 Galois Instructions ..... 331
2.5.9 Viterbi Instructions ..... 344
2.6 Rounding Mode ..... 379
3 Trigonometric Math Unit (TMU) ..... 381
3.1 Overview ..... 382
3.2 Components of the C28x+FPU Plus TMU ..... 382
3.2.1 Interrupt Context Save and Restore ..... 382
3.3 Data Format ..... 382
3.4 Pipeline ..... 383
3.4.1 Pipeline and Register Conflicts ..... 383
3.4.2 Delay Slot Requirements ..... 385
3.4.3 Effect of Delay Slot Operations on the Flags ..... 386
3.4.4 Multi-Cycle Operations in Delay Slots ..... 386
3.4.5 Moves From FPU Registers to C28x Registers ..... 386
3.5 TMU Instruction Set ..... 388
3.5.1 Instruction Descriptions ..... 388
3.5.2 Common Restrictions ..... 389
3.5.3 Instructions ..... 389
Revision History ..... 403

## List of Figures

1-1. FPU Functional Block Diagram ..... 10
1-2. C28x With Floating-Point Registers ..... 14
1-3. Floating-point Unit Status Register (STF) ..... 16
1-4. Repeat Block Register (RB) ..... 18
1-5. FPU Pipeline ..... 19
2-1. C28x + VCU Block Diagram ..... 142
2-2. $\quad \mathrm{C} 28 \mathrm{x}+\mathrm{FPU}+\mathrm{VCU}$ Registers ..... 146
2-3. VCU Status Register (VSTATUS) ..... 149
2-4. Repeat Block Register (RB) ..... 152
2-5. C 28 x + FCU + VCU Pipeline ..... 154
List of Tables
1-1. $28 x$ Plus Floating-Point CPU Register Summary ..... 15
1-2. Floating-point Unit Status (STF) Register Field Descriptions ..... 16
1-3. Repeat Block (RB) Register Field Descriptions ..... 18
1-4. Operand Nomenclature ..... 27
1-5. Summary of Instructions ..... 29
2-1. Viterbi Decode Performance ..... 141
2-2. Complex Math Performance ..... 141
2-3. VCU Register Set ..... 147
2-4. 28x CPU Register Summary ..... 148
2-5. VCU Status (VSTATUS) Register Field Descriptions ..... 149
2-6. Operation Interaction With VSTATUS Bits ..... 150
2-7. Repeat Block (RB) Register Field Descriptions ..... 152
2-8. Operations Requiring a Delay Slot(s) ..... 155
2-9. Operand Nomenclature ..... 159
2-10. INSTRUCTION dest, source1, source2 Short Description ..... 160
2-11. General Instructions ..... 161
2-12. Arithmetic Math Instructions ..... 205
2-13. Complex Math Instructions ..... 212
2-14. CRC Instructions ..... 271
2-15. Deinterleaver Instructions ..... 287
2-16. FFT Instructions ..... 303
2-17. Galois Field Instructions ..... 331
2-18. Viterbi Instructions ..... 344
2-19. Example: Values Before Shift Right ..... 379
2-20. Example: Values after Shift Right ..... 379
2-21. Example: Addition with Right Shift and Rounding. ..... 379
2-22. Example: Addition with Rounding After Shift Right ..... 379
2-23. Shift Right Operation With and Without Rounding ..... 380
3-1. TMU Supported Instructions ..... 382
3-2. IEEE 32-Bit Single Precision Floating-Point Format ..... 382
3-3. Delay Slot Requirements for TMU Instructions ..... 385
3-4. Operand Nomenclature ..... 388
3-5. Summary of Instructions ..... 389

Read This First

This document describes the architecture, pipeline, and instruction sets of the TMU, VCU-II, and FPU accelerators.

## 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 $0 x$. For example, the following number is 40 hexadecimal (decimal 64): 40h or $0 x 40$.
- Registers in this document are shown as 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 Tl 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 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 DSP Peripheral Reference Guide describes the peripheral reference guides of the $28 x$ digital signal processors (DSPs).

SPRUFBO - TMS320x2833x, 2823x System Control and Interrupts Reference Guide describes the various interrupts and system control features of the $2833 x$ and $2823 x$ 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 DSC External Interface (XINTF) Reference Guide describes the XINTF, which is a nonmultiplexed asynchronous bus, as it is used on the $2833 x$ and $2823 x$ devices.

SPRU963 - TMS320x2833x, 2823x Boot ROM Reference 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) Reference Guide describes the McBSP available on the 2833x and 2823x devices. The McBSPs allow direct interface between a DSP and other devices in a system.

SPRUFB8 - TMS320x2833x, 2823x Direct Memory Access (DMA) Module Reference Guide describes the DMA on the 2833x and 2823x 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) Reference Guide 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) Module 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 - TMS320x2833x, 2823x Serial Communications 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 DSC 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) Module Reference Guide describes the features and operation of the inter-integrated circuit (I2C) module.

## Tools Guides-

SPRU513 - TMS320C28x Assembly Language Tools v5.0.0 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/C++ Compiler v5.0.0 User's Guide describes the TMS320C28x ${ }^{\text {TM }} \mathrm{C} / \mathrm{C}++$ compiler. This compiler accepts ANSI standard C/C++ source code and produces TMS320 DSP assembly language source code for the TMS320C28x device.
SPRU608 - 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 $\mathrm{C} 28 \mathrm{x}^{\mathrm{TM}}$ core.
SPRU625 - TMS320C28x DSP/BIOS 5.32 Application Programming Interface (API) Reference Guide describes development using DSP/BIOS.

## Trademarks

TMS320C28x, C28x, TMS320C2000 are trademarks of Texas Instruments.

## Floating Point Unit (FPU)

The TMS320C2000 ${ }^{\text {™ }}$ DSP family consists of fixed-point and floating-point digital signal controllers (DSCs). TMS320C2000 ${ }^{\text {TM }}$ Digital Signal Controllers combine control peripheral integration and ease of use of a microcontroller (MCU) with the processing power and C efficiency of Tl'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 Overview ..... 10
1.2 Components of the C28x plus Floating-Point CPU ..... 11
1.3 CPU Register Set ..... 13
1.4 Pipeline ..... 19
1.5 Floating Point Unit Instruction Set ..... 26

### 1.1 Overview

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-toregister 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.


### 1.1.1 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


### 1.1.1.1 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 C 28 x compiler V5.0, or later, is also required to generate C 28 x 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 floatingpoint. 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 floatingpoint 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.

### 1.2 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 1-5.
- 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 2 s -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.


### 1.2.1 Emulation Logic

The emulation logic is identical to that on the C28x fixed-point CPU. This logic includes the following features:

- 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 ESTOPO 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.

For more details about these features, refer to the TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430.

### 1.2.2 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 4 G words ( 1 word $=16$ bits) in data space and 4 M 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.

### 1.2.3 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 1 K words in size. M0 is mapped at addresses $0 \times 0000-0 \times 03 \mathrm{FF}$ and M1 is mapped at addresses $0 \times 0400-0 \times 07 F F$. Like all other memory blocks on the C $28 \mathrm{x}+\mathrm{FPU}$ 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.

### 1.2.4 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.

### 1.2.5 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.

### 1.2.5.1 Address and Data Buses

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

## - PAB: Program address bus

INSTRUMENTS

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.

### 1.2.5.2 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.

### 1.3 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

### 1.3.1 CPU Registers

Devices with the $\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}$ include the standard C 28 x 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 1-2 shows a diagram of both register sets and Table 1-1 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 1-2. C28x With Floating-Point Registers

Standard C28x Register Set

| ACC (32-bit) |
| :---: |
| P (32-bit) |
| XT (32-bit) |


| XAR0 (32-bit) |
| :--- |
| XAR1 (32-bit) |
| XAR2 (32-bit) |
| XAR3 (32-bit) |
| XAR4 (32-bit) |
| XAR5 (32-bit) |
| XAR6 (32-bit) |
| XAR7 (32-bit) |


| PC (22-bit) |
| :---: |
| RPC (22-bit) |
| DP (16-bit) |
| SP (16-bit) |


| ST0 (16-bit) |
| :---: |
| ST1 (16-bit) |
| IER (16-bit) |
| IFR (16-bit) |
| DBGIER (16-bit) |

Additional 32-bit FPU Registers


FPU Status Register (STF)

Repeat Block Register (RB)
FPU registers R0H - R7H and STF are shadowed for fast context save and restore

Table 1-1. 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 | 32 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 | $0 \times 080 B^{(1)}$ |
| 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.

### 1.3.1.1 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 C 28 x , program flow is controlled by C 28 x instructions that read status flags in the status register 0 (STO). If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into STO flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVSTO FLAGinstruction is used to load the current value of specified STF flags into the respective bits of STO. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified.

## Example 1-1. Moving STF Flags to the STO Register

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

Figure 1-3. Floating-point Unit Status Register (STF)


LEGEND: R/W = Read/Write; R = Read only; $-n=$ value after reset
Table 1-2. Floating-point Unit Status (STF) Register Field Descriptions

| Bits | Field | Value |  |
| :---: | :---: | :---: | :--- |
| 31 | SHDWS | ( <br> 1 <br> 1 | Shadow Mode Status Bit <br> This bit is forced to 0 by the RESTORE instruction. <br> This bit is set to 1 by the SAVE instruction. <br> 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 |

Table 1-2. Floating-point Unit Status (STF) Register Field Descriptions (continued)

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

### 1.3.1.2 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 1-2.

## Example 1-2. 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
    MOVSTO 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 1-4. 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: $\mathrm{R}=$ Read only; $-n=$ value after reset
Table 1-3. Repeat Block (RB) Register Field Descriptions

| Bits | Field | Value | Description |
| :---: | :---: | :---: | :--- | :--- |
| 31 | RAS |  | Repeat Block Active Shadow Bit <br> When an interrupt occurs the repeat active, RA, bit is copied to the RAS bit and the RA bit is cleared. <br> When an interrupt return instruction occurs, the RAS bit is copied to the RA bit and RAS is cleared. |
| 30 | RA | 0 | A repeat block was not active when the interrupt was taken. <br> A repeat block was active when the interrupt was taken. |
| $29-23$ | RSIZE | Repeat Block Active Bit <br> This bit is cleared when the repeat counter, RC, reaches zero. <br> When an interrupt occurs the RA bit is copied to the repeat active shadow, RAS, bit and RA is cleared. <br> When an interrupt return, IRET, instruction is executed, the RAS bit is copied to the RA bit and RAS is <br> cleared. <br> This bit is set when the RPTB instruction is executed to indicate that a RPTB is currently active. |  |
| 1 | Repeat Block Size <br> This 7-bit value specifies the number of 16-bit words within the repeat block. This field is initialized <br> when the RPTB instruction is executed. The value is calculated by the assembler and inserted into the <br> RPTB instruction's RSIZE opcode field. |  |  |
| 0-7 <br> Illegal block size. |  |  |  |
| $8 / 9-0 \times 7 \mathrm{~F}$ | A RPTB block that starts at an even address must include at least 9 16-bit words and a block that <br> starts at an odd address must include at least 816 -bit words. The maximum block size is 127 16-bit <br> words. The codegen assembler will check for proper block size and alignment. |  |  |

Texas
INSTRUMENTS

Table 1-3. Repeat Block (RB) Register Field Descriptions (continued)

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

### 1.4 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.

### 1.4.1 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 1-5. 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 1-5. FPU Pipeline


### 1.4.2 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 $M A C$ ) 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 1-3. ADDF32 is a 2 p instruction and therefore requires one delay slot. The destination register for the operation, ROH, will be updated one cycle after the instruction for a total of 2 cycles. Therefore, a NOP or instruction that does not use ROH 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.

Please note that on certain devices instructions make take additional cycles to complete under specific conditions. These exceptions will be documented in the device errata.

## Example 1-3. 2p Instruction Pipeline Alignment

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


### 1.4.3 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 1-4 and Example 1-5.

## Example 1-4. Floating-Point to C28x Register Software Pipeline Alignment

```
; MINF32: 32-bit floating-point minimum: single-cycle operation
; An alignment cycle is required before copying ROH to ACC
MINF32 R0H, R1H ; Single-cycle instruction
    ; <-- ROH is valid
NOP ; Alignment cycle
MOV32 @ACC, ROH ; 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 1-4. 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 1-5.

## Example 1-5. Floating-Point to C28x Register Software Pipeline Alignment

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


### 1.4.4 Moves from C28x Registers to FPU Registers

Transfers from the standard C28x CPU registers to the floating-point registers require four alignment cycles. For the 2833x, 2834x, 2806x, 28M35xx and 28M26xx, the four alignment cycles can be filled with NOPs or any non-conflicting instruction except for FRACF32, Ul16TOF32, I16TOF32, F32TOUI32, and F32TOI32. These instructions cannot replace any of the four alignment NOPs. On newer devices any nonconflicting instruction can go into the four alignment cycles. Please refer to the device errata for specific exceptions to these rules.

## Example 1-6. 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 ROH
NOP
NOP
NOP
NOP ; Wait 4 cycles
ADDF32 R2H,R1H,R0H ; R0H is valid
```


### 1.4.5 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 $2 p / 1$ instructions. The math portion of the operation takes two 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 1-7.

## Example 1-7. 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
NOP ; Any instruction
```

Parallel math instructions are referred to as $2 p / 2 p$ 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 1-8.

## Example 1-8. 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
```


### 1.4.6 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 1-9.

In Example 1-9 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 1-10.

INSTRUMENTS

## Example 1-9. 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 1-10. 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 1-11. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 1-13.

In Example 1-11 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 1-12.

## Example 1-11. Destination/Source Register Conflict

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


## Example 1-12. Destination/Source Register Conflict Resolved

```
; Valid delay instruction.
```

    MPYF32 R2H, R1H, R0H
    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 1-13 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 1-13. Parallel Instruction Destination/Source Exception

```
; Valid parallel operation.
```

    MPYF32 R2H, R1H, R0H ; 2p/1 instruction
    || MOV32 mem32, R2H ; <-- Uses R2H before the MPYF32
NOP $\quad ; \quad$ <-- mem32 updated
; <-- 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 1-14 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 1-14. 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 1-15.
If both operations within a parallel instruction try to update the same destination register as shown in Example 1-15 the assembler will issue an error.

## Example 1-15. 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 MOVSTO.

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

InsTRUMENTS

### 1.4.7 Optimizing the Pipeline

The following example shows how delay slots can be used to improve the performance of an algorithm. The example performs two $Y=M X+B$ operations. In Example 1-16, no optimization has been done. The $Y$ $=M X+B$ 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 1-16. 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
    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
    <
NOP ; Wait for ADDF32 to complete
    , <-- ADDF32 completes, R1H is valid
; 14 cycles
; 48 bytes
```

The code shown in Example 1-17 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 $\mathrm{Y}=\mathrm{MX}+\mathrm{B}$ calculations are now interleaved and both calculations complete in only nine cycles.

## Example 1-17. 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 ROH,@M2 ; Load ROH with M2 - single cycle
    MOV32 R1H,@X2 ; Load R1H with X2 - single cycle
    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
    ADDF32 R0H,R2H,R0H ; ROH = 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
```


### 1.5 Floating Point Unit 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).

### 1.5.1 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 C 28 x . 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 1-4. For information on the operands of standard C28x instructions, see the TMS320C28x DSP CPU and Instruction Set Reference Guide (SPRU430).

Table 1-4. 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
Description

Restrictions

Pipeline

Example

See Also

This section shows the opcode for the instruction.
Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed.

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

This section describes the instruction in terms of pipeline cycles as described in Section 1.4.

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.

Lists related instructions.

### 1.5.2 Instructions

The instructions are listed alphabetically, preceded by a summary.

## Table 1-5. Summary of Instructions

ABSF32 RaH, RbH - 32-bit Floating-Point Absolute Value ..... 31
ADDF32 RaH, \#16FHi, RbH - 32-bit Floating-Point Addition ..... 32
ADDF32 RaH, RbH, \#16FHi - 32-bit Floating-Point Addition ..... 34
ADDF32 RaH, RbH, RcH - 32-bit Floating-Point Addition ..... 36
ADDF32 RdH, ReH, RfH ||MOV32 mem32, RaH — 32-bit Floating-Point Addition with Parallel Move ..... 38
ADDF32 RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Addition with Parallel Move. ..... 40
CMPF32 RaH, RbH - 32-bit Floating-Point Compare for Equal, Less Than or Greater Than ..... 42
CMPF32 RaH, \#16FHi - 32-bit Floating-Point Compare for Equal, Less Than or Greater Than ..... 43
CMPF32 RaH, \#0.0 - 32-bit Floating-Point Compare for Equal, Less Than or Greater Than ..... 45
EINVF32 RaH, RbH - 32-bit Floating-Point Reciprocal Approximation ..... 46
EISQRTF32 RaH, RbH — 32-bit Floating-Point Square-Root Reciprocal Approximation ..... 48
F32TOI16 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Integer ..... 50
F32TOI16R RaH, RbH - Convert 32-bit Floating-Point Value to 16-bit Integer and Round ..... 51
F32TOI32 RaH, RbH — Convert 32-bit Floating-Point Value to 32-bit Integer ..... 52
F32TOUI16 RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer ..... 53
F32TOUI16R RaH, RbH — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round ..... 54
F32TOUI32 RaH, RbH —Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer ..... 55
FRACF32 RaH, RbH — Fractional Portion of a 32-bit Floating-Point Value ..... 56
I16TOF32 RaH, RbH — Convert 16-bit Integer to 32-bit Floating-Point Value ..... 57
I16TOF32 RaH, mem16 - Convert 16-bit Integer to 32-bit Floating-Point Value ..... 58
I32TOF32 RaH, mem32 -Convert 32-bit Integer to 32-bit Floating-Point Value ..... 59
I32TOF32 RaH, RbH —Convert 32-bit Integer to 32-bit Floating-Point Value ..... 60
MACF32 R3H, R2H, RdH, ReH, RfH - 32-bit Floating-Point Multiply with Parallel Add ..... 61
MACF32 R3H, R2H, RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Multiply and Accumulate with Parallel Move ..... 63
MACF32 R7H, R3H, mem32, *XAR7++ —32-bit Floating-Point Multiply and Accumulate ..... 65
MACF32 R7H, R6H, RdH, ReH, RfH - 32-bit Floating-Point Multiply with Parallel Add ..... 67
MACF32 R7H, R6H, RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Multiply and Accumulate with Parallel Move ..... 69
MAXF32 RaH, RbH - 32-bit Floating-Point Maximum ..... 71
MAXF32 RaH, \#16FHi - 32-bit Floating-Point Maximum ..... 72
MAXF32 RaH, RbH ||MOV32 RcH, RdH - 32-bit Floating-Point Maximum with Parallel Move ..... 73
MINF32 RaH, RbH - 32-bit Floating-Point Minimum ..... 74
MINF32 RaH, \#16FHi - 32-bit Floating-Point Minimum ..... 75
MINF32 RaH, RbH ||MOV32 RcH, RdH - 32-bit Floating-Point Minimum with Parallel Move ..... 76
MOV16 mem16, RaH —Move 16-bit Floating-Point Register Contents to Memory ..... 77
MOV32 *(0:16bitAddr), loc32 —Move the Contents of loc32 to Memory ..... 78
MOV32 ACC, RaH — Move 32-bit Floating-Point Register Contents to ACC ..... 79
MOV32 loc32, *(0:16bitAddr) —Move 32-bit Value from Memory to loc32 ..... 80
MOV32 mem32, RaH —Move 32-bit Floating-Point Register Contents to Memory ..... 81
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 1-5. 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
MOVSTO 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 ..... 105
MPYF32 RaH, RbH, RcH ||ADDF32 RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Add ..... 107
MPYF32 RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Multiply with Parallel Move ..... 109
MPYF32 RdH, ReH, RfH ||MOV32 mem32, RaH — 32-bit Floating-Point Multiply with Parallel Move ..... 111
MPYF32 RaH, RbH, RcH \|SUBF32 RdH, ReH, RfH — 32-bit Floating-Point Multiply with Parallel Subtract. ..... 112
NEGF32 RaH, RbH\{, CNDF\} - Conditional Negation ..... 113
POP RB —Pop the RB Register from the Stack ..... 114
PUSH RB —Push the RB Register onto the Stack ..... 116
RESTORE — Restore the Floating-Point Registers ..... 117
RPTB label, loc16 —Repeat A Block of Code ..... 119
RPTB label, \#RC - Repeat a Block of Code ..... 121
SAVE FLAG, VALUE — Save Register Set to Shadow Registers and Execute SETFLG ..... 123
SETFLG FLAG, VALUE —Set or clear selected floating-point status flags ..... 125
SUBF32 RaH, RbH, RcH - 32-bit Floating-Point Subtraction ..... 126
SUBF32 RaH, \#16FHi, RbH - 32-bit Floating Point Subtraction ..... 127
SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Subtraction with Parallel Move ..... 128
SUBF32 RdH, ReH, RfH ||MOV32 mem32, RaH — 32-bit Floating-Point Subtraction with Parallel Move ..... 130
SWAPF RaH, RbH\{, CNDF\} - Conditional Swap ..... 132
TESTTF CNDF - Test STF Register Flag Condition ..... 133
UI16TOF32 RaH, mem16 - Convert unsigned 16-bit integer to 32-bit floating-point value ..... 134
Ul16TOF32 RaH, RbH —Convert unsigned 16-bit integer to 32-bit floating-point value ..... 135
UI32TOF32 RaH, mem32 - Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value ..... 136
UI32TOF32 RaH, RbH —Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value ..... 137
ZERO RaH —Zero the Floating-Point Register RaH ..... 138
ZEROA —Zero All Floating-Point Registers ..... 139

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

## Operands

| RaH | floating-point destination register $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| RbH | floating-point source register (R0H to R 7 H$)$ |

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 < O) {RaH = -RbH}
    else {RaH = RbH}
```

Flags $\quad$ 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.


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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| \#16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The |
| RbH | low 16-bits of the mantissa are assumed to be all 0 . <br> floating-point source register ( ROH to R 7 H$)$ |

## Opcode

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


## Description

Flags
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 ( $0 \times 40000000$ ), 4.0 ( $0 \times 40800000$ ), 0.5 ( $0 \times 3 F 000000$ ), and -1.5 ( $0 x B F C 00000$ ). 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.
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 ; l 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
```

; Add to R1H the value 2.0 in 32-bit floating-point format
ADDF32 R0H, \#2.0, R1H ; R0H = 2.0 + R1H
ADDF32 R0H, \#2.0, R1H ; R0H = 2.0 + R1H
NOP ; Delay for ADDF32 to complete
NOP ; Delay for ADDF32 to complete
; <-- ADDF32 completes, ROH updated
; <-- ADDF32 completes, ROH updated
NOP ;
NOP ;
; Add to R3H the value -2.5 in 32-bit floating-point format
; Add to R3H the value -2.5 in 32-bit floating-point format
ADDF32 R2H, \#-2.5, R3H ; R2H = -2.5 + R3H
ADDF32 R2H, \#-2.5, R3H ; R2H = -2.5 + R3H
NOP ; Delay for ADDF32 to complete
NOP ; Delay for ADDF32 to complete
; <-- ADDF32 completes, R2H updated
; <-- ADDF32 completes, R2H updated
NOP ;
NOP ;
; Add to R5H the value 0x3FC00000 (1.5)
; Add to R5H the value 0x3FC00000 (1.5)
ADDF32 R5H, \#0x3FC0, R5H ; R5H = 1.5 + R5H
ADDF32 R5H, \#0x3FC0, R5H ; R5H = 1.5 + R5H
NOP ; Delay for ADDF32 to complete
NOP ; Delay for ADDF32 to complete
; <-- ADDF32 completes, R5H updated

```
    ; <-- ADDF32 completes, R5H updated
```

| www.ti.com | ADDF32 RaH, \#16FHi, RbH - 32-bit Floating-Point Addition |
| :--- | :--- |
| Nee 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 $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |
| \#16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The <br> low 16-bits of the mantissa are assumed to be all 0. |

## Opcode

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


## Description

Flags
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 ( $0 \times 40000000$ ), 4.0 ( $0 \times 40800000$ ), 0.5 ( $0 \times 3 F 000000$ ), and -1.5 ( $0 \times \mathrm{xBFC00000}$ ). 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.
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, ROH 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
    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
```

| www.ti.com | ADDF32 RaH, RbH, \#16FHi - 32-bit Floating-Point Addition |
| :--- | :--- |
| 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, RcH 32-bit Floating-Point Addition

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |
| RcH | floating-point source register (ROH to R7H) |

Opcode
LSW: 1110011100010000
MSW: 0000 000c ccbb baaa
Description
Add the contents of RcH to the contents of RbH and load the result into RaH .
$\mathrm{RaH}=\mathrm{RbH}+\mathrm{RcH}$
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 ( $2 p$ ). That is:


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

## Example

See also
Calculate $Y=M 1 * X 1+B 1$. This example assumes that $M 1, X 1, B 1$ and $Y$ are all on the same data page.

| MOVW | DP, \#M1 | ; Load the data page |
| :--- | :--- | :--- |
| MOV32 | ROH, @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 $\mathrm{Y}=\mathrm{A}+\mathrm{B}$
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 +R 1 H
MOVL XAR4, \#Y

MOV32 *XAR4,R0H ; Store the result
ADDF32 RaH, RbH, \#16FHi
ADDF32 RaH, \#16F, RbH
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

Flags $\quad$ 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
    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] ;
        SUBF32 R6H, R6H, R4H
        SUBF32 R3H, R1H, R7H
    MOV32 *+XAR5[2], R3H
        ADDF32 R4H, R7H, R1H
        | MOV32 *+XAR5[6], R6H
; <-- MOV32 completes, (B) stored
store R3H (C)
<-- MOV32 completes, (C) stored
<-- ADDF32 (D) completes, R4H valid
store R4H (D) ;
```

See also<br>ADDF32 RaH, \#16FHi, RbH<br>ADDF32 RaH, RbH, \#16FHi<br>ADDF32 RaH, RbH, RcH<br>MACF32 R3H, R2H, RdH, ReH, RfH<br>MPYF32 RaH, RbH, RcH || ADDF32 RdH, ReH, RfH<br>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 (ROH to R7H). <br> RdH cannot be the same register as RaH. |
| :--- | :--- |
| ReH | floating-point source register for the ADDF32 (R0H to R7H) <br> floating-point source register for the ADDF32 (ROH to R7H) |
| RaH | floating-point destination register for the MOV32 (ROH to R7H). <br> RaH cannot be the same register as RdH. <br> pointer to a 32-bit memory location. This is the source for the MOV32. |
| mem32 |  |

## Opcode

```
LSW: 111000110001 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 $=\operatorname{ReH}+\mathrm{RfH}$,
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 $(2 p)$ 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
```

$$
\text { Calculate } \mathrm{Y}=\mathrm{A}+\mathrm{B}-\mathrm{C}:
$$

```
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
SUBF32 R0H,R0H,R2H ; Subtract C from (A + B)
NOP ;
MOV32 *XAR4,R0H ; Store the result
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
```

See also

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

## Operands

| RaH | floating-point source register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |

Opcode
LSW: 1110011010010100
MSW: 00000000 00bb baaa

## Description Set ZF and NF flags on the result of $\mathrm{RaH}-\mathrm{RbH}$. 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 ROH, #5.0 ; ROH = 5.0 (0x40A00000)
    CMPF32 R1H, R0H ; ZF = 0, NF = 1
    CMPF32 R0H, R1H ; ZF = 0, NF = 0
    CMPF32 ROH, ROH ; 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
    MOVSTO ZF, NF ; Copy ZF and NF to STO 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

Instruments

## CMPF32 RaH, \#16FHi

32-bit Floating-Point Compare for Equal, Less Than or Greater Than

## Operands

| RaH | floating-point source register $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| $\# 16 \mathrm{FHi}$ | 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 OIII
MSW: IIII IIII IIII Iaaa
```


## Description

Flags
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 ( $0 x 40000000$ ), 4.0 ( $0 \times 40800000$ ), 0.5 ( $0 \times 3 F 000000$ ), 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.

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
Example

This is a single-cycle instruction

```
; Behavior of ZF and NF flags for different comparisons
    MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
    MOVIZF32 ROH, #5.0 ; ROH = 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
    MOVSTO ZF, NF ; Copy ZF and NF to STO 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 (ROH to R 7 H$)$ |
| :--- | :--- |
| $\# 0.0$ | zero |

## Opcode

## Description

Flags
LSW: 111001011010 Oaaa
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.

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 > #O.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 ROH, #5.0 ; ROH = 5.0 (0x40A00000)
    MOVIZF32 R1H, #-2.0 ; R1H = -2.0 (0xC0000000)
    MOVIZF32 R2H, #0.0 ; R2H = 0.0 (0x00000000)
    CMPF32 ROH, #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, #O.O ; Set/clear ZF and NF
    MOVSTO ZF, NF ; Copy ZF and NF to STO 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 $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| RbH | floating-point source register (R0H to R 7 H$)$ |

## Opcode

LSW: 1110011010010011
MSW: 00000000 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:

$$
\begin{aligned}
& \mathrm{Ye}=\operatorname{Estimate}(\mathrm{I} / \mathrm{X}) ; \\
& \mathrm{Ye}=\mathrm{Ye}(2.0-\mathrm{Ye} \mathrm{X}) \\
& \mathrm{Ye}=\mathrm{Ye} \mathrm{~K}^{\star}(2.0-\mathrm{Ye} \mathrm{X})
\end{aligned}
$$

After two 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 $1 / \mathrm{RbH}$
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
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 ROH, R0H, EQ ; Fixes sign for A/0.0
MPYF32 R2H, R2H, R3H ; R2H = Ye = Ye*(2.0 - Ye*B)
NOP
MPYF32 ROH, ROH, R2H ; ROH = 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 $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| RbH | floating-point source register (R0H to R 7 H$)$ |

Opcode
LSW: 1110011010010010
MSW: 00000000 00bb baaa

## Description

Flags
This operation generates an estimate of $1 /$ sqrt $(\mathrm{X})$ 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 )
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 ( $2 p$ ) 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:
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 ROH, R0H, R1H ; R0H = Y = X*Ye = sqrt(X)
LRETR
EINVF32 RaH, RbH
```

See also

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

## Operands

| RaH | floating-point destination register (ROH to R7H) <br> RbH |
| :--- | :--- |

Opcode LSW: 1110011010001100

Description Convert a 32-bit floating point value in RbH to a 16-bit integer and truncate. The result will be stored in RaH.
$\operatorname{RaH}(15: 0)=\operatorname{F32TOI16}(\mathrm{RbH})$
$\operatorname{RaH}(31: 16)=$ sign extension of $\operatorname{RaH}(15)$
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 ( $2 p$ ) 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)
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)
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 |
| F32TOUU116R RaH, RbH |  |
|  | I16TOF32 RaH, RbH |
|  | I16TOF32 RaH, mem16 |
|  | Ul16TOF32 RaH, mem16 |
|  | Ul16TOF32 RaH, RbH |

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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |

Opcode LSW: 1110011010001100

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 .
$\operatorname{RaH}(15: 0)=$ F32ToI16round (RbH)
$\operatorname{RaH}(31: 16)=$ sign extension of $\operatorname{RaH}(15)$
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 ( $2 p$ ) instruction. That is:

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

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
    ; ROH = 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)
    ; 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 |
|  | Ul16TOF32 RaH, mem16 |
|  | Ul16TOF32 RaH, RbH |

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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |

Opcode LSW: 1110011010001000

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)
R5H = F32TOI32 (R4H)
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 |
|  | Ul32TOF32 RaH, mem32 |

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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |

Opcode LSW: 1110011010001110

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.
$\operatorname{RaH}(15: 0)=\operatorname{F32ToUI16(RbH)} \operatorname{RaH}(31: 16)=0 \times 0000$

## Flags <br> This instruction does not affect any flags:

| Flag | TF | ZI | NI | ZF | NF | LUF | LVF |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Modified | No | No | No | No | No | No | No |

## Pipeline

## Example

```
F32TOI16 RaH, RbH
F32TOUI16R RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
Ul16TOF32 RaH, mem16
Ul16TOF32 RaH, RbH
```

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) $=9.0(0 \times 0009)$
R5H $(31: 16)=0.0(0 \times 0000)$
F32TOUI16 R7H, R6H ; R7H (15:0) = F32TOUI16 (R6H)
R7H $(31: 16)=0 \times 0000$
NOP ; 1 Cycle delay for F32TOUI16 to complete
; <-- F32TOUI16 complete, R7H (15:0) $=0.0(0 \times 0000)$
R7H $(31: 16)=0.0(0 \times 0000)$

## See also

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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |



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.


See also |  | F32TOI16 RaH, RbH |
| :--- | :--- |
|  | F32TOI16R RaH, RbH |
|  | F32TOUI16 RaH, RbH |
|  | I16TOF32 RaH, RbH |
|  | I16TOF32 RaH, mem16 |
|  | Ul16TOF32 RaH, mem16 |
|  | Ul16TOF32 RaH, RbH |

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

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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |

Opcode LSW: 1110011010001010

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 $\quad$ 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 ( $2 p$ ) 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
Ul32TOF32 RaH, RbH
Ul32TOF32 RaH, mem32

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

## Operands

| RaH | floating-point destination register (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |


| Opcode | LSW: 1110011011110001 |
| :--- | :--- |
|  | MSW: 00000000 bbb baaa |

Description Returns in RaH the fractional portion of the 32-bit floating-point value in RbH
Flags $\quad$ This instruction does not affect any flags:

| Flag | TF | ZI | NI | ZF | NF | LUF | LVF |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Modified | No | No | No | No | No | No | No |

Pipeline $\quad$ 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

INSTRUMENTS

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

## Operands

| RaH | floating-point destination register (ROH to R7H) <br> floating-point source register (ROH to R7H) |
| :--- | :--- |

Opcode LSW: 1110011010001101

Description Convert the 16-bit signed integer in RbH to a 32-bit floating point value and store the result in RaH.
$\mathrm{RaH}=$ 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 $\quad$ 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

See also

```
MOVIZ ROH, #0x0000 ; ROH[31:16] = 0.0 (0x0000)
MOVXI ROH, #0x0004 ; ROH[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)
```

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, mem16
Ul16TOF32 RaH, mem16
Ul16TOF32 RaH, RbH

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

## Operands

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

Opcode LSW: 1110001011001000
MSW: 0000 Oaaa mem16
Description Convert the 16-bit signed integer indicated by the mem16 pointer to a 32-bit floatingpoint 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
Ul16TOF32 RaH, mem16
Ul16TOF32 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 <br> using any of the direct or indirect addressing modes supported by the C28x CPU |

## Opcode

## Description

Flags

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

Convert the 32-bit signed integer indicated by the mem32 pointer to a 32-bit floating point value and store the result in RaH .
$\mathrm{RaH}=$ I32ToF32[mem32]

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 ( $2 p$ ) 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

## See also

```
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)
```

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, RbH
UI32TOF32 RaH, RbH
Ul32TOF32 RaH, mem32

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

## Operands

| RaH | floating-point destination register (ROH to R 7 H$)$ <br> floating-point source register (ROH to R7H) |
| :--- | :--- |

Opcode LSW: 1110011010001001
MSW: 00000000 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
Ul32TOF32 RaH, RbH
Ul32TOF32 RaH, mem32

INSTRUMENTS

## 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) <br> RdH cannot be R3H |
| ReH | floating-point source register for MPYF32 operation (R0H to R7H) <br> RfH |

Opcode LSW: 11100111010000 ff
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

See also

```
; 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
;
;
```

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
; $\mathrm{R} 3 \mathrm{H}=\mathrm{B}=\mathrm{X} 1$ * Y 1
MPYF32 R3H, R0H, R1H ; In parallel R0H = X2
|| MOV32 ROH, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y2
; $\mathrm{R} 3 \mathrm{H}=\mathrm{A}+\mathrm{B}$
; $\mathrm{R} 2 \mathrm{H}=\mathrm{C}=\mathrm{X} 2$ * Y 2
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H $=\mathrm{X} 3$
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = Y3
; $\mathrm{R} 3 \mathrm{H}=(\mathrm{A}+\mathrm{B})+\mathrm{C}$
; $\mathrm{R} 2 \mathrm{H}=\mathrm{D}=\mathrm{X} 3$ * Y3
MACF32 R3H, R2H, R2H, R0H, R1H ; In parallel R0H $=\mathrm{X} 4$
|| MOV32 R0H, *XAR4
MOV32 R1H, *XAR5 ; R1H = Y4
; The next MACF32 is an alias for
; MPYF32 || ADDF32
; $\mathrm{R} 2 \mathrm{H}=\mathrm{E}=\mathrm{X} 4$ * Y 4
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

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

## 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 <br> RdH cannot be the same register as RaH <br> ReH <br> floating-point source register (R0H to R7H) for the multiply operation <br> RaH |
|  | floating-point source register (R0H to R7H) for the multiply operation <br> floating-point destination register for the MOV32 operation (R0H to R7H). <br> mem32 |

Opcode LSW: 111000110011 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,
$\mathrm{RdH}=\mathrm{ReH} * \mathrm{RfH}$,
$\mathrm{RaH}=$ [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 $\quad$ 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
    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.


## 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, XAR7 is post incremented. |

## Opcode LSW: 1110001001010000

MSW: 00011111 mem32
Description Perform a multiply and accumulate operation. When used as a standalone 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 $3+\mathrm{N}$ cycles where N is the number of times the instruction is repeated. When repeated, this instruction has the following pipeline restrictions:


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

```
<instruction1> ; No restriction
<instruction2> ; Cannot be a 2p instruction that writes
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 (ROH to R7H) <br> RdH cannot be R3H |
| ReH | floating-point source register for MPYF32 operation (R0H to R7H) <br> RfH |

Opcode

Description

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 accumula | te operations: |
| :---: | :---: | :---: |
|  | ; 1st multiply: $\mathrm{A}=\mathrm{xo} \mathrm{*} \mathrm{yo}$ |  |
|  | ; 2nd multiply: $\mathrm{B}=\mathrm{x} 1$ * Y 1 |  |
|  | ; 3rd multiply: $\mathrm{C}=\mathrm{x} 2$ * Y2 |  |
|  | ; 4th multiply: D = x3 * Y3 |  |
|  | 5th multiply: $\mathrm{E}=\mathrm{X} 3$ * Y3 |  |
|  | ; ${ }^{\text {c }}$ |  |
|  | Result $=\mathrm{A}+\mathrm{B}+\mathrm{C}+\mathrm{D}+\mathrm{E}$ |  |
|  | Mov32 Roh, *XAR4++ | ; $\mathrm{ROH}=\mathrm{xO}$ |
|  | Mov32 R1H, *XAR5++ | ; R1H = Y0 |
|  |  | ; $\mathrm{R} 6 \mathrm{H}=\mathrm{A}=\mathrm{x} 0$ * y 0 |
|  | MPYF32 R6H, R0H, R1H | ; In parallel $\mathrm{ROH}=\mathrm{x} 1$ |
|  | \|| Mov32 R0H, *XAR4++ |  |
|  | MOV32 R1H, *XAR5++ | ; $\mathrm{R} 1 \mathrm{H}=\mathrm{Y} 1$ |
|  |  | ; $\mathrm{R} 7 \mathrm{H}=\mathrm{B}=\mathrm{X1} * \mathrm{Y} 1$ ( In parallel $\mathrm{ROH}=\mathrm{X} 2$ |
|  | MPYF32 R7H, ROH, R1H <br> \|| MOV32 ROH, *XAR4++ | ; In parallel ROH = X 2 |
|  | Mov32 R1H, *XAR5++ | ; $\mathrm{R} 1 \mathrm{H}=\mathrm{Y} 2$ |
|  | Hov3 Rin, Xhat | ; $\mathrm{R} 7 \mathrm{H}=\mathrm{A}+\mathrm{B}$ |
|  |  | R6H $=\mathrm{C}=\mathrm{x} 2$ * Y 2 |
|  | MACF32 R7H, R6H, R6H, R0H, R1H | ; In parallel $\mathrm{ROH}=\mathrm{x} 3$ |
|  | \|| MOV32 ROH, *XAR4++ |  |
|  | MOV32 R1H, *XAR5++ | ; R1H $=$ Y3 |
|  |  | ; $\mathrm{R} 7 \mathrm{H}=(\mathrm{A}+\mathrm{B})+\mathrm{C}$ |
|  |  | ; R6H = D = x3 * Y3 |
|  |  | ; In parallel ROH = X 4 |
|  | \|| MOV32 ROH, *XAR4 |  |
|  | MOV32 R1H, *XAR5 | ; $\mathrm{R1} \mathrm{H}=\mathrm{Y} 4$ |
|  | ; Next MACF32 is an alias for |  |
|  | ; MPYF32 \|| ADDF32 |  |
|  | MACF32 R7H, R6H, R6H, R0H, R1H | ; $\mathrm{R} 6 \mathrm{H}=\mathrm{E}=\mathrm{X} 4$ * Y 4 <br> ; in parallel R7H $=(A+B+C)+D$ |
|  | nop | ; Wait for MPYF32 \|| ADDF32 to complete |
|  | ADDF32 R7H, R7H, R6H | ; R7H $=(\mathrm{A}+\mathrm{B}+\mathrm{C}+\mathrm{D})+\mathrm{E}$ |
|  | NOP | ; Wait for ADDF32 to complete |
|  | MOV32 @Result, R7H | ; Store the result |
| See also | MACF32 R3H, R2H, RdH, ReH, RfH <br> MACF32 R3H, R2H, RdH, ReH, RfH \|| MOV32 RaH, mem32 <br> MACF32 R7H, R3H, mem32, *XAR7++ <br> MACF32 R7H, R6H, RdH, ReH, RfH \|| MOV32 RaH, mem32 <br> 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. <br> RdH cannot be the same register as RaH. |
| ReH | floating-point source register (R0H to R7H) for the multiply operation <br> RfH |
| RaH | floating-point source register (R0H to R7H) for the multiply operation <br> floating-point destination register for the MOV32 operation (R0H to R7H). <br> RaH cannot be R3H or the same as RdH. <br> mem32 |

Opcode LSW: 111000111100 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 $\quad$ 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 ( $2 p$ ) and the MOV32 takes a single cycle. That is:



## 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 R 7 H$)$ |

Opcode LSW: 1110011010010110
Description if $(\mathrm{RaH}<\mathrm{RbH}) \mathrm{RaH}=\mathrm{RbH}$

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.

$$
\begin{aligned}
& \text { if }(\operatorname{RaH}==\operatorname{RbH})\left\{\begin{array} { l } 
{ \mathrm { ZF } = 1 , } \\
{ \mathrm { NF } = 0 \} } \\
{ \text { if } ( \mathrm { RaH } > \operatorname { R b H } ) } \\
{ \text { if } ( \mathrm { RaH } < \mathrm { ZF } = 0 , \mathrm { NF } ) }
\end{array} \left\{\begin{array}{l}
\mathrm{ZF}=0, \\
\mathrm{ZFF}=1\}
\end{array}\right.\right.
\end{aligned}
$$

Pipeline
This is a single-cycle instruction.

## Example

| MOVIZF32 | R0H, \#5.0 | ; R0H $=5.0 \quad(0 \times 40 A 00000)$ |
| :--- | :--- | :--- | :--- |
| MOVIZF32 | R1H, \#-2.0 | ; R1H $=-2.0(0 \times C 0000000)$ |
| MOVIZF32 | R2H, \#-1.5 | ; R2H $=-1.5(0 \times B F C 00000)$ |
| MAXF32 | R2H, R1H | ; R2H $=-1.5, \mathrm{ZF}=\mathrm{NF}=0$ |
| MAXF32 | R1H, R2H | ; R1H $=-1.5, \mathrm{ZF}=0, \mathrm{NF}=1$ |
| MAXF32 | R2H, R0H | ; R2H $=5.0, \mathrm{ZF}=0, \mathrm{NF}=1$ |
| MAXF32 | R0H, R2H | ; R2H $=5.0, \mathrm{ZF}=1, \mathrm{NF}=0$ |

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 <br> 32-bit Floating-Point Maximum

## Operands

| RaH | floating-point source/destination register (R0H to R 7 H$)$ |
| :--- | :--- |
| $\# 16 \mathrm{FHi}$ | 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

## Description

Flags

```
LSW: 1110 1000 0010 OIII
MSW: IIII IIII IIII Iaaa
```

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 ( $0 \times 40000000$ ), $4.0(0 \times 40800000)$, 0.5 ( $0 \times 3 F 000000$ ), 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.

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 | ROH, \#5.0 | ; ROH = 5.0 | (0x40A00000) |
| :---: | :---: | :---: | :---: | :---: |
|  | movizf32 | R1H, \#4.0 | ; R1H $=4.0$ | (0x40800000) |
|  | movizf32 | R2H, \#-1.5 | ; R2H $=-1.5$ | (0xBFC00000) |
|  | MAXF32 | ROH, \#5.5 | ; ROH = 5.5 | $5, \mathrm{ZF}=0, \mathrm{NF}=$ |
|  | MAXF32 | R1H, \#2.5 | ; R1H $=4.0$ | , $\mathrm{ZF}=0, \mathrm{NF}=$ |
|  | MAXF32 | R2H, \#-1.0 | ; R2H $=-1.0$ | , $\mathrm{ZF}=0, \mathrm{NF}=$ |
|  | MAXF32 | R2H, \#-1.0 | ; R2H $=-1.5$ | , $\mathrm{ZF}=1, \mathrm{NF}=$ |
| See also | MAXF32 RaH, RbH |  |  |  |
|  | MAXF32 RaH, RbH \|| MOV32 RcH, RdHMINF32 RaH, RbHMINF32 RaH, \#16FHi |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Instruments

## 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 (ROH to R7H) <br> RaH cannot be the same register as RcH |
| :--- | :--- |
| RbH |  |
| floating-point source register for the MAXF32 operation (ROH to R7H) |  |
| floating-point destination register for the MOV32 operation (ROH to R7H) |  |
| RdH cannot be the same register as RaH |  |

Opcode LSW: 1110011010011100

## Description <br> If RaH is less than RbH , then load RaH with RbH . Thus RaH will always have the

Restrictions

Flags 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.

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

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

## See also

MAXF32 RaH, RbH
MAXF32 RaH, \#16FHi

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

## Operands

| RaH | floating-point source/destination register $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| RbH | floating-point source register $(\mathrm{ROH}$ to R 7 H$)$ |


| Opcode | LSW: 1110011010010111 |
| :--- | :--- |
|  | MSW: 00000000 00bb baaa |
| Description | if $(\mathrm{RaH}>\mathrm{RbH}) \quad \mathrm{RaH}=\mathrm{RbH}$ |

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

## Example

See also

This is a single-cycle instruction.


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 R 7 H$)$ |
| :--- | :--- |
| $\# 16 \mathrm{FHi}$ | 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

## Description

Flags
LSW: 111010000011 OIII
MSW: IIII IIII IIII Iaaa
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 $2.0(0 \times 40000000), 4.0(0 \times 40800000), 0.5(0 \times 3 F 000000)$, 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.

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 | movizf 32 | R0H, \#5.0 | ; $\mathrm{ROH}=5.0$ | (0x40A00000) |
| :---: | :---: | :---: | :---: | :---: |
|  | MOVIZF32 | R1H, \#4.0 | ; R1H = 4.0 | (0x40800000) |
|  | Movizf32 | R2H, \#-1.5 | ; $\mathrm{R} 2 \mathrm{H}=-1.5$ | (0xBFC00000) |
|  | MINF32 | R0H, \#5.5 | ; ROH = 5.0 | , $\mathrm{zF}=0, \mathrm{NF}=$ |
|  | MINF32 | R1H, \#2.5 | ; R1H = 2.5 | , $\mathrm{zF}=0, \mathrm{NF}=0$ |
|  | MINF32 | R2H, \#-1.0 | ; R2H $=-1.5$ | , $\mathrm{ZF}=0, \mathrm{NF}=$ |
|  | minf3 2 | R2H, \#-1.5 | ; $\mathrm{R} 2 \mathrm{H}=-1.5$ | , $\mathrm{ZF}=1, \mathrm{NF}=$ |
| See also |  |  |  |  |
|  | MAXF32 RaH, \#16FHi MAXF32 RaH, RbH |  |  |  |
|  | MINF32 RaH, RbH |  |  |  |
|  | MINF32 RaH, RbH \|| MOV32 RcH, RdH |  |  |  |

Texas
InsTruments

## 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) <br> RaH cannot be the same register as RcH <br> fbH <br> RcH |
| :--- | :--- |
| floating-point source register for the MIN32 operation (R0H to R7H) <br> floating-point destination register for the MOV32 operation (R0H to R7H) <br> RcH cannot be the same register as RaH |  |


| Opcode | LSW: 1110011010011101 |
| :--- | :--- |
| MSW: 0000 dddc ccbb baaa |  |
| Description | if $(\mathrm{RaH}>\mathrm{RbH}) \quad\{\mathrm{RaH}=\mathrm{RbH} ; \mathrm{RcH}=\mathrm{RdH} ;\}$ |

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
Example

## See also

This is a single-cycle instruction.


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 <br> faH |
| :--- | :--- |

Opcode LSW: 1110001000010011

Description Move 16-bit value from the lower 16-bits of the floating-point register ( $\mathrm{RaH}[15: 0])$ to the location pointed to by mem16.
[mem16] $=\operatorname{RaH}[15: 0]$
Flags $\quad$ 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 @O, 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

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

Opcode
LSW: 10111101 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, ${ }^{*}(0: 16$ bitAddr $)$ |

Instruments

## MOV32 ACC, RaH

## Move 32-bit Floating-Point Register Contents to ACC

## Operands

| ACC | 28x accumulator <br> RaH <br> floating-point source register (ROH 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 floatingpoint register indicated by RaH .
$\mathrm{ACC}=\mathrm{RaH}$
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 (STO) of the $28 x$ CPU are affected.

## Pipeline $\quad$ 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,ROH ; Copy ROH 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

```
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
MOVIZF32 ROH, #2.5 ; ROH = 2.5 = 0x40200000
F32TOUI32 R0H, R0H
NOP ; Delay for conversion instruction
    ; < -- Conversion complete, ROH 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 |
| :--- | :--- |
| $0: 16$ bitAddr | 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 $\quad$ No STF flags are affected. If loc32 is the ACC register, then the $Z$ and $N$ flag in status register zero (STO) 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

See also MOV32 RaH, mem32\{, CNDF\}

```
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
```

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

Instruments

## 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: 1110001000000011
MSW: 0000 Oaaa mem32

## Description Move from memory to STF.

[mem32] $=$ RaH
Flags $\quad$ 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

## Example

This is a single-cycle instruction.

```
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++ ; ROH = X0
    MOV32 R1H, *XAR5++ ; R1H = Y0
    MPYF32 R6H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
    MOV32 R1H, *XAR5++ ; R1H = 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
    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 | $; R 7 H=(A+B+C+D)+E$ 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: 1110001000000000
MSW: 00000000 mem32
Description Copy the floating-point status register, STF, to memory.
[mem32] $=$ STF
Flags $\quad$ 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

## Example 1

This is a single-cycle instruction.

```
MOVW DP, #0x0280 ; DP = 0x0280
    MOVIZF32 R0H, #2.0 ; ROH = 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
MOV32 mem32, RaH
MOV32 *(0:16bitAddr), loc32
MOVSTO FLAG
```

See also

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

## Operands

| P | 28x product register P <br> floating-point source register (ROH to R 7 H$)$ |
| :--- | :--- |



Pipeline

| Example | MOVIZF32 R0H, \#2.5 | ; $\mathrm{ROH}=2.5=0 \times 40200000$ |
| :---: | :---: | :---: |
|  | F32TOUI32 ROH, ROH |  |
|  | NOP | ; Delay for conversion instruction <br> ; <-- Conversion complete, ROH valid |
|  | NOP | ; Alignment cycle |
|  | MOV32 P, R0H | ; $\mathrm{P}=2=0 \times 0000002$ |
| 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 (ROH to R7H) |
| :--- | :--- |
| acc | accumulator |

Opcode
LSW: 10111101 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 $\quad$ While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard $28 x$ CPU register to a floating-point register. The four alignment cycles can be filled with any non-conflicting instructions except for the following: FRACF32, Ul16TOF32, 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 ;
; <-- ROH is valid
```

Example

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 (ROH to R7H) |
| :--- | :--- |
| mem32 | pointer to the 32-bit source memory location |
| CNDF | optional condition. |

Opcode LSW: 111000101010 CNDF
MSW: 0000 Oaaa mem32
Description
If the condition is true, then move the 32-bit value referenced by mem32 to the floatingpoint register indicated by RaH .
if (CNDF == TRUE) $\mathrm{RaH}=$ [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 $\mathrm{ZF}, \mathrm{NF}, \mathrm{ZI}$, 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.


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

Operands

| RaH | floating-point register (ROH 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 $\quad$ 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 cycles are required after any copy from a stan floating-point register. The four alignment cycles can be instructions except for the following: FRACF32, UI16TOF and F32TOI32. |
| :---: | :---: |
| Example |  |
| 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 (ROH to R7H) |
| :--- | :--- |
| RbH | floating-point source register (ROH to R7H) |
| CNDF | optional condition. |

## Opcode LSW: 111001101100 CNDF

MSW: 00000000 OObb baaa

## Description

If the condition is true, then move the 32-bit value referenced by mem32 to the floatingpoint 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 |

$$
\begin{aligned}
& \text { if }(\mathrm{CNDF}==\mathrm{UNCF}) \quad\{\mathrm{NF}=\mathrm{RaH}(31) ; \mathrm{ZF}=0 ; \\
& \text { if }(\operatorname{RaH}[30: 23]==0) \quad\{\mathrm{ZF}=1 ; \mathrm{NF}=0 ;\} \mathrm{NI}=\operatorname{RaH}(31) ; \mathrm{ZI}=0 ; \\
& \text { if }(\operatorname{RaH}[31: 0]==0) \mathrm{ZI}=1 ;\} \text { else No flags modified; }
\end{aligned}
$$

Pipeline
This is a single-cycle instruction.

## Example

See als

$$
\begin{aligned}
& \text { MOVIZF32 R3H, \#8.0 ; R3H = 8.0 (0x41000000) } \\
& \text { MOVIZF32 R4H, \#7.0 ; R4H = 7.0 (0x40E00000) } \\
& \text { MAXF32 R3H, R4H ; } \mathrm{ZF}=0, \mathrm{NF}=0 \\
& \text { MOV32 R1H, R3H, GT ; R1H = 8.0 ( } 0 \times 41000000 \text { ) } \\
& \text { MOV32 RaH, mem32\{, CNDF\} }
\end{aligned}
$$

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

## Operands

| RaH <br> XARn | floating-point register (ROH to R7H) <br> auxiliary register (XARO - XAR7) |
| :--- | :--- |

Opcode
LSW: 10111101 loc32
MSW: IIII IIII IIII IIII
Description Move the 32-bit value in the auxiliary register XARn to the floating point register RaH. $\mathrm{RaH}=\mathrm{XARn}$

Flags $\quad$ 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 $\quad$ While this is a single-cycle instruction, additional pipeline alignment is required. Four alignment cycles are required after any copy from a standard 28 x 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 |
| MOVL XAR1, \#0x0200 ; XAR1 $=512$ |  |
| MOV32 R0H, XAR1 |  |
| NOP |  |
| NOP |  |
| NOP |  |
| NOP |  |
| UI32TOF32 R0H, R0H ; R0H $=512.0(0 \times 44000000)$ |  |

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 (ROH to R7H) <br> XT |
| :--- | :--- |

Opcode
LSW: 10111101 loc32
MSW: IIII IIII IIII IIII
Description Move the 32-bit value in temporary register, XT, to the floating-point register RaH.
RaH $=x T$
Flags $\quad$ 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 alig alignment cycles are required after any copy from a standard floating-point register. The four alignment cycles can be filled instructions except for the following: FRACF32, UI16TOF32, 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 ; |
|  | ; <-- ROH is valid |
|  | ADDF32 R2H,R1H,R0H ; Instruction can use R0H as a sourc |
| Example | MOVIZF32 R6H, \#5.0 ; R6H $=5.0$ ( $0 \times 40 \mathrm{~A} 00000$ ) |
|  | NOP ; 1 Alignment cycle |
|  | MOV32 XT, R6H ; XT = 5.0 (0x40A00000) |
|  | MOV32 R1H, XT ; R1H = 5.0 ( $0 \times 40 \mathrm{~A} 00000$ ) |
| 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 $\quad$ 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

## Example 1

Example 2

See also
This is a single-cycle instruction.

 MOV32 STF, *--SP ; Restore STF from stack

MOV32 mem32, STF MOVSTO FLAG

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

## Operands

| XARn | 28x auxiliary register (XARO - XAR7) <br> floating-point source register (ROH to R7H) |
| :--- | :--- |



Pipeline

## Example

## See also

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 |
| MOVIZF32 R0H, \#2.5 | ; $\mathrm{ROH}=2.5=0 \times 40200000$ |
| F32TOUI32 R0H, R0H |  |
| NOP | ; Delay for conversion instruction |
|  | ; <-- Conversion complete, ROH valid |
| NOP | ; Alignment cycle |
| MOV32 XAR0, R0H | ; $\mathrm{XARO}=2=0 \times 00000002$ |

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 (ROH 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.
$\mathrm{XT}=\mathrm{RaH}$
Flags No flags affected in floating-point unit.

| Flag | TF | ZI | NI | ZF | NF | LUF | LVF |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| Modified | No | No | No | No | No | No | No |



## Example

| MOVIZF32 R0H, \#2.5 | ; $\mathrm{R} 0 \mathrm{H}=2.5=0 \times 40200000$ |
| :---: | :---: |
| F32TOUI32 R0H, R0H |  |
| NOP | ; Delay for conversion instruction |
|  | ; <-- Conversion complete, R0H valid |
| NOP | ; Alignment cycle |
| MOV32 XT, R0H | ; $\mathrm{XT}=2=0 \times 00000002$ |
| 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 (ROH to R7H) |
| :--- | :--- |
| mem32 | pointer to the 32-bit source memory location |

Opcode LSW: 1110001000100011
MSW: 0000 Oaaa mem32
Description Move the 32-bit value referenced by mem32 to the floating-point register indicated by RaH.
$\mathrm{RaH}=$ [mem32] [mem32+2] $=$ [mem32]
Flags $\quad$ 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
Example

See also

This is a single-cycle instruction.
MOVW DP, \#0x02C0 ; DP $=0 \times 02 \mathrm{C} 0$
MOV @2, \#0x0000 ; [0x00B002] $=0 \times 0000$
MOV @3, \#0x4110 ; [0x00B003] $=0 \times 4110$
MOVD32 R7H, @2 ; R7H $=0 \times 41100000$,
; $[0 \times 00 \mathrm{~B} 004]=0 \times 0000,[0 \times 00 \mathrm{BO} 05]=0 \times 4110$
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: 111010000000 OIII (opcode of MOVIZ RaH, \#16FHiHex) MSW: IIII IIII IIII Iaaa |
|  | LSW: 111010000000 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 floatingpoint 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 IEEE 32-bit floating-point format of \#32F are zeros, then the convert MOVF32 into only MOVIZ instruction. If the lower 16-bits of floating-point format of \#32F are not zeros, then the assembler will MOVIZ and MOVXI instructions. |
| :---: | :---: |
| Example | MOVF32 R1H, \#3.0 $;$ R1H $=3.0(0 \times 40400000)$ <br>  $;$ Assembler converts this instruction as <br>  $;$ MOVIZ R1H, \#0x4040 |
|  |  |
|  | ```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 |

INSTRUMENTS

## 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 $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| \#32FHex | A 32-bit immediate value that represents an IEEE 32-bit floating-point value. |

## Opcode

```
LSW: 1110 1000 0000 OIII (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 $\# 0 \times 40400000$. \#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 16bits 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 $=0 \times 40400000$ <br>  $;$ Assembler converts this instruction as <br>  $;$ 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 $=0 \times 00004040$ <br>  $;$ Assembler converts this instruction as <br>  $;$ 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 $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| $\# 16 \mathrm{FHiHex}$ | 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 OIII
MSW: IIII IIII IIII Iaaa
```


## Description <br> Note: This instruction only accepts a hex value as the immediate operand. To specify the

Flags 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 32bit 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 ( $0 \times 40000000$ ), 4.0 ( $0 \times 40800000$ ), 0.5 ( $0 \times 3 F 000000$ ), and -1.5 ( $0 \times B F C 00000$ ). If a constant requires all 32bits of a floating-point register to be initialized, then use MOVIZ along with the MOVXI instruction.

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

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 $\quad$ This is a single-cycle instruction.

## Example

```
; Load R0H with -1.5 (0xBFC00000)
    MOVIZ ROH, #OxBFCO ; ROH = 0xBFCOOOOO
; Load ROH with pi = 3.141593 (0x40490FDB)
    MOVIZ ROH, #Ox4049 ; ROH = 0x40490000
    MOVXI ROH, #Ox0FDB ; ROH = 0x40490FDB
```

See also MOVIZF32 RaH, \#16FHi
MOVXI RaH, \#16FLoHex

Instruments

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

## Operands

| RaH | floating-point register $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| $\# 16 \mathrm{FHi}$ | 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 OIII
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 ( $0 \times 40000000$ ), 4.0 ( $0 \times 40800000$ ), 0.5 ( $0 \times 3 F 000000$ ), and -1.5 ( $0 x B F C 00000$ ). \#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, 0xBFCO.

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

Flags $\quad$ 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 ROH, #3.0 ; ROH = 3.0 = 0x40400000
MOVIZF32 R1H, #1.0 ; R1H = 1.0 = 0\times3F800000
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 ROH with a more accurate pi = 3.141593 (0x40490FDB)
    MOVIZF32 R0H,#0x4049 ; ROH = 0x40490000
    MOVXI ROH,#OxOFDB ; ROH = 0x40490FDB
```

;

See also MOVIZ RaH, \#16FHiHex
MOVXI RaH, \#16FLoHex

## MOVSTO FLAG Load Selected STF Flags into STO

## Operands

FLAG Selected flag

| Opcode | LSW: 10101101 FFFF fFFF |
| :--- | :--- |
| Description | Load selected flags from the STF register into the STO register of the $28 \times$ CPU where |
|  | FLAG is one or more of TF, CI, ZI, ZF, NI, NF, LUF or LVF. The specified flag maps to |
| the STO register as follows: |  |

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 (STO). If a decision needs to be made based on a floating-point operation, the information in the STF register needs to be loaded into STO flags (Z,N,OV,TC,C) so that the appropriate branch conditional instruction can be executed. The MOVSTO FLAG instruction is used to load the current value of specified STF flags into the respective bits of STO. When this instruction executes, it will also clear the latched overflow and underflow flags if those flags are specified. |
|  | ```Loop: MOV32 ROH,*XAR4++ MOV32 R1H,*XAR3++ CMPF32 R1H, ROH MOVSTO ZF, NF BF Loop, GT ; Loop if (R1H > ROH)``` |
| 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 $(\mathrm{ROH}$ to R 7 H$)$ |
| :--- | :--- |
| \#16FLoHex | A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value. The <br> 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.
$\operatorname{RaH}[15: 0]=$ \#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 $=3.141593$ ( $0 \times 40490$ FDB) MOVIZ ROH, \#0x4049; ROH $=0 \times 40490000$ MOVXI ROH, \#Ox0FDB ; ROH $=0 x 40490 F D B$

See also
MOVIZ RaH, \#16FHiHex
MOVIZF32 RaH, \#16FHi

## MPYF32 RaH, RbH, RcH 32-bit Floating-Point Multiply

## Operands

| RaH | floating-point destination register (ROH 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.
$\mathrm{RaH}=\mathrm{RbH} * \mathrm{RcH}$
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

Example

## See also

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.

Calculate $\mathrm{Y}=\mathrm{A}$ * 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
MOV32 *XAR4,R0H ; Save the result
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 (ROH to R7H) |
| :--- | :--- |
| \#16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The <br> low 16-bits of the mantissa are assumed to be all 0. |
| RcH | floating-point source register (R0H to R7H) |

## Opcode

## Description

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 ( $2 p$ ) 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 R 7 H$)$ |
| :--- | :--- |
| RbH | floating-point source register (ROH 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

## Description

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 ( $2 p$ ) instruction. That is:
MPYF32 RaH, RbH, \#16FHi
NOP

NOP | $\quad ; 2$ pipeline cycles (2p) |
| :--- |
|  |

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 $=2.0(0 \times 40000000)$ |
| :--- | :--- | :--- |
| MPYF32 | R4H, R3H, \#3.0 | $;$ R4H $=$ R3H * 3.0 |
| MOVL | XAR1, \#0xB008 | $;<--$ Non conflicting instruction |
|  |  | $;<--$ MPYF32 complete, R4H $=6.0$ ( $0 \times 40 \mathrm{C} 00000)$ |
| MOV32 | *XAR1, R4H | $;$ Save the result in memory location 0xB008 |

## Example 2



## 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 (ROH to R7H) <br> RaH cannot be the same register as RdH |
| :--- | :--- |
| RbH | floating-point source register for MPYF32 (R0H to R7H) <br> RcH |
| RdH | floating-point source register for MPYF32 (R0H to R7H) <br> floating-point destination register for ADDF32 (R0H to R7H) <br> RdH cannot be the same register as RaH |
| ReH | floating-point source register for ADDF32 (R0H to R7H) <br> RfH |

Opcode | LSW: 1110 0111 010000 ff |
| :--- |
| MSW: feee dddc ccbb baaa |

Description Multiply the contents of two floating-point registers with parallel addition of two registers.
$\mathrm{RaH}=\mathrm{RbH} * \mathrm{RcH}$
RdH $=$ ReH $+R f H$
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.

```
MPYF32 RaH, RbH, RcH |ADDF32 RdH, ReH, RfH - 32-bit Floating-Point Multiply with Parallel Add
```


## Example

```
; Perform 5 multiply and accumulate operations:
```

; Perform 5 multiply and accumulate operations:
;
; 1st multiply: A = X0 * Y0
; 1st multiply: A = X0 * Y0
; 2nd multiply: B = X1 * Y1
; 2nd multiply: B = X1 * Y1
; 3rd multiply: C = X2 * Y2
; 3rd multiply: C = X2 * Y2
; 4th multiply: D = X3 * Y3
; 4th multiply: D = X3 * Y3
; 5th multiply: E = X3 * Y3
; 5th multiply: E = X3 * Y3
;
; Result = A + B + C + D + E
; Result = A + B + C + D + E
MOV32 R0H, *XAR4++ ; R0H = X0
MOV32 R1H, *XAR5++ ; R1H = Y0
MPYF32 R2H, R0H, R1H ; In parallel R0H = X1
|| MOV32 R0H, *XAR4++
MOV32 R1H, *XAR5++ ; R1H = 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
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
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

```

Instruments
www.ti.com
MPYF32 RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Multiply with Parallel Move

\section*{MPYF32 RdH, ReH, RfH ||MOV32 RaH, mem32 32-bit Floating-Point Multiply with Parallel Move}

\section*{Operands}
\begin{tabular}{ll}
\hline RdH & \begin{tabular}{l} 
floating-point destination register for the MPYF32 (R0H to R7H) \\
RdH cannot be the same register as RaH
\end{tabular} \\
ReH & \begin{tabular}{l} 
floating-point source register for the MPYF32 (R0H to R7H) \\
RfH \\
floating-point source register for the MPYF32 (R0H to R7H) \\
floating-point destination register for the MOV32 (R0H to R7H) \\
mem32
\end{tabular} \\
\hline
\end{tabular}

Opcode LSW: 111000110000 fffe
MSW: eedd daaa mem32
Description Multiply the contents of two floating-point registers and load another.
RdH \(=\) ReH * Rff
\(\mathrm{RaH}=\) [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:.
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & Yes & Yes & Yes & Yes & Yes & Yes \\
\hline
\end{tabular}

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;

```

\section*{Pipeline}

MPYF32 takes 2 pipeline-cycles ( \(2 p\) ) 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.

\section*{Example}

Calculate \(Y=M 1 * X 1+B 1\). This example assumes that \(M 1, X 1, B 1\) and \(Y 1\) 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
Add M*X1 to B1 and store in R1H
; Wait 1 cycle for ADDF32 to complete
; <-- ADDF32 complete
; Store the result

```
Calculate \(\mathrm{Y}=(\mathrm{A} * \mathrm{~B}){ }^{*} \mathrm{C}\) :
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++

\section*{MPYF32 RdH, ReH, RfH ||MOV32 mem32, RaH 32-bit Floating-Point Multiply with Parallel Move}

\section*{Operands}
\begin{tabular}{ll}
\hline RdH & floating-point destination register for the MPYF32 (ROH to R7H) \\
ReH & floating-point source register for the MPYF32 (ROH 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 (ROH to R7H)
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 111000000000 fffe \\
MSW: eedd daaa mem32
\end{tabular}

Description Multiply the contents of two floating-point registers and move from memory to register. RdH \(=\) ReH * Rfy, [mem32] = RaH

Flags
This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

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.

\section*{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)

```

\section*{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++

\section*{MPYF32 RaH, RbH, RcH ||SUBF32 RdH, ReH, RfH 32-bit Floating-Point Multiply with Parallel Subtract}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & \begin{tabular}{l} 
floating-point destination register for MPYF32 (ROH to R7H) \\
RaH cannot be the same register as RdH \\
floating-point source register for MPYF32 (ROH to R7H)
\end{tabular} \\
RbH & \begin{tabular}{l} 
floating-point source register for MPYF32 (ROH to R7H) \\
RcH \\
RdH
\end{tabular} \\
\begin{tabular}{l} 
RdH cannot be the same register as RaH
\end{tabular} \\
ReH & \begin{tabular}{l} 
Rloating-point source register for SUBF32 (R0H to R7H) \\
flo \\
floating-point source register for SUBF32 (ROH to R7H)
\end{tabular} \\
\hline
\end{tabular}

Opcode LSW: \(11100111010100 f f\) MSW: feee dddc ccbb baaa
Description Multiply the contents of two floating-point registers with parallel subtraction of two registers.
\(\mathrm{RaH}=\mathrm{RbH} * \mathrm{RcH}\),
RdH \(=\operatorname{ReH}-\operatorname{RfH}\)
Restrictions The destination register for the MPYF32 and the SUBF32 must be unique. That is, RaH cannot be the same register as RdH.

Flags \(\quad\) This instruction modifies the following flags in the STF register:.
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

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.

\section*{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.

\section*{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 (0\times40000000)

```

\author{
See also \\ SUBF32 RaH, RbH, RcH \\ SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32 \\ SUBF32 RdH, ReH, RfH || MOV32 mem32, RaH
}

\section*{NEGF32 RaH, RbH\{, CNDF\} Conditional Negation}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & floating-point destination register (ROH to R7H) \\
RbH & floating-point source register (ROH to R7H) \\
CNDF & condition tested \\
\hline
\end{tabular}

Opcode
```

LSW: 1110 0110 1010 CNDF

```
MSW: 00000000 00bb baaa

Description
if (CNDF == true) \(\{\mathrm{RaH}=-\mathrm{RbH}\}\)
else \(\{\mathrm{RaH}=\mathrm{RbH}\}\)
CNDF is one of the following conditions:
\begin{tabular}{|c|c|c|c|}
\hline Encode \({ }^{(1)}\) & CNDF & Description & STF Flags Tested \\
\hline 0000 & NEQ & Not equal to zero & ZF == 0 \\
\hline 0001 & EQ & Equal to zero & \(Z F==1\) \\
\hline 0010 & GT & Greater than zero & \(\mathrm{ZF}==0 \mathrm{AND}\) NF \(==0\) \\
\hline 0011 & GEQ & Greater than or equal to zero & \(N F=0\) \\
\hline 0100 & LT & Less than zero & \(N F=1\) \\
\hline 0101 & LEQ & Less than or equal to zero & \(\mathrm{ZF}==1\) AND NF \(==1\) \\
\hline 1010 & TF & Test flag set & \(\mathrm{TF}==1\) \\
\hline 1011 & NTF & Test flag not set & TF \(=0\) \\
\hline 1100 & LU & Latched underflow & LUF \(=1\) \\
\hline 1101 & LV & Latched overflow & LVF = = 1 \\
\hline 1110 & UNC & Unconditional & None \\
\hline 1111 & UNCF \({ }^{(2)}\) & Unconditional with flag modification & None \\
\hline
\end{tabular}
(1) Values not shown are reserved.
\({ }^{(2)}\) This is the default operation if no CNDF field is specified. This condition will allow the \(\mathrm{ZF}, \mathrm{NF}, \mathrm{ZI}\), and NI flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags \(\quad\) This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & Yes & Yes & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

Example

See also

This is a single-cycle instruction.
```

MOVIZF32 R0H, \#5.0 ; ROH = 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

```

ABSF32 RaH, RbH

\section*{POP RB}

\section*{Pop the RB Register from the Stack}

\section*{Operands}
RB repeat block register

\section*{Opcode}

\section*{Description}

Flags
This instruction does not affect any flags floating-point Unit:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

Example

This is a single-cycle instruction.
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

```
\begin{tabular}{ll} 
See also & PUSH RB \\
& RPTB \#RSIZE, RC \\
& RPTB \#RSIZE, loc16
\end{tabular}

\section*{PUSH RB}

\section*{Push the RB Register onto the Stack}

\section*{Operands}
RB repeat block register
Opcode LSW: 1111111111110000

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:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}
```

Pipeline
Example

```

\section*{RESTORE}

\section*{Restore the Floating-Point Registers}

Operands
none \(\quad\) This instruction does not have any operands

\section*{Opcode}

\section*{Description}

Restrictions

Flags
Restoring the status register will overwrite all flags:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & Yes & Yes & Yes & Yes & Yes & Yes & Yes \\
\hline
\end{tabular}

\footnotetext{
Pipeline \(\quad\) This is a single-cycle instruction.
}

\section*{Example}

See also

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.

\section*{RPTB label, loc16 Repeat A Block of Code}

\section*{Operands}
\begin{tabular}{ll}
\hline 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.
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 10110101 0bbb bbbb \\
MSW: 00000000 loc16
\end{tabular}

Description Initialize repeat block loop, repeat count from [loc16]

\section*{Restrictions}

Flags This instruction does not affect any flags in the floating-point unit:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.
Example \(\quad\) The minimum size for the repeat block is 9 words if the block is even-aligned and 8 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 evenaligned. 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,XARO
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

```

\section*{See also}

POP RB
PUSH RB
RPTB label, RC

\section*{RPTB label, \#RC}

\section*{Repeat a Block of Code}

\section*{Operands}
\begin{tabular}{ll}
\hline 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 location
\end{tabular}

Opcode LSW: 10110101 1bbb bbbb
MSW: cccc cccc cccc cccc

\section*{Description}

Repeat a block of code. The repeat count is specified as a immediate value.

\section*{Restrictions}

Flags This instruction does not affect any flags int the floating-point unit:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

Example \(\quad\) 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,XARO
MOV32 R1H,*XAR0++ ; 8 or 9 words block size 127 words
MAXF32 R0H,R1H
MOVSTO 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

INSTRUMENTS

\section*{SAVE FLAG, VALUE Save Register Set to Shadow Registers and Execute SETFLG}

\section*{Operands}
\begin{tabular}{ll}
\hline FLAG & 11 bit mask indicating which floating-point status flags to change. \\
VALUE & 11 bit mask indicating the flag value; 0 or 1. \\
\hline
\end{tabular}

\section*{Opcode LSW: 11100110 01FF FFFF}

MSW: FFFF FVVV VVVV VVVV
Description This operation copies the current working floating-point register set (ROH 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 \(\quad\) This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & Yes & Yes & Yes & Yes & Yes & Yes & Yes \\
\hline
\end{tabular}

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, ARO, AR1 and PC.

SAVE FLAG, VALUE - Save Register Set to Shadow Registers and Execute SETFLG
```

_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 PAGEO,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

InsTRUMENTS

\section*{SETFLG FLAG, VALUE Set or clear selected floating-point status flags}

\section*{Operands}
\begin{tabular}{ll}
\hline FLAG & 11 bit mask indicating which floating-point status flags to change. \\
VALUE & 11 bit mask indicating the flag value; 0 or 1. \\
\hline
\end{tabular}

Opcode LSW: 11100110 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:
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline reserved & RNDF32 & reserved & reserved & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline
\end{tabular}

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 \(\quad\) This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & Yes & Yes & Yes & Yes & Yes & Yes & Yes \\
\hline
\end{tabular}

Any flag can be modified by this instruction.

\section*{Pipeline}

\section*{Example}

See also

This is a single-cycle instruction.
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

```

SAVE FLAG, VALUE

\section*{SUBF32 RaH, RbH, RcH 32-bit Floating-Point Subtraction}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & floating-point destination register (ROH to R1) \\
RbH & floating-point source register (R0H to R1) \\
RcH & floating-point source register (R0H to R1) \\
\hline
\end{tabular}
\begin{tabular}{llllll} 
Opcode & LSW: 1110 & 0111 & 0010 & 0000 \\
MSW: 0000 & \(000 c\) & ccbb & baaa
\end{tabular}

\section*{Description Subtract the contents of two floating-point registers}
\(\mathrm{RaH}=\mathrm{RbH}-\mathrm{RcH}\)
Flags \(\quad\) This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

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.
\begin{tabular}{|c|c|c|}
\hline \multirow[t]{13}{*}{Example} & \multicolumn{2}{|l|}{Calculate \(Y-A+B-C\) :} \\
\hline & \multicolumn{2}{|l|}{MOVL XAR4, \#A} \\
\hline & MOV32 R0H, *XAR4 & ; Load R0H with A \\
\hline & MOVL XAR4, \#B & \\
\hline & MOV32 R1H, *XAR4 & ; Load R1H with B \\
\hline & MOVL XAR4, \#C & \\
\hline & ADDF32 R0H,R1H, R0H & ; Add A + B and in parallel \\
\hline & || MOV32 R2H,*XAR4 & ; Load R2H with C \\
\hline & & ; <-- ADDF32 complete \\
\hline & SUBF32 R0H,R0H, R2H & ; Subtract C from ( \(\mathrm{A}+\mathrm{B}\) ) \\
\hline & NOP & \\
\hline & & ; <-- SUBF32 completes \\
\hline & MOV32 *XAR4,R0H & Store the result \\
\hline
\end{tabular}

\section*{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

\section*{SUBF32 RaH, \#16FHi, RbH 32-bit Floating Point Subtraction}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & floating-point destination register (ROH to R1) \\
\#16FHi & \begin{tabular}{l} 
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.
\end{tabular} \\
RbH & floating-point source register (R0H to R1)
\end{tabular}

\section*{Opcode}

\section*{Description}

Flags
```

LSW: 1110 1000 11II IIII
MSW: IIII IIII IIbb baaa

```

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 \(0 . \# 16 \mathrm{FHi}\) is most useful for representing constants where the lowest 16-bits of the mantissa are 0. Some examples are 2.0 ( \(0 \times 40000000\) ), 4.0 ( \(0 \times 40800000\) ), 0.5 ( \(0 \times 3 F 000000\) ), 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
This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

The STF register flags are modified as follows:
- LUF = 1 if MPYF32 generates an underflow condition.
- LVF = 1 if MPYF32 generates an overflow condition.

\section*{Pipeline}

Example

\section*{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

Texas
Instruments

SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Subtraction with Parallel Move
www.ti.com

\section*{SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32 32-bit Floating-Point Subtraction with Parallel Move}

Operands
\begin{tabular}{ll}
\hline RdH & \begin{tabular}{l} 
floating-point destination register \((R 0 H\) to \(R 7 H)\) for the SUBF32 operation \\
RdH cannot be the same register as \(R a H\)
\end{tabular} \\
ReH & \begin{tabular}{l} 
floating-point source register \((R 0 H\) to \(R 7 H)\) for the SUBF32 operation
\end{tabular} \\
RfH & \begin{tabular}{l} 
floating-point source register \((R 0 H\) to \(R 7 H)\) for the SUBF32 operation \\
floating-point destination register \((R 0 H\) to \(R 7 H)\) for the MOV32 operation \\
RaH cannot be the same register as \(R d H\)
\end{tabular} \\
mem32 & \begin{tabular}{l} 
pointer to 32-bit source memory location for the MOV32 operation
\end{tabular} \\
\hline
\end{tabular}

Opcode LSW: 111000110010 fffe
MSW: eedd daaa mem32
Description Subtract the contents of two floating-point registers and move from memory to a floatingpoint register.
RdH \(=\) ReH - Rft, \(R a H=\) [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 \(\quad\) This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & Yes & Yes & Yes & Yes & Yes & Yes \\
\hline
\end{tabular}

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 \(\quad\) 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.
www.ti.com SUBF32 RdH, ReH, RfH ||MOV32 RaH, mem32 - 32-bit Floating-Point Subtraction with Parallel Move

\section*{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 ;
; <-- ROH 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

\section*{SUBF32 RdH, ReH, RfH ||MOV32 mem32, RaH 32-bit Floating-Point Subtraction with Parallel Move}

\section*{Operands}
\begin{tabular}{ll}
\hline RdH & floating-point destination register (ROH to R7H) for the SUBF32 operation \\
ReH & floating-point source register \((\) ROH to R7H \()\) for the SUBF32 operation \\
RfH & floating-point source register (ROH to R7H) for the SUBF32 operation \\
mem32 & \begin{tabular}{l} 
pointer to 32-bit destination memory location for the MOV32 operation \\
faH
\end{tabular} \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 111000000010 fffe \\
MSW: eedd daaa mem32
\end{tabular}

Description Subtract the contents of two floating-point registers and move from a floating-point register to memory.
\(\mathrm{RdH}=\mathrm{ReH}-\mathrm{RfH}\),
[mem32] \(=\mathrm{RaH}\)
Flags
This instruction modifies the following flags in the STF register: SUBF32 RdH, ReH, RfH || MOV32 RaH, mem32
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

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
; 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.

\section*{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
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
; R4H = D = R7H + R1H and store R6H (B)
ADDF32 R4H, R7H, R1H
; <-- SUBF32 (C) completes, R3H valid
; <-- MOV32 completes, (B) stored
; store R3H (C)
; <-- MOV32 completes, (C) stored
; <-- ADDF32 (D) completes, R4H valid
; store R4H (D)
; <-- MOV32 completes, (D) stored

```
www.ti.com SUBF32 RdH, ReH, RfH ||MOV32 mem32, RaH - 32-bit Floating-Point Subtraction with Parallel Move
\begin{tabular}{ll} 
See also & SUBF32 RaH, RbH, RcH \\
& SUBF32 RaH, \#16FHi, RbH \\
& SUBF32 RdH, ReH, RHH || MOV32 RaH, mem32 \\
& MPYF32 RaH, RbH, RcH || SUBF32 RdH, ReH, RfH
\end{tabular}

\section*{SWAPF RaH, RbH\{, CNDF\} Conditional Swap}

\section*{Operands}
\begin{tabular}{ll}
RaH & floating-point register (ROH to R7H) \\
RbH & floating-point register (ROH to R7H) \\
CNDF & condition tested
\end{tabular}

Opcode
```

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

```

\section*{Description Conditional swap of RaH and RbH .}
if (CNDF == true) swap RaH and RbH
CNDF is one of the following conditions:
\begin{tabular}{llll}
\hline Encode \({ }^{(1)}\) & CNDF & Description & STF Flags Tested \\
\hline 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 \\
\hline
\end{tabular}
\({ }^{(1)}\) 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:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

No flags affected
Pipeline
This is a single-cycle instruction.

\section*{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:

```

\section*{TESTTF CNDF}

\section*{Test STF Register Flag Condition}

\section*{Operands}
CNDF condition to test

\section*{Opcode}

\section*{Description}

LSW: 111001011000 CNDF
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) \(\mathrm{TF}=1\); else \(\mathrm{TF}=0\);
CNDF is one of the following conditions:
\begin{tabular}{llll}
\hline Encode \(^{(1)}\) & CNDF & Description & STF Flags Tested \\
\hline 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 \\
\hline
\end{tabular}
(1) Values not shown are reserved.
\({ }^{(2)}\) This is the default operation if no CNDF field is specified. This condition will allow the \(\mathrm{ZF}, \mathrm{NF}, \mathrm{ZI}\), 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:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & Yes & No & No & No & No & No & No \\
\hline
\end{tabular}
\(\mathrm{TF}=0\); if (CNDF == true) \(\mathrm{TF}=1\);
Note: If (CNDF == UNC or UNCF), the TF flag will be set to 1 .
Pipeline
This is a single-cycle instruction.
Example
```

CMPF32 R0H, \#O.0 ; Compare R0H against 0
TESTTF LT ; Set TF if ROH less than 0 (NF == 0)
ABS ROH, ROH ; Get the absolute value of ROH
; Perform calculations based on ABS ROH
MOVST0 TF ; Copy TF to TC in ST0
SBF End, NTC ; Branch to end if TF was not set
NEGF32 ROH, ROH
End

```

\section*{See also}

\section*{Ul16TOF32 RaH, mem16 Convert unsigned 16-bit integer to 32-bit floating-point value}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & floating-point destination register (ROH to R7H) \\
mem16 & pointer to 16-bit source memory location \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110001011000100 \\
& MSW: 0000 0aaa mem16 \\
Description & RaH \(=\) UI16ToF32[mem16]
\end{tabular}

\section*{Flags}

This instruction does not affect any flags:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

\section*{Example}

See also
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.
```

float32 y,m,b;
AdcRegs.RESULTO 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, ROH valid
MPYF32 R0H, R1H, R0H ; R0H = (float)X * M
MOV32 R1H, *-SP[8] ; R1H = B
; <-- MPYF32 complete, ROH valid
ADDF32 R0H, R0H, R1H ; R0H = Y = (float)X * M + B
NOP
MOV32 *-[SP], R0H ; Store Y

```

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
Ul16TOF32 RaH, RbH

Instruments

\section*{Ul16TOF32 RaH, RbH Convert unsigned 16-bit integer to 32-bit floating-point value}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & \begin{tabular}{l} 
floating-point destination register (ROH to R 7 H\()\) \\
floating-point source register (ROH to R7H)
\end{tabular} \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011010001111 \\
& MSW: 0000000000 bb baaa \\
Description & RaH \(=\) UI16ToF32[RbH]
\end{tabular}

Flags \(\quad\) This instruction does not affect any flags:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

Example

See also
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.
```

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)

```

F32TOI16 RaH, RbH
F32TOI16R RaH, RbH
F32TOUI16 RaH, RbH
F32TOUI16R RaH, RbH
I16TOF32 RaH, RbH
I16TOF32 RaH, mem16
Ul16TOF32 RaH, mem16

\section*{Ul32TOF32 RaH, mem32 Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & floating-point destination register (ROH to R7H) \\
mem32 & pointer to 32-bit source memory location \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110001010000100 \\
& MSW: 0000 0aaa mem32 \\
Description & RaH \(=\) UI32ToF32[mem32]
\end{tabular}

Flags
This instruction does not affect any flags:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

\section*{Example}

\section*{See also}

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

\section*{UI32TOF32 RaH, RbH Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value}

\section*{Operands}
\begin{tabular}{ll}
\hline RaH & \begin{tabular}{l} 
floating-point destination register (ROH to R 7 H\()\) \\
floating-point source register (ROH to R7H)
\end{tabular} \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011010001011 \\
& MSW: 0000000000 bb baaa \\
Description & RaH \(=\) UI32ToF32[RbH]
\end{tabular}

Flags \(\quad\) This instruction does not affect any flags:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

Example

See also
This is a 2 pipeline cycle ( \(2 p\) ) 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.
```

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)

```

F32TOI32 RaH, RbH
F32TOUI32 RaH, RbH
I32TOF32 RaH, mem32
I32TOF32 RaH, RbH
Ul32TOF32 RaH, mem32

\section*{ZERO RaH}

\section*{Zero the Floating-Point Register RaH}

Operands
RaH floating-point register (R0H to R7H)
Opcode LSW: 111001011001 Oaaa

\section*{Description Zero the indicated floating-point register:}

RaH \(=0\)
Flags
This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

Pipeline

\section*{Example}

See also

This is a single-cycle instruction.
```

;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 ARO, ACC
MOV32 R0H, *+XAR4[AR0] ; R0H = x[2*i]
MOV32 R1H, *+XAR5[AR0] ; R1H = y[2*i]
ADD ARO, \#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--

```
ZEROA

\section*{ZEROA}

\section*{Zero All Floating-Point Registers}

\section*{Operands}

\section*{Opcode}

\section*{Description}

Flags

LSW: 1110010101100011
Zero all floating-point registers:
ROH \(=0\)
\(\mathrm{R} 1 \mathrm{H}=0\)
R2H \(=0\)
R3H \(=0\)
\(\mathrm{R} 4 \mathrm{H}=0\)
R5H \(=0\)
R6H \(=0\)
R7H \(=0\)

This instruction modifies the following flags in the STF register:
\begin{tabular}{llllllll}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}

\section*{Pipeline}

See also

\section*{Example}

No flags affected.

This is a single-cycle instruction.
```

;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
ZEROA ; Clear all RaH registers
LOOP
MOV AL, AR7
MOV ACC, AL << 2
MOV ARO, ACC
MOV32 R0H, *+XAR4[ARO] ; 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--

```

\section*{C28 Viterbi, Complex Math and CRC Unit-II (VCU-II)}

This chapter provides an overview of the architectural structure and instruction set of the Viterbi, Complex Math and CRC Unit (VCU-II) and describes the architecture, pipeline, instruction set, and interrupts. The VCU is a fully-programmable block which accelerates the performance of communications-based algorithms. In addition to eliminating the need for a second processor to manage the communications link, the performance gains of the VCU provides headroom for future system growth and higher bit rates or, conversely, enables devices to operate at a lower MHz to reduce system cost and power consumption.
Any references to VCU or VCU-II in this chapter relate to Type 2 specifically. Information pertaining to an older VCU will have the module type listed explicitly. See the TMS320x28xx, 28xxx DSP Peripheral Reference Guide (SPRU566) for a list of all devices with a VCU module of the same type, to determine the differences between the types, and for a list of device-specific differences within a type.
Topic Page
2.1 Overview ..... 141
2.2 Components of the C28x Plus VCU ..... 142
2.3 Register Set ..... 146
2.4 Pipeline ..... 154
2.5 Instruction Set ..... 159
2.6 Rounding Mode ..... 379

Texas
InsTruments

\subsection*{2.1 Overview}

The C28x with VCU (C28x+VCU) processor extends the capabilities of the C28x fixed-point or floatingpoint CPU by adding registers and instructions to support the following algorithm types:
- Viterbi decoding

Viterbi decoding is commonly used in baseband communications applications. The viterbi decode algorithm consists of three main parts: branch metric calculations, compare-select (viterbi butterfly) and a traceback operation. Table 2-1 shows a summary of the VCU performance for each of these operations.

Table 2-1. Viterbi Decode Performance
\begin{tabular}{|c|c|}
\hline Viterbi Operation & VCU Cycles \\
\hline Branch Metric Calculation (code rate \(=1 / 2\) ) & 1 \\
\hline Branch Metric Calculation (code rate \(=1 / 3\) ) & 2 p \\
\hline Viterbi Butterfly (add-compare-select) & \(2^{(1)}\) \\
\hline Traceback per Stage & \(3^{(2)}\) \\
\hline
\end{tabular}
\({ }^{(1)} \mathrm{C} 28 \mathrm{x}\) CPU takes 15 cycles per butterfly.
(2) C 28 x CPU takes 22 cycles per stage.
- Cyclic redundancy check (CRC)

CRC algorithms provide a straightforward method for verifying data integrity over large data blocks, communication packets, or code sections. The C28x+VCU can perform 8-, 16-, 24-, and 32-bit CRCs. For example, the VCU can compute the CRC for a block length of 10 bytes in 10 cycles. A CRC result register contains the current CRC which is updated whenever a CRC instruction is executed.
- Complex math

Complex math is used in many applications. The VCU A few of which are:
- Fast fourier transform (FFT)

The complex FFT is used in spread spectrum communications, as well in many signal processing algorithms.
- Complex filters

Complex filters improve data reliability, transmission distance, and power efficiency. The \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) can perform a complex I and Q multiply with coefficients (four multiplies) in a single cycle. In addition, the \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) can read/write the real and imaginary parts of 16 -bit complex data to memory in a single cycle.
Table 2-2 shows a summary of the VCU operations enabled by the VCU:
Table 2-2. Complex Math Performance
\begin{tabular}{|c|c|l|}
\hline Complex Math Operation & \multicolumn{1}{|c|}{ VCU Cycles } & \multicolumn{1}{|c|}{ Notes } \\
\hline Add Or Subtract & 1 & \(32+/-32=32\)-bit (Useful for filters) \\
\hline Add or Subtract & 1 & \(16+/-32=15\)-bit (Useful for FFT) \\
\hline Multiply & \(2 p\) & \(16 \times 16=32\)-bit \\
\hline Multiply \& Accumulate (MAC) & \(2 p\) & \(32+32=32\)-bit, \(16 \times 16=32\)-bit \\
\hline RPT MAC & \(2 p+N\) & Repeat MAC. Single cycle after the first operation. \\
\hline
\end{tabular}

This \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) draws from the best features of digital signal processing; reduced instruction set computing (RISC); and microcontroller architectures, firmware, and tool sets. The C2000 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.
- C28x plus VCU and C28x+VCU both refer to the C28x CPU with enhancements to support viterbi decode, complex math, forward error correcting algorithms, and CRC.
- Some devices have both the FPU and the VCU. These are referred to as C28x+FPU+VCU.

\subsection*{2.2 Components of the C28x Plus VCU}

The VCU extends the capabilities of the C28x CPU and C28x+FPU processors by adding additional instructions. No changes have been made to existing instructions, pipeline, or memory bus architecture. Therefore, programs written for the C28x are completely compatible with the C28x+VCU. All of the features of the C28x documented in TMS320C28x DSP CPU and Instruction Set Reference Guide (literature number SPRU430) apply to the C28x+VCU. All features documented in the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUE02) apply to the C28x+FPU+VCU.

Figure 2-1 shows the block diagram of the VCU.
Figure 2-1. C28x + VCU Block Diagram


The C28x+VCU contains the same features as the C28x fixed-point CPU:
- 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.
- 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.
- Arithmetic logic unit (ALU). The 32-bit ALU performs 2s-complement arithmetic and Boolean logic operations.
- Address register arithmetic unit (ARAU). The ARAU generates data memory addresses and increments or decrements pointers in parallel with ALU operations.
- 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.
- 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 2 s -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.
The VCU adds the following features:
- Instructions to support Cyclic Redundancy Check (CRC) or a polynomial code checksum
- CRC8
- CRC16
- CRC32
- CRC24
- Clocked at the same rate as the main CPU (SYSCLKOUT).
- Instructions to support a software implementation of a Viterbi Decoder of constraint length 4-7 and code rates of \(1 / 2\) and \(1 / 3\)
- Branch metrics calculations
- Add-Compare Select or Viterbi Butterfly
- Traceback
- Complex Math Arithmetic Unit
- Add or Subtract
- Multiply
- Multiply and Accumulate (MAC)
- Repeat MAC (RPT || MAC).
- Independent register space. These registers function as source and destination registers for VCU instructions.
- Some VCU instructions require pipeline alignment. This alignment is done through software to allow the user to improve performance by taking advantage of required delay slots. See Section 2.4 for more information.
Devices with the floating-point unit also include:
- Floating point unit (FPU). The 32-bit FPU performs IEEE single-precision floating-point operations.
- Dedicated floating-point registers.

\subsection*{2.2.1 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 ESTOPO or ESTOP1 instruction.
- An access to a specified program-space or data-space location. When a debug event causes the C 28 x to enter the debug-halt state, the event is called a break event.
- Real-time mode of operation.

\subsection*{2.2.2 Memory Map}

Like the C28x, the C28x+VCU uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4 G words ( 1 word \(=16\) bits) in data space and 4 M words in program space. Memory blocks on all \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) designs are uniformly mapped to both program and data space. For specific details about each of the map segments, see the device-specific data manual.

\subsection*{2.2.3 CPU Interrupt Vectors}

The C28x+VCU 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. For more information about the CPU vectors, see TMS320C28x CPU and Instruction Set Reference Guide (literature number SPRU430). Typically the CPU interrupt vectors are only used during the boot up of the device by the boot ROM. Once an application has taken control it should initialize and enable the peripheral interrupt expansion block (PIE).

\subsection*{2.2.4 Memory Interface}

The C28x+VCU memory interface is identical to that on the C28x. The C28x+VCU 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 CPU 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.

\subsection*{2.2.5 Address and Data Buses}

Like the C28x, the memory interface has three address buses:
- PAB: Program address bus: The 22-bit PAB carries addresses for reads and writes from program space.
- 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 32-bit PRDB carries instructions during reads from program space.
- DRDB: Data-read data bus: The 32-bit DRDB carries data during reads from data space.
- 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.

\subsection*{2.2.6 Alignment of 32-Bit Accesses to Even Addresses}

The C28x+VPU 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.

\subsection*{2.3 Register Set}

Devices with the C28x+VCU include the standard C28x register set plus an additional set of VCU specific registers. The additional VCU registers are the following:
- Result registers: VR0, VR1... VR8
- Traceback registers: VT0, VT1
- Configuration and status register: VSTATUS
- CRC result register: VCRC
- Repeat block register: RB

Figure 2-2 shows the register sets for the 28x CPU, the FPU and the VCU. The following section discusses the VCU register set in detail.

Figure 2-2. C28x + FPU + VCU Registers
\begin{tabular}{|c|}
\hline Standard C28x Register Set \\
\hline ACC (32-bit) \\
\hline P (32-bit) \\
\hline XT (32-bit) \\
\hline XAR0 (32-bit) \\
\hline XAR1 (32-bit) \\
\hline XAR2 (32-bit) \\
\hline XAR3 (32-bit) \\
\hline XAR4 (32-bit) \\
\hline XAR5 (32-bit) \\
\hline XAR6 (32-bit) \\
\hline XAR7 (32-bit) \\
\hline PC (22-bit) \\
\hline RPC (22-bit) \\
\hline DP (16-bit) \\
\hline SP (16-bit) \\
\hline ST0 (16-bit) \\
\hline ST1 (16-bit) \\
\hline IER (16-bit) \\
\hline IFR (16-bit) \\
\hline DBGIER (16-bit) \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Additional 32-bit FPU Registers & Standard VCU Register Set \\
\hline ROH (32-bit) & VR0 \\
\hline & VR1 \\
\hline R1H (32-bit) & VR2 \\
\hline R2H (32-bit) & VR3 \\
\hline & VR4 \\
\hline R3H (32-bit) & VR5 \\
\hline & VR6 \\
\hline & VR7 \\
\hline R5H (32-bit) & VR8 \\
\hline R6H (32-bit) & VT0 \\
\hline & VT1 \\
\hline R7H (32-bit) & VSTATUS \\
\hline & VCRC \\
\hline FPU Status Register (STF) & VSMO \\
\hline Repeat Block Register (RB) & VSM1 \\
\hline \multirow[t]{2}{*}{FPU registers ROH - R7H and STF are shadowed for fast context save and restore} & \\
\hline & VSM63 \\
\hline
\end{tabular}

\subsection*{2.3.1 VCU Register Set}

Table 2-3 describes the VCU module register set. The last three columns indicate whether the particular module within the VCU can make use of the register.

Table 2-3. VCU Register Set
\begin{tabular}{|c|c|c|c|c|c|}
\hline Register Name & Size & Description & Viterbi & Complex Math & CRC \\
\hline VR0 & 32 bits & General purpose register 0 & Yes & Yes & No \\
\hline VR1 & 32 bits & General purpose register 1 & Yes & Yes & No \\
\hline VR2 & 32 bits & General purpose register 2 & Yes & Yes & No \\
\hline VR3 & 32 bits & General purpose register 3 & Yes & Yes & No \\
\hline VR4 & 32 bits & General purpose register 4 & Yes & Yes & No \\
\hline VR5 & 32 bits & General purpose register 5 & Yes & Yes & No \\
\hline VR6 & 32 bits & General purpose register 6 & Yes & Yes & No \\
\hline VR7 & 32 bits & General purpose register 7 & Yes & Yes & No \\
\hline VR8 & 32 bits & General purpose register 8 & Yes & No & No \\
\hline VT0 & 32 bits & 32-bit transition bit register 0 & Yes & No & No \\
\hline VT1 & 32 bits & 32-bit transition bit register 1 & Yes & No & No \\
\hline VSTATUS & 32 bits & VCU status and configuration register \({ }^{(1)}\) & Yes & Yes & No \\
\hline VCRC & 32 bits & Cyclic redundancy check (CRC) result register & No & No & Yes \\
\hline VSMOVSM63 & 32 bits & Viterbi Decoding State Metric registers & Yes & No & No \\
\hline \[
\begin{aligned}
& \hline \text { VRx.By } \\
& x=0-7 \\
& y=0-3
\end{aligned}
\] & 32 bits & Aliased address space for each byte of the VRx registers, leftshifted by one & No & No & No \\
\hline
\end{tabular}
\({ }^{(1)}\) Debugger writes are not allowed to the VSTATUS register.
Table 2-4 lists the CPU registers available on devices with the C 28 x , the \(\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}\), the \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) and the \(\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}+\mathrm{VCU}\).

\section*{Table 2-4. 28x CPU Register Summary}
\begin{tabular}{|c|c|c|c|c|c|}
\hline Register & C28x CPU & C28x+FPU & C28x+VCU & C28x+FPU+VCU & Description \\
\hline ACC & Yes & Yes & Yes & Yes & Fixed-point accumulator \\
\hline AH & Yes & Yes & Yes & Yes & High half of ACC \\
\hline AL & Yes & Yes & Yes & Yes & Low half of ACC \\
\hline XAR0 - XAR7 & Yes & Yes & Yes & Yes & Auxiliary register 0-7 \\
\hline AR0-AR7 & Yes & Yes & Yes & Yes & Low half of XAR0-XAR7 \\
\hline DP & Yes & Yes & Yes & Yes & Data-page pointer \\
\hline IFR & Yes & Yes & Yes & Yes & Interrupt flag register \\
\hline IER & Yes & Yes & Yes & Yes & Interrupt enable register \\
\hline DBGIER & Yes & Yes & Yes & Yes & Debug interrupt enable register \\
\hline P & Yes & Yes & Yes & Yes & Fixed-point product register \\
\hline PH & Yes & Yes & Yes & Yes & High half of \(P\) \\
\hline PL & Yes & Yes & Yes & Yes & Low half of \(P\) \\
\hline PC & Yes & Yes & Yes & Yes & Program counter \\
\hline RPC & Yes & Yes & Yes & Yes & Return program counter \\
\hline SP & Yes & Yes & Yes & Yes & Stack pointer \\
\hline ST0 & Yes & Yes & Yes & Yes & Status register 0 \\
\hline ST1 & Yes & Yes & Yes & Yes & Status register 1 \\
\hline XT & Yes & Yes & Yes & Yes & Fixed-point multiplicand register \\
\hline T & Yes & Yes & Yes & Yes & High half of XT \\
\hline TL & Yes & Yes & Yes & Yes & Low half of XT \\
\hline ROH - R7H & No & Yes & No & Yes & Floating-point Unit result registers \\
\hline STF & No & Yes & No & Yes & Floating-point Uint status register \\
\hline RB & No & Yes & Yes & Yes & Repeat block register \\
\hline VR0 - VR8 & No & No & Yes & Yes & VCU general purpose registers \\
\hline VT0, VT1 & No & No & Yes & Yes & VCU transition bit register 0 and 1 \\
\hline VSTATUS & No & No & Yes & Yes & VCU status and configuration \\
\hline VCRC & No & No & Yes & Yes & CRC result register \\
\hline VSM0-VSM63 & No & No & Yes \({ }^{(1)}\) & Yes \({ }^{(1)}\) & Viterbi State Metric Registers \\
\hline \[
\begin{array}{|l}
\hline \text { VRx.By } \\
x=0-7 \\
y=0-3
\end{array}
\] & No & No & Yes \({ }^{(1)}\) & Yes \({ }^{(1)}\) & Aliased address space for each byte of the VRx registers, left-shifted by one \\
\hline
\end{tabular}
\({ }^{(1)}\) Present on Type-2 VCU only

\subsection*{2.3.2 VCU Status Register (VSTATUS)}

The VCU status register (VSTATUS) register is described in Figure 2-3. There is no single instruction to directly transfer the VSTATUS register to a C28x register. To transfer the contents:
1. Store VSTATUS into memory using VMOV32 mem32, VSTATUS instruction
2. Load the value from memory into a main C28x CPU register.

Configuration bits within the VSTATUS registers are set or cleared using VCU instructions.
Figure 2-3. VCU Status Register (VSTATUS)


LEGEND: R/W = Read/Write; \(\mathrm{R}=\) Read only; \(-n=\) value after reset

\section*{Table 2-5. VCU Status (VSTATUS) Register Field Descriptions}
\begin{tabular}{|c|c|c|c|}
\hline Bits & Field & Value & Description \\
\hline 31 & \[
\underset{\text { LIP }^{(1)}}{\text { CRGF }}
\] & 0 & \begin{tabular}{l}
CRC Message Flip \\
This bit affects the order in which the bits in the message are taken for CRC calculation by all the CRC instructions. \\
Message bits are taken starting from most-significant to least-significant for CRC computation. In this case, bytes loaded from memory are fed directly for CRC computation. \\
Message bits are taken starting from least-significant to most-significant for CRC computation. In this case, bytes loaded from memory are "flipped" and then fed for CRC computation.
\end{tabular} \\
\hline 30 & DIVE \({ }^{(1)}\) & \[
0
\]
\[
1
\] & \begin{tabular}{l}
Divide-by-zero Error \\
Indicates whether a "divide by zero" occurred during a VMOD32 computation. This bit is cleared by executing the VCLRDIVE instruction
\end{tabular} \\
\hline 29-27 & \(K^{(1)}\) & \[
\begin{gathered}
0 \times 7 \\
1
\end{gathered}
\] & \begin{tabular}{l}
Constraint Length for Viterbi Decoding \\
This field sets the constraint length for the Viterbi decoding algorithm. It accepts values of 4 to 7 . Values outside this range will be treated as 7 by the hardware.
\end{tabular} \\
\hline 26-24 & \[
\underset{1)}{\text { GFORPR}^{( }}
\] & 0x7 & \begin{tabular}{l}
Galois Field Polynomial Order \\
This field holds the Order of the polynomial for all the Galois Field instructions. This field is initialized by the VGFINIT mem16 instruction. The actual order of the polynomial is GFORDER+1
\end{tabular} \\
\hline 23-16 & \(\mathrm{GFPOLY}^{(1)}\) & \[
\begin{aligned}
& 0 \\
& 1
\end{aligned}
\] & \begin{tabular}{l}
Galois Field Polynomial \\
This field holds the Polynomial for all the Galois Field instructions. This field is initialized by the VGFINIT mem16 instruction.
\end{tabular} \\
\hline 15 & OPACK \({ }^{(1)}\) & \[
\begin{aligned}
& 0 \\
& 1
\end{aligned}
\] & \begin{tabular}{l}
Viterbi Traceback Packing Order \\
This bit affects the packing order of the traceback output bits (using the VTRACE instructions) \\
Big-endian (compatible with VCU Type-0 output packing order) \\
Little-endian (VCU Type-2 mode)
\end{tabular} \\
\hline 14 & CPACK \({ }^{(1)}\) & \[
\begin{aligned}
& 0 \\
& 1
\end{aligned}
\] & \begin{tabular}{l}
Complex Packing Order \\
This bit affects the packing order of the 16-bit real and 16-bit imaginary part of a complex numbers inside the 32-bit general purpose VRx register. \\
VRx[31:16] holds Real part, VRx[15:0] holds Imaginary part (VCU-I compatible mode) \\
VRx[31:16] holds Imaginary part; VRx[15:0] holds Real part
\end{tabular} \\
\hline
\end{tabular}

\footnotetext{
\({ }^{(1)}\) Present on Type-2 VCU only.
}

Table 2-5. VCU Status (VSTATUS) Register Field Descriptions (continued)
\begin{tabular}{|c|c|c|c|}
\hline Bits & Field & Value & Description \\
\hline 13 & OVRI & \[
0
\] & \begin{tabular}{l}
Overflow or Underflow Flag: Imaginary Part \\
No overflow or underflow has been detected. \\
Indicates an overflow or underflow has occurred during the computation of the imaginary part of operations shown in Table 10-6. This bit will be set regardless of the value of the VSTATUS[SAT] bit.OVRI bit will remain set until it is cleared by executing the VCLROVFI instruction.
\end{tabular} \\
\hline 12 & OVFR & \[
\begin{aligned}
& 0 \\
& 1
\end{aligned}
\] & \begin{tabular}{l}
Overflow or Underflow Flag: Real Part \\
No overflow or underflow has been detected. \\
Indicates overflow or underflow has occurred during a real number calculation for operations shown in Table 2-6. This bit will be set regardless of the value of the VSTATUS[SAT] bit. This bit will remain set until it is cleared by executing the VCLROVFR instruction.
\end{tabular} \\
\hline 11 & RND & \[
\begin{aligned}
& 0 \\
& 1
\end{aligned}
\] & \begin{tabular}{l}
Rounding \\
When a right-shift operation is performed the lower bits of the value will be lost. The RND bit determines if the shifted value is rounded or if the shifted-out bits are simply truncated. This is described in Section 2.3.2. Operations which use right-shift and rounding are shown in Table 2-6. The RND bit is set by the VRNDON instruction and cleared by the VRNDOFF instruction. \\
Rounding is not performed. Bits shifted out right are truncated. \\
Rounding is performed. Refer to the instruction descriptions for information on how the operation is affected by the RND bit.
\end{tabular} \\
\hline 10 & SAT & \[
0
\] & \begin{tabular}{l}
Saturation \\
This bit determines whether saturation will be performed for operations shown in Table 2-6. The SAT bit is set by the VSATON instruction and is cleared by the VSATOFF instruction. \\
No saturation is performed. \\
Saturation is performed.
\end{tabular} \\
\hline 9-5 & SHIFTL & \[
\begin{gathered}
0 \\
0 \times 01- \\
0 \times 1 F
\end{gathered}
\] & \begin{tabular}{l}
Left Shift \\
Operations which use left-shift are shown in Table 2-6 \\
The shift SHIFTL field can be set or cleared by the VSETSHL instruction. \\
No left shift. \\
Refer to the instruction description for information on how the operation is affected by the shift value. During the left-shift, the lower bits are filled with 0's.
\end{tabular} \\
\hline 4-0 & SHIFTR & \[
\begin{gathered}
0 \\
0 \times 01- \\
0 \times 1 F
\end{gathered}
\] & \begin{tabular}{l}
Right Shift \\
Operations which use right-shift and rounding are shown in Table 2-6. \\
The shift SHIFTR field can be set or cleared by the VSETSHR instruction. \\
No right shift. \\
Refer to the instruction descriptions for information on how the operation is affected by the shift value. During the right-shift, the lower bits are lost, and the shifted value is sign extended. If rounding is enabled (VSTATUS[RND] ==1), then the value will be rounded instead of truncated.
\end{tabular} \\
\hline
\end{tabular}

Table 2-6 shows a summary of the operations that are affected by or modify bits in the VSTATUS register.
Table 2-6. Operation Interaction With VSTATUS Bits
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline Operation \({ }^{(1)}\) & Description & OVFI & OVFR & RND & SAT & SHIFTL & SHIFTR & CPACK & OPACK & DIVE \\
\hline VITDLADDSUB & Viterbi Add and Subtract Low & - & Y & - & Y & - & - & - & - & - \\
\hline VITDHADDSUB & Viterbi Add and Subtract High & - & Y & - & Y & - & - & - & - & - \\
\hline VITDLSUBADD & Viterbi Subtract and Add Low & - & Y & - & Y & - & - & - & - & - \\
\hline VITDHSUBADD & Viterbi Subtract and Add High & - & Y & - & Y & - & - & - & - & - \\
\hline VITBM2 & Viterbi Branch Metric CR 1/2 & - & Y & - & Y & - & - & - & - & - \\
\hline VITBM3 & Viterbi Branch Metric CR 1/3 & - & Y & - & Y & - & - & - & - & - \\
\hline VTRACE \({ }^{(2)}\) & Viterbi Trace-back & - & - & - & - & - & - & - & Y & - \\
\hline
\end{tabular}
\({ }^{(1)}\) Some parallel instructions also include these operations. In this case, the operation will also modify, or be affected by, VSTATUS bits as when used as part of a parallel instruction.
\({ }^{(2)}\) Present on Type-2 VCU only.

Table 2-6. Operation Interaction With VSTATUS Bits (continued)
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline Operation \({ }^{(1)}\) & Description & OVFI & OVFR & RND & SAT & SHIFTL & SHIFTR & CPACK & OPACK & DIVE \\
\hline VITSTAGE \({ }^{(2)}\) & Viterbi Compute 32 Butterfly & - & Y & - & Y & - & - & - & - & - \\
\hline VCADD & Complex \(32+32=32\) & Y & Y & Y & Y & - & Y & - & - & - \\
\hline VCDADD16 & Complex \(16+32=32\) & Y & Y & Y & Y & Y & Y & - & - & - \\
\hline VCDSUB16 & Complex 16-32 = 32 & Y & Y & Y & Y & Y & Y & - & - & - \\
\hline VCMAC & \[
\begin{aligned}
& \text { Complex } 32+32=32 \\
& 16 \times 16=32
\end{aligned}
\] & Y & Y & Y & Y & - & Y & - & - & - \\
\hline VCCMAC \({ }^{(2)}\) & Complex Conjugate \(32+\)
\[
\begin{aligned}
& 32=32 \\
& 16 \times 16=32
\end{aligned}
\] & Y & Y & Y & Y & - & Y & Y & - & - \\
\hline VCMPY & Complex \(16 \times 16=32\) & Y & Y & - & Y & - & - & Y & - & - \\
\hline VCCMPY \({ }^{(2)}\) & Complex Conjugate 16 x \(16=32\) & Y & Y & - & Y & - & - & Y & - & - \\
\hline VCSUB & Complex 32-32 = 32 & Y & Y & Y & Y & - & Y & - & - & - \\
\hline \(\mathrm{VCCON}^{(2)}\) & Complex Conjugate & Y & - & - & Y & - & - & Y & - & - \\
\hline VCSHL16 \({ }^{(2)}\) & Complex Shift Left & Y & Y & - & Y & - & - & Y & - & - \\
\hline VCHR16 \({ }^{(2)}\) & Complex Shift Right & - & - & Y & - & - & - & - & - & - \\
\hline VCMAG \({ }^{(2)}\) & Complex Number Magnitude & - & Y & Y & Y & - & - & - & - & - \\
\hline VNEG & Two's Complement Negation & - & Y & - & Y & - & - & - & - & - \\
\hline VASHR32 \({ }^{(2)}\) & Arithmetic Shift Right & - & - & Y & - & - & - & - & - & - \\
\hline VASHL32 \({ }^{(2)}\) & Arithmetic Shift Left & - & Y & - & Y & - & - & - & - & - \\
\hline VMPYADD \({ }^{(2)}\) & Arithmetic Multiply Add
\[
\begin{aligned}
& 16+((16 \times 16) \gg S H R)= \\
& 16
\end{aligned}
\] & - & Y & Y & Y & - & Y & - & - & - \\
\hline VCFFTx \({ }^{(2)}\) & Complex FFT calculation step
\[
(x=1-10)
\] & Y & Y & Y & Y & - & Y & - & - & - \\
\hline VMOD32 & Modulo 32 \% 16-16 & - & - & - & - & - & - & - & - & Y \\
\hline
\end{tabular}

\subsection*{2.3.3 Repeat Block Register (RB)}

The repeat block instruction (RPTB) applies to devices with the \(\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}\) and the \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\). This instruction allows you to repeat a block of code as shown in Example 2-1.

\section*{Example 2-1. The Repeat Block (RPTB) Instruction uses the RB Register}
```

; find the largest element and put its address in XAR6
; This example makes use of floating-point (C28x + FPU) instructions
;
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,XARO
MOV32 R1H,*XAR0++ ; RSIZE reflects the size of the RPTB block
MAXF32 R0H,R1H ; in this case the block size is 8
MOVSTO NF,ZF
MOVL XAR6,ACC,LT
VECTOR_MAX_END: ; RE indicates the end address. RA is cleared

```

The C28x FPU or VCU 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 2-4. Repeat Block Register (RB)


LEGEND: \(\mathrm{R}=\) Read only; \(-n=\) value after reset
Table 2-7. Repeat Block (RB) Register Field Descriptions
\begin{tabular}{|c|c|c|l|}
\hline Bits & Field & Value & \multicolumn{1}{c|}{ Description } \\
\hline 31 & RAS & & \begin{tabular}{l} 
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. \\
A repeat block was not active when the interrupt was taken. \\
A repeat block was active when the interrupt was taken.
\end{tabular} \\
\hline 30 & RA & 0 & \begin{tabular}{l} 
Repeat Block Active Bit \\
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. \\
This bit is set when the RPTB instruction is executed to indicate that a RPTB is currently active.
\end{tabular} \\
\hline \(29-23\) & RSIZE & & \\
\hline
\end{tabular}

Table 2-7. Repeat Block (RB) Register Field Descriptions (continued)
\begin{tabular}{|c|c|c|l|}
\hline Bits & Field & Value & \\
\hline \(22-16\) & RE & & \begin{tabular}{l} 
Description \\
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. \\
\(R E=\) lower 7 bits of (PC + 1 + RSIZE)
\end{tabular} \\
\hline \(15-0\) & RC & 0 & \begin{tabular}{l} 
Repeat Count \\
The block will not be repeated; it will be executed only once. In this case the repeat active, RA, bit will \\
not be set. \\
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.
\end{tabular} \\
\hline
\end{tabular}

\subsection*{2.4 Pipeline}

This section describes the VCU pipeline stages and presents cases where pipeline alignment must be considered.

\subsection*{2.4.1 Pipeline Overview}

The C28x VCU 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, a FPU instruction, or a VCU instruction. The pipeline flow is shown in Figure 2-5.
Notice that stalls due to normal C28x pipeline stalls (D2) and memory waitstates (R2 and W) will also stall any C28x VCU instruction. Most C28x VCU instructions are single cycle and will complete in the VCU 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 \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) will issue an error if a delay slot has not been handled correctly.

Figure 2-5. C28x + FCU + VCU Pipeline


\subsection*{2.4.2 General Guidelines for VCU Pipeline Alignment}

The majority of the VCU instructions do not require any special pipeline considerations. This section lists the few operations that do require special consideration.
While the C28x+VCU 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 \(\mathrm{C} 28 \mathrm{x}+\mathrm{VCU}\) assembly code.
VCU instructions that require delay slots have a 'p' after their cycle count. For example ' \(2 p\) ' 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.
Table 2-8 outlines the instructions that need delay slots.

INSTRUMENTS

Table 2-8. Operations Requiring a Delay Slot(s)
\begin{tabular}{|c|l|c|}
\hline Operation \(^{(1)}\) & \multicolumn{1}{|c|}{ Description } & \multicolumn{1}{c|}{ Cycles } \\
\hline VITBM3 & Viterbi Branch Metric CR \(1 / 3\) & \(2 \mathrm{p} / 2^{(2)}\) \\
\hline VCMAC & \begin{tabular}{l} 
Complex \(32+32=32\), \\
\(16 \times 16=32\)
\end{tabular} & \(2 p\) \\
\hline VCCMAC \(^{(3)}\) & \begin{tabular}{l} 
Complex Conjugate \(32+32=32\), \\
\(16 \times 16=32\)
\end{tabular} & \(2 p\) \\
\hline VCMPY \(^{(3)}\) & Complex \(16 \times 16=32\) & 2 p \\
\hline VCCMPY \(^{(3)}\) & Complex Conjugate \(16 \times 16=32\) & 2 p \\
\hline VCMAG \(^{(3)}\) & Complex Number Magnitude & 2 \\
\hline VCFFTx \(^{(3)}\) & Complex FFT calculation step \((\mathrm{x}=1-10)\) & \(2 \mathrm{p} / 2^{(2)}\) \\
\hline VMOD32 \(^{\text {VMPYADD }^{(3)}}\) & Modulo \(32 \% 16=16\) & \(9 p\) \\
\hline & \begin{tabular}{l} 
Arithmetic Multiply Add \\
\(16+((16 \times 16) \gg\) SHR \()=16\)
\end{tabular} & 2 p \\
\hline
\end{tabular}
(1) Some parallel instructions also include these operations. In this case, the operation will also modify, or be affected by, VSTATUS bits as when used as part of a parallel instruction.
(2) Variations of the instruction execute differently. In these cases, the user is referred to the description Example 2-2 of the instruction(s) in Section 2.5.
\({ }^{(3)}\) Present on Type-2 VCU only.
An example of the complex multiply instruction is shown in Example 2-2. VCMPY is a \(2 p\) instruction and therefore requires one delay slot. The destination registers for the operation, VR2 and VR3, will be updated one cycle after the instruction for a total of two cycles. Therefore, a NOP or instruction that does not use VR2 or VR3 must follow this instruction.
Any memory stall or pipeline stall will also stall the VCU. This keeps the VCU aligned with the C28x pipeline and there is no need to change the code based on the waitstates of a memory block.

\section*{Example 2-2. 2p Instruction Pipeline Alignment}
```

VCMPY VR3, VR2, VR1, VR0 ; 2 pipeline cycles (2p)
NOP ; 1 cycle delay or non-conflicting instruction
NOP ; Any instruction

```

\subsection*{2.4.3 Parallel Instructions}

Parallel instructions are single opcodes that perform two operations in parallel. The guidelines provided in Section 2.4.2 apply to parallel instructions as well. In this case the cycle count will be given for both operations. For example, a branch metric calculation for code rate of \(1 / 3\) with a parallel load takes \(2 p / 1\) cycles. This means the branch metric portion of the operation takes two pipelined cycles while the move portion of the operation is single cycle. NOPs or other non conflicting instructions must be inserted to align the branch metric calculation portion of the operation as shown in Example 2-4.

\section*{Example 2-3. Branch Metric CR 1/2 Calculation with Parallel Load}
```

; VITBM2 || VMOV32 instruction: branch metrics calculation with parallel load
; VBITM2 is a 1 cycle operation (code rate = 1/2)
; VMOV32 is a 1 cycle operation
VITBM2 VR0 ; Load VRO with the 2 branch metrics
| VMOV32 VR2, @Val ; VR2 gets the contents of Val
<-- VMOV32 completes here (VR2 is valid)
<-- VITBM2 completes here (VRO is valid)
Any instruction, can use VR2 and/or VRO

```

\section*{Example 2-4. Branch Metric CR 1/3 Calculation with Parallel Load}
```

VITBM3 || VMOV32 instruction: branch metrics calculation with parallel load
VBITM3 is a 2p cycle operation (code rate = 1/3)
VMOV32 is a 1 cycle operation
VITBM3 VR0, VR1, VR2 ; Load VR0 and VR1 with the 4 branch metrics
VMOV32 VR2, @Val ; VR2 gets the contents of Val
<instruciton 2> ; <-- VMOV32 completes here (VR2 is valid)
; <-- VITBM3 completes here (VR0, VR1 are valid)
<instruction 3> ; Any instruction, can use VR2 and/or VR0

```

\subsection*{2.4.4 Invalid Delay Instructions}

All VCU, FPU and fixed-point instructions can be used in VCU instruction delay slots as long as source and destination register conflicts are avoided. The C28x+VCU 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.

\section*{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 2-5.

In Example 2-5 the VCMPY instruction uses VR2 and VR3 as its destination registers. The next instruction should not use VR2 or VR3 as a destination. Since the VMOV32 instruction uses the VR3 register a pipeline conflict will be issued by the assembler. This conflict can be resolved by using a register other than VR2 for the VMOV32 instruction as shown in Example 2-6.

INSTRUMENTS

\section*{Example 2-5. Destination Register Conflict}
```

Invalid delay instruction.
Both instructions use the same destination register (VR3)
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VMOV32 VR3, mem32 ; Invalid delay instruction
; <-- VCMPY completes, VR3, VR2 are valid

```

\section*{Example 2-6. Destination Register Conflict Resolved}
```

; Valid delay instruction
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VMOV32 VR7, mem32 ; Valid delay instruction

```

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 2-7. For parallel instructions, the current value of a register can be used in the parallel operation before it is overwritten as shown in Example 2-9.

In Example 2-7 the VCMPY instruction again uses VR3 and VR2 as its destination registers. The next instruction should not use VR3 or VR2 as its source since the VCMPY will take an additional cycle to complete. Since the VCADD instruction uses the VR2 as a source register a pipeline conflict will be issued by the assembler. The use of VR3 will also cause a pipeline conflict. This conflict can be resolved by using a register other than VR2 or VR3 or by inserting a non-conflicting instruction between the VCMPY and VCADD instructions. Since the VNEG does not use VR2 or VR3 this instruction can be moved before the VCADD as shown in Example 2-8.

\section*{Example 2-7. Destination/Source Register Conflict}
```

; Invalid delay instruction.
; VCADD should not use VR2 or VR3 as a source operand
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VCADD VR5, VR4, VR3, VR2 ; Invalid delay instruction
VNEG VRO ; <- VCMPY completes, VR3, VR2 valid

```

\section*{Example 2-8. Destination/Source Register Conflict Resolved}
```

; Valid delay instruction.
;
VCMPY VR3, VR2, VR1, VR0 ; 2p instruction
VNEG VRO ; Non conflicting instruction or NOP
VCADD VR5, VR4, VR3, VR2 ; <- VCMPY completes, VR3, VR2 valid

```

It should be noted that a source register for the second 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 second operation is not in the delay slot of the first operation. Consider Example 2-9 where the VCMPY uses VR3 and VR2 as its destination registers. The VMOV32 is the second operation in the instruction and can freely use VR3 or VR2 as a source register. In the example, the contents of VR3 before the multiply will be used by MOV32.

\section*{Example 2-9. Parallel Instruction Destination/Source Exception}
```

; Valid parallel operation.
;
VCMPY VR3, VR2, VR1, VRO ; 2p/1 instruction
|| VMOV32 mem32, VR3 ; <-- Uses VR3 before the VCMPY update
; <-- mem32 updated
NOP ; <-- Delay for VCMPY
; <-- VR2, VR3 updated

```

Likewise, the source register for the second operation within a parallel instruction can be the same as one of the source registers of the first operation. The VCMPY operation in Example 2-10 uses the VR0 register as one of its sources. This register is also updated by the VMOV32 instruction. The multiplication operation will use the value in VR0 before the VMOV32 updates it.

\section*{Example 2-10. Parallel Instruction Destination/Source Exception}
```

; Valid parallel operation.
VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
|| VMOV32 VR0, mem32 ; <-- Uses VR3 before the VCMPY update
; <-- mem32 updated
NOP ; <-- Delay for VCMPY
; <-- VR2, VR3 updated

```

\section*{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 2-11.
If both operations within a parallel instruction try to update the same destination register as shown in Example 2-11 the assembler will issue an error.

\section*{Example 2-11. Invalid Destination Within a Parallel Instruction}
```

Invalid parallel instruction. Both operations use VR3 as a destination register
VCMPY VR3, VR2, VR1, VR0 ; 2p/1 instruction
VMOV32 VR3, mem32 ; <-- Invalid

```

TEXAS
Instruments

\subsection*{2.5 Instruction Set}

This section describes the assembly language instructions of the VCU. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are independent from C 28 x and \(\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}\) instruction sets.

\subsection*{2.5.1 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. VCU 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 C28x VCU are given in Table 2-9.

Table 2-9. Operand Nomenclature
\begin{tabular}{|l|l|}
\hline Symbol & Description \\
\hline \#16FHi & \begin{tabular}{l} 
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.
\end{tabular} \\
\hline \#16FHiHex & \begin{tabular}{l} 
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.
\end{tabular} \\
\hline \#16FLoHex & A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value \\
\hline \#32Fhex & 32-bit immediate value that represents an IEEE 32-bit floating-point value \\
\hline \#32F & Immediate float value represented in floating-point representation \\
\hline \#0.0 & Immediate zero \\
\hline \#5-bit & 5-bit immediate unsigned value \\
\hline addr & Opcode field indicating the addressing mode \\
\hline Im(X), Im(Y) & Imaginary part of the input X or input Y \\
\hline Im(Z) & Real part of the input X or input Y \\
\hline Re(X), Re(Y) & Real part of the output Z \\
\hline Re(Z) & Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location \\
\hline mem16 & Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location \\
\hline mem32 & VR0 - VR8 registers. Some instructions exclude VR8. Refer to the instruction description for details. \\
\hline VRa & VR0 - VR7 registers, high half. \\
\hline \begin{tabular}{l} 
VR0H, \\
VR1H...VR7H
\end{tabular} & \begin{tabular}{ll|}
\hline VR0L, VR1L....VR7L & VR0 - VR7 registers, low half. \\
\hline VT0, VT1 & Transition bit register VT0 or VT1. \\
\hline VSMn+1: VSMn & Pair of State Metric Registers ( \(n=0: 62, \mathrm{n}\) is even) \\
\hline VRx.By & 32 bit Aliased address space foe each byte of the VRx registers (x=0:7,y =0:3) \\
\hline
\end{tabular} \\
\hline
\end{tabular}

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).

Table 2-10. INSTRUCTION dest, source1, source2 Short Description
\begin{tabular}{|l|l|}
\hline & Description \\
\hline dest1 & Description for the 1st operand for the instruction \\
\hline source1 & Description for the 2nd operand for the instruction \\
\hline source2 & Description for the 3rd operand for the instruction \\
\hline Opcode & This section shows the opcode for the instruction \\
\hline Description & \begin{tabular}{l} 
Detailed description of the instruction execution is described. Any constraints on the operands imposed by \\
the processor or the assembler are discussed.
\end{tabular} \\
\hline Restrictions & Any constraints on the operands or use of the instruction imposed by the processor are discussed. \\
\hline Pipeline & This section describes the instruction in terms of pipeline cycles as described in Section 2.4. \\
\hline Example & \begin{tabular}{l} 
Examples of instruction execution. If applicable, register and memory values are given before and after \\
instruction execution. Some examples are code fragments while other examples are full tasks that assume \\
the VCU is correctly configured and the main CPU has passed it data.
\end{tabular} \\
\hline Operands & \begin{tabular}{l} 
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).
\end{tabular} \\
\hline
\end{tabular} INSTRUMENTS
2.5.2 General InstructionsThe instructions are listed alphabetically, preceded by a summary.
Table 2-11. General Instructions
Title Page
POP RB — Pop the RB Register from the Stack ..... 162
PUSH RB —Push the RB Register onto the Stack ..... 164
RPTB label, loc16 — Repeat A Block of Code ..... 166
RPTB label, \#RC —Repeat a Block of Code ..... 168
VCLEAR VRa - Clear General Purpose Register ..... 170
VCLEARALL —Clear All General Purpose and Transition Bit Registers ..... 171
VCLRCPACK —Clears CPACK bit in the VSTATUS Register ..... 172
VCLRCRCMSGFLIP —Clears CRCMSGFLIP bit in the VSTATUS Register ..... 173
VCLROPACK —Clears OPACK bit in the VSTATUS Register ..... 174
VCLROVFI —Clear Imaginary Overflow Flag ..... 175
VCLROVFR —Clear Real Overflow Flag ..... 176
VMOV16 mem16, VRaH — Store General Purpose Register, High Half ..... 177
VMOV16 mem16, VRaL — Store General Purpose Register, Low Half ..... 178
VMOV16 VRaH, mem16 — Load General Purpose Register, High Half ..... 179
VMOV16 VRaL, mem16 —Load General Purpose Register, Low Half ..... 180
VMOV32 *(0:16bitAddr), loc32 — Move the contents of loc32 to Memory ..... 181
VMOV32 loc32, *(0:16bitAddr) —Move 32-bit Value from Memory to loc32 ..... 182
VMOV32 mem32, VRa - Store General Purpose Register ..... 183
VMOV32 mem32, VSTATUS - Store VCU Status Register ..... 184
VMOV32 mem32, VTa - Store Transition Bit Register ..... 185
VMOV32 VRa, mem32 —Load 32-bit General Purpose Register ..... 186
VMOV32 VRb, VRa —Move 32-bit Register to Register ..... 187
VMOV32 VSTATUS, mem32 -Load VCU Status Register ..... 188
VMOV32 VTa, mem32 —Load 32-bit Transition Bit Register ..... 189
VMOVD32 VRa, mem32 —Load Register with Data Move ..... 190
VMOVIX VRa, \#16I —Load Upper Half of a General Purpose Register with I6-bit Immediate ..... 191
VMOVZI VRa, \#16I —Load General Purpose Register with Immediate ..... 192
VMOVXI VRa, \#16I —Load Low Half of a General Purpose Register with Immediate ..... 193
VRNDOFF —Disable Rounding. ..... 194
VRNDON -Enable Rounding. ..... 195
VSATOFF —Disable Saturation ..... 196
VSATON —Enable Saturation ..... 197
VSETCPACK —Set CPACK bit in the VSTATUS Register ..... 198
VSETCRCMSGFLIP —Set CRCMSGFLIP bit in the VSTATUS Register ..... 199
VSETOPACK — Set OPACK bit in the VSTATUS Register ..... 200
VSETSHL \#5-bit —Initialize the Left Shift Value ..... 201
VSETSHR \#5-bit —Initialize the Left Shift Value ..... 202
VSWAP32 VRb, VRa - 32-bit Register Swap ..... 203
VXORMOV32 VRa, mem32 - 32-bit Load and XOR From Memory ..... 204

\section*{Operands}

RB repeat block register
\begin{tabular}{|c|c|}
\hline Opcode & LSW: 1111111111110001 \\
\hline 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. \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register. \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{12}{*}{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. \\
\hline & ```
; Repeat Block within a High-Priority Interrupt (Non-Interruptible)
;
``` \\
\hline & ; Interrupt: ; RAS \(=\) RA, RA \(=0\) \\
\hline & PUSH RB \(\quad\); Save RB register only if a RPTB block is used in the ISR \\
\hline & . \\
\hline & RPTB _BlockEnd, AL ; Execute the block AL+1 times \\
\hline & . \\
\hline & \\
\hline & _BlockEnd ; End of block to be repeated \\
\hline & ... \\
\hline & POP RB \(\quad\); Restore RB register \\
\hline & IRET ; RA \(=\) RAS, RAS \(=0\) \\
\hline
\end{tabular}

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

```

\section*{See also}

RPTB label, loc16
RPTB label, \#RC

\section*{PUSH RB}

\section*{Push the RB Register onto the Stack}

Operands
RB repeat block register

\section*{Opcode}

\section*{Description}

Flags
Pipeline
Example

\section*{See also}

LSW: 1111111111110000
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.

This instruction does not affect any flags in the VSTATUS register.
This is a single-cycle instruction.
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

```

POP RB

RPTB label, loc16
RPTB label, \#RC

\section*{RPTB label, loc16 Repeat A Block of Code}

\section*{Operands}
\begin{tabular}{ll}
\hline 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.
\end{tabular}
\begin{tabular}{ccccc} 
Opcode & LSW: 1011 & 0101 & \(0 b b b\) bbbb \\
MSW: 0000 & 0000 & loc16
\end{tabular}

\section*{Description}

\section*{Restrictions}

Initialize repeat block loop, repeat count from [loc16]
- The maximum block size is \(\leq 12716\)-bit words.
- An even aligned block must be \(\geq 916\)-bit words.
- An odd aligned block must be \(\geq 816\)-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.

\section*{Flags}

Pipeline

\section*{Example}

This instruction does not affect any flags in the VSTATUS register.
This instruction takes four cycles on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

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)
Note: This example makes use of floating-point (C28x+FPU) instructions
;
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,XARO MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H ; max size = 127 words
MOVSTO 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

\section*{RPTB label, \#RC Repeat a Block of Code}

\section*{Operands}
\begin{tabular}{ll}
\hline 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.
\end{tabular}

\section*{Opcode LSW: 10110101 1bbb bbbb}

MSW: cccc cccc cccc cccc

\section*{Description}

\section*{Restrictions}

\section*{Flags}

Pipeline

\section*{Example}

Repeat a block of code. The repeat count is specified as a immediate value.
- The maximum block size is \(\leq 12716\)-bit words.
- An even aligned block must be \(\geq 916\)-bit words.
- An odd aligned block must be \(\geq 816\)-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.

This instruction does not affect any flags in the VSTATUS register.
This instruction takes one cycle on the first iteration and zero cycles thereafter. No special pipeline alignment is required.

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)
Note: This example makes use of floating-point (C28x+FPU) instructions
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,XARO MOV32 R1H,*XAR0++ ; min size = 8, 9 words
MAXF32 R0H,R1H ; max size = 127 words
MOVSTO 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, \#5 ; 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

```

\section*{See also}

POP RB
PUSH RB
RPTB label, loc16

\section*{VCLEAR VRa}

Operands

Opcode

Description

Flags
Pipeline

\section*{Example}

See also

Clear General Purpose Register
VRa General purpose register: VR0, VR1... VR8
```

LSW: 1110 0110 1111 1000
MSW: 0000 0000 0000 aaaa

```

Clear the specified general purpose register. \(\mathrm{VRa}=0 \times 00000000\);

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
;
; Code fragment from a viterbi traceback
; For the first iteration the previous state metric must be initalized to zero (VRO).
;
VCLEAR VR0 ; Clear the VR0 register
MOVL XAR5,*+XAR4[0] ; Point XAR5 to an array
;
For first stage
;
VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++,VR0,VT0,VT1 ; Uses VRO (which is zero)
;
etc...
;
VCLEARALL
VTCLEAR

INSTRUMENTS

\section*{VCLEARALL}

\section*{Clear All General Purpose and Transition Bit Registers}


\section*{See also}

VCLEAR VRa
VTCLEAR

\section*{VCLRCPACK}

Operands
Opcode
Description

Flags
Pipeline
Example

See also
VSETCPACK

\section*{VCLRCRCMSGFLIP Clears CRCMSGFLIP bit in the VSTATUS Register}
\begin{tabular}{|c|c|}
\hline \multirow[t]{2}{*}{Operands} & \\
\hline & none \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110010100101101 \\
\hline & MSW: 0000000000000000 \\
\hline Description & Clear the CRCMSGFLIP bit in the VSTATUS register. This causes the VCU to process message bits starting from most-significant to least-significant for CRC computation. In this case, bytes loaded from memory are fed directly for CRC computation. \\
\hline Flags & This instruction clears the CRCMSGFLIP bit in the VSTATUS register. \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{3}{*}{Example} & \begin{tabular}{l}
; Clear the CRCMSGFLIP bit to have the CRC routine process the \\
; input message in big-endian format. The CRCMSGFLIP bit is \\
; cleared on reset
\end{tabular} \\
\hline & VCLRCRCMSGFLIP \\
\hline & LCR _CRC_run8Bit \\
\hline See also & VSETCRCMSGFLIP \\
\hline
\end{tabular}

\section*{VCLROPACK}

Operands

Opcode

Description

Pipeline This is a single-cycle instruction.

\section*{Example}

See also

Flags \(\quad\) This instruction clears the OPACK bit in the VSTATUS register.
Clears OPACK bit in the VSTATUS Register
none

LSW: 1110010100100101 MSW: 0000000000000000

Clear the OPACK bit in the VSTATUS register. This bit affects the packing order of the traceback output bits (using the VTRACE instructions). When the bit is set to 0 it forces the bits generated from the traceback operation to be loaded through the LSb of the destination register (or memory location) with the older bits being left shifted.

VSETOPACK

\section*{VCLROVFI}

Operands

Opcode

\section*{Description}

Flags
Pipeline

\section*{Clear Imaginary Overflow Flag}
none

LSW: 1110010100001011
Clear the real overflow flag in the VSTATUS register. To clear the real flag, use the VCLROVFR instruction. The imaginary flag bit can be set by instructions shown in Table 2-6. Refer to individual instruction descriptions for details.
VSTATUS [OVFR] \(=0\);
This instruction clears the OVFI flag.

Example
See also

This is a single-cycle instruction.

VCLROVFR
VRNDON
VSATFOFF
VSATON

\section*{VCLROVFR}

Operands

Opcode

\section*{Description}

Flags
Pipeline

\section*{Clear Real Overflow Flag}
none

LSW: 1110010100001010
Clear the real overflow flag in the VSTATUS register. To clear the imaginary flag, use the VCLROVFI instruction. The imaginary flag bit can be set by instructions shown in Table 2-6. Refer to individual instruction descriptions for details.
VSTATUS [OVFR] \(=0\);
This instruction clears the OVFR flag.

Example
See also

VCLROVFI
VRNDON
VSATFOFF
VSATON

\section*{VMOV16 mem16, VRaH Store General Purpose Register, High Half}

\section*{Operands}
\begin{tabular}{ll}
\hline mem16 & Pointer to a 16-bit memory location. This will be the source for the VMOV16. \\
VRaH & High word of a general purpose register: VR0H, VR1H...VR8H. \\
\hline
\end{tabular}

Opcode

Description

\section*{Flags}

Pipeline
Example
See also
VMOV16 VRaH, mem16

\section*{VMOV16 mem16, VRaL Store General Purpose Register, Low Half}

Operands
\begin{tabular}{ll}
\hline mem16 & Pointer to a 16-bit memory location. This will be the destination of the VMOV16. \\
VRaL & Low word of a general purpose register: VR0L, VR1L...VR8L.
\end{tabular}

Opcode

Description

\section*{Flags}

Pipeline
Example
See also
```

LSW: 1110 00100001 1000
MSW: 0000 aaaa mem16

```

Store the low 16-bits of the specified general purpose register into the 16-bit memory location.
[mem16] = VRa[15:0];
This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.

VMOV16 VRaL, mem16

\section*{VMOV16 VRaH, mem16 Load General Purpose Register, High Half}

\section*{Operands}
\begin{tabular}{ll}
\hline VRHL & High word of a general purpose register: VR0H, VR1H....VR8H \\
mem16 & Pointer to a 16-bit memory location. This will be the source for the VMOV16. \\
\hline
\end{tabular}

Opcode

Description

\section*{Flags}

\section*{Pipeline}

This instruction does not affect any flags in the VSTATUS register.

\section*{Example}

See also
```

;1st Iteration
VMOV32 VR4, *+XAR3[0] ; VR4H = m, VR4L=n Load m,n
VMOV16 VR0H, *+XAR5[0] ; VR0H = J, VR0L = I Init I, J
VMOV32 VR1, *+XAR3[4] ; VR1H = u, VR1L = a load u, a
VMOV32 VR6, VR0 ; Save current {J,I} in VR6
; etc.

```

VMOV16 mem16, VRaH

\section*{VMOV16 VRaL, mem16 Load General Purpose Register, Low Half}

Operands
\begin{tabular}{ll}
\hline VRaL & Low word of a general purpose register: VROL, VR1L....VR8L \\
mem16 & Pointer to a 16-bit memory location. This will be the source for the VMOV16. \\
\hline
\end{tabular}

Opcode

\section*{Description}

Flags

\section*{Pipeline}
```

LSW: 1110 0010 1100 1001
MSW: 0000 aaaa mem16

```

\section*{Example}
```

Loop will run }106\mathrm{ times for }212\mathrm{ inputs to decoder
Code fragment from viterbi decoder
;
_LOOP:
;
;
Calculate the branch metrics for code rate = 1/3
Load VROL, VR1L and VR2L with inputs
to the decoder from the array pointed to by XAR5
VMOV16 VR0L, *XAR5++
VMOV16 VR1L, *XAR5++
VMOV16 VR2L, *XAR5++
;
VROL = BMO
VR0H = BM1
VR1L = BM2
VR1H = BM3
VR2L = pt_old[0]
VR2H = pt_old[1]
VITBM3 VR0, VR1, VR2
VMOV32 VR2, *XAR1++
; etc...

```

See also VMOV16 mem16, VRaL

\section*{VMOV32 *(0:16bitAddr), loc32 Move the contents of loc32 to Memory}

\section*{Operands}
\begin{tabular}{ll}
\hline\(I^{*}(0: 16 \mathrm{bitAddr})\) & Address of 32-bit Destination Location (VCU register) \\
loc32 & Source Location (CPU register) \\
\hline
\end{tabular}

Opcode

Description
```

LSW: 1011 1101 loc32
MSW: IIII IIII IIII IIII

```

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]

\section*{Flags}

\section*{Pipeline}

\section*{Example}

See also
VMOV32 VRa, mem32
VMOV32 VRb, VRa
VMOV32 loc32, *(0:16bitAddr)

\section*{VMOV32 loc32, *(0:16bitAddr) Move 32-bit Value from Memory to loc32}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Operands} & \\
\hline & loc32 Destination Location (CPU register) \\
\hline & *(0:16bitAddr) Address of 32-bit Source Value (VCU register) \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 10111111 loc32 \\
\hline & MSW: IIII IIII IIII 1 III \\
\hline Description & Copy the 32-bit value referenced by 0:16bitAddr to the location indicated by loc32 [loc32] \(=\) [0:16bitAddr] \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register. \\
\hline Pipeline & This is two-cycle instruction. \\
\hline \multirow[t]{11}{*}{Example} & ; A single NOP is needed before the operation so as to read the \\
\hline & ; correct VCU's VRx register value \\
\hline & vmov32 VRa,*XAR7++ ; VRa = [*XAR7] \\
\hline & NOP ; Pipeline alignment \\
\hline & VMOV32 @ACC, VRa ; ACC = VRa \\
\hline & ; Two NOPs are needed before the operation so as to read the \\
\hline & ; correct VCU's VSMx or VRx.By register value. \\
\hline & VMOV32 VSM1: VSM0, *XAR7 ; VSM1:VSM0 \(=\) [*XAR7]
NOP \\
\hline & NOP \(\quad\); Pipeline alignment \\
\hline & NOP ; Pipeline alignment \\
\hline & vMOV32 @ACC, VSMO ; AH:AL = VSM1:VSMO \\
\hline \multirow[t]{3}{*}{See also} & VMOV32 VRa, mem32 \\
\hline & VMOV32 VRb, VRa \\
\hline & VMOV32 *(0:16bitAddr), loc32 \\
\hline
\end{tabular}

\section*{VMOV32 mem32, VRa Store General Purpose Register}

\section*{Operands}
\begin{tabular}{ll}
\hline mem32 & Pointer to a 32-bit memory location. This will be the destination of the VMOV32. \\
VRa & General purpose register VR0, VR1... VR8 \\
\hline
\end{tabular}

Opcode

\section*{Description}

Flags

\section*{Pipeline}
```

LSW: 1110 0010 0000 0100
MSW: 0000 aaaa mem32

```

Example
See also
VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32 VTa, mem32

\section*{VMOV32 mem32, VSTATUS Store VCU Status Register}

Operands
\begin{tabular}{ll}
\hline mem32 & Pointer to a 32-bit memory location. This will be the destination of the VMOV32. \\
VSTATUS & VCU status register. \\
\hline
\end{tabular}

Opcode

Description
Store the VSTATUS register into the memory location pointed to by mem32.
```

[mem32] = VSTATUS;

```

Flags
Pipeline
This instruction does not modify any flags in the VSTATUS register.

Example
See also
VMOV32 mem32, VRa
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32 VSTATUS, mem32
VMOV32 VTa, mem32

\section*{VMOV32 mem32, VTa Store Transition Bit Register}

\section*{Operands}
\begin{tabular}{ll}
\hline mem32 & pointer to a 32-bit memory location. This will be the destination of the VMOV32. \\
VTa & Transition bits register VT0 or VT1 \\
\hline
\end{tabular}

Opcode

\section*{Description}

Flags

\section*{Pipeline}

LSW: 1110001000000101
MSW: 0000 00tt mem32
Store the 32-bits of the specified transition bits register into the memory location pointed to by mem32.
[mem32] = VTa;

Example
See also
VMOV32 mem32, VRa
VMOV32 mem32, VSTATUS
VMOV32 VRa, mem32
VMOV32 VSTATUS, mem32
VMOV32 VTa, mem32

\section*{VMOV32 VRa, mem32 Load 32-bit General Purpose Register}

\section*{Operands}
\begin{tabular}{ll} 
VRa & General purpose register VR0, VR1....VR8 \\
mem32 & Pointer to a 32-bit memory location. This will be the source of the VMOV32. \\
\hline
\end{tabular}

Opcode

Description

Flags

\section*{Pipeline}
```

LSW: 1110 0011 11110000
MSW: 0000 aaaa mem32

```

Example
See also
VMOV32 mem32, VRa
VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VSTATUS, mem32
VMOV32 VTa, mem32

\section*{VMOV32 VRb, VRa Move 32-bit Register to Register}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Operands} & \\
\hline & VRa General purpose destination register VR0....VR8 \\
\hline & VRb General purpose source register VR0...VR8 \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110011011110010 \\
\hline & MSW: 00000010 bbbb aaaa \\
\hline Description & Move a 32-bit value from one general purpose VCU register to another. vRa = [mem32]; \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register. \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{3}{*}{Example} & ; Swap VRO and VR1 using VR2 as temporary storage ; \\
\hline & vMov32 VR2, VR1 \\
\hline & \begin{tabular}{l}
vMov32 VR1, vR0 \\
VMOV32 VR0, VR2
\end{tabular} \\
\hline \multirow[t]{4}{*}{See also} & VMOV32 mem32, VRa \\
\hline & VMOV32 mem32, VSTATUS \\
\hline & VMOV32 mem32, VTa \\
\hline & VMOV32 VTa, mem32 \\
\hline
\end{tabular}

\section*{VMOV32 VSTATUS, mem32 Load VCU Status Register}

Operands
\begin{tabular}{ll}
\hline VSTATUS & VCU status register \\
mem32 & Pointer to a 32-bit memory location. This will be the source of the VMOV32. \\
\hline
\end{tabular}

Opcode

Description
LSW: 1110001010110000
MSW: 00000000 mem32
Load the VSTATUS register with the 32-bit value in memory pointed to by mem32. vSTATUS = [mem32];

Flags
Pipeline
This instruction modifies all bits within the VSTATUS register.

Example
See also
VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32 VTa, mem32

\section*{VMOV32 VTa, mem32 Load 32-bit Transition Bit Register}

\section*{Operands}
\begin{tabular}{ll}
\hline VTa & Transition bit register: VT0, VT1 \\
mem32 & Pointer to a 32-bit memory location. This will be the source of the VMOV32. \\
\hline
\end{tabular}

Opcode

Description

Flags

\section*{Pipeline}

LSW: 1110001111110001
MSW: 0000 00tt mem32
Load the specified transition bit register with the 32-bit value in memory pointed to by mem32.
\(\mathrm{VTa}=\) [mem32];

Example
See also
VMOV32 mem32, VSTATUS
VMOV32 mem32, VTa
VMOV32 VRa, mem32
VMOV32 VSTATUS, mem32

\section*{VMOVD32 VRa, mem32 Load Register with Data Move}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Operands} & \\
\hline & VRa General purpose registger, VR0, VR1.... VR8 \\
\hline & mem32 Pointer to a 32-bit memory location. This will be the source of the VMOV32. \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001000100100 \\
\hline & MSW: 0000 aaaa mem32 \\
\hline \multirow[t]{2}{*}{Description} & Load the specified general purpose register with the 32 -bit value in memory pointed to by mem32. In addition, copy the next 32-bit value in memory to the location pointed to by mem32. \\
\hline & \[
\begin{aligned}
& \mathrm{VRa}=[\text { mem32]; } \\
& {[\text { mem32 }+2]=[\text { mem32]; }}
\end{aligned}
\] \\
\hline
\end{tabular}

Flags
Pipeline
Example
See also

\section*{VMOVIX VRa, \#16I Load Upper Half of a General Purpose Register with I6-bit Immediate}

Operands
\begin{tabular}{ll}
\hline VRa & General purpose registger, VR0, VR1... VR8 \\
\(\# 161\) & 16-bit immediate value \\
\hline
\end{tabular}

Opcode

Description
```

LSW: 1110 0111 1110 IIII

```
MSW: IIII IIII IIII aaaa

Load the upper 16-bits of the specified general purpose register with an immediate value. Leave the upper 16 -bits of the register unchanged.
```

VRa[15:0] = unchanged;
VRa[31:16] = \#16I;

```

\section*{Flags}

\section*{Pipeline}

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.

\section*{Example}

\section*{See also}

VMOVZI VRa, \#16I VMOVXI VRa, \#16I

\section*{VMOVZI VRa, \#16I Load General Purpose Register with Immediate}

Operands
\begin{tabular}{ll}
\hline VRa & General purpose registger, VR0, VR1 ..VR8 \\
\(\# 161\) & 16-bit immediate value \\
\hline
\end{tabular}

Opcode

Description

\section*{Flags}

Pipeline
This is a single-cycle instruction.
Example
See also
VMOVIX VRa, \#16I VMOVXI VRa, \#16I

\section*{VMOVXI VRa, \#16I Load Low Half of a General Purpose Register with Immediate}

Operands
\begin{tabular}{ll}
\hline VRa & General purpose register, VR0 - VR8 \\
\(\# 161\) & 16-bit immediate value \\
\hline
\end{tabular}

Opcode

Description

\section*{Flags}

\section*{Pipeline}

\section*{Example}

\section*{See also}

VMOVIX VRa, \#16I VMOVZI VRa, \#16I

\section*{VRNDOFF}

Operands

Opcode

\section*{Description}

Flags

\section*{Pipeline}

\section*{Example}

See also

Disable Rounding
none

LSW: 1110010100001001
This instruction disables the rounding mode by clearing the RND bit in the VSTATUS register. When rounding is disabled, the result of the shift right operation for addition and subtraction operations will be truncated instead of rounded. The operations affected by rounding are shown in Table 2-6. Refer to the individual instruction descriptions for information on how rounding effects the operation. To enable rounding use the VRNDON instruction.

For more information on rounding, refer to Section 2.3.2.
vSTATUS[RND] = 0;
This instruction clears the RND bit in the VSTATUS register. It does not change any flags.

This is a single-cycle instruction.

VCLROVFI
VCLROVFR
VRNDON
VSATFOFF
VSATON

\section*{VRNDON}

Operands

Opcode

\section*{Description}

Flags

\section*{Pipeline}

Example
See also

\section*{Enable Rounding}
none

LSW: 1110010100001000
This instruction enables the rounding mode by setting the RND bit in the VSTATUS register. When rounding is enabled, the result of the shift right operation for addition and subtraction operations will be rounded instead of being truncated. The operations affected by rounding are shown in Table 2-6. Refer to the individual instruction descriptions for information on how rounding effects the operation. To disable rounding use the VRNDOFF instruction.

For more information on rounding, refer to Section 2.3.2.
VSTATUS[RND] = 1;
This instruction sets the RND bit in the VSTATUS register. It does not change any flags.
This is a single-cycle instruction.

VCLROVFI
VCLROVFR
VRNDOFF
VSATFOFF
VSATON

\section*{VSATOFF}

Operands

Opcode

\section*{Description}

Flags

Pipeline

\section*{Example}

See also

\section*{Disable Saturation}
none

LSW: 1110010100000111
This instruction disables the saturation mode by clearing the SAT bit in the VSTATUS register. When saturation is disabled, results of addition and subtraction are allowed to overflow or underflow. When saturation is enabled, results will instead be set to a maximum or minimum value instead of being allowed to overflow or underflow. To enable saturation use the VSATON instruction.

\section*{VSTATUS[SAT] = 0}

This instruction clears the the SAT bit in the VSTATUS register. It does not change any flags.

This is a single-cycle instruction.

VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON

\section*{VSATON}

Operands

Opcode

\section*{Description}

Flags
Pipeline

\section*{Example}

See also

\section*{Enable Saturation}
none

LSW: 1110010100000110
This instruction enables the saturation mode by setting the SAT bit in the VSTATUS register. When saturation is enables, results of addition and subtraction are not allowed to overflow or underflow. Results will, instead, be set to a maximum or minimum value. To disable saturation use the VSATOFF instruction..
```

VSTATUS[SAT] = 1

```

This instruction sets the SAT bit in the VSTATUS register. It does not change any flags.
This is a single-cycle instruction.

VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATOFF

\section*{VSETCPACK}

\section*{Operands}

Opcode

\section*{Description}

\section*{Flags}

Pipeline

\section*{Example}

\section*{See also}

\section*{Set CPACK bit in the VSTATUS Register}
none

LSW: 1110010100100001
Set the CPACK bit in the VSTATUS register. This causes the VCU to process complex data, in complex math operations, in the VRx registers as follows: \(\operatorname{VRx}[31: 16]\) holds the Imaginary part, VRx[15:0] holds the Real part

This instruction sets the CPACK bit in the VSTATUS register.
This is a single-cycle instruction.
```

; complex conjugate multiply| (a + jb)*(c + jd)=(ac+bd) +j(bc-ad)
VSETCPACK ; cpack = 1 imag part in low word
VMOV32 VR0, *XAR4++ ; load 1st complex input | a + jb
VMOV32 VR1, *XAR4++ ; load second complex input | c + jd
VCCMPY VR3, VR2, VR1, VR0

```

VCLRCPACK

\section*{VSETCRCMSGFLIP Set CRCMSGFLIP bit in the VSTATUS Register}
\begin{tabular}{|c|c|}
\hline \multirow[t]{2}{*}{Operands} & \\
\hline & none \\
\hline Opcode & LSW: 1110010100101100 \\
\hline Description & Set the CRCMSGFLIP bit in the VSTATUS register. This causes the VCU to process message bits starting from least-significant to most-significant for CRC computation. In this case, bytes loaded from memory are "flipped" and then fed for CRC computation. \\
\hline Flags & This instruction sets the CRCMSGFLIP bit in the VSTATUS register. \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline Example & ```
; Set the CRCMSGFLIP bit, each word has all its bits reversed
    prior to the CRC being calculated
;
    VSETCRCMSGFLIP
    LCR _CRC_run8Bit
VCLRCRCMSGFLIP
``` \\
\hline See also & VCLRCRCMSGFLIP \\
\hline
\end{tabular}

\section*{VSETOPACK}

Operands

Opcode

\section*{Description}

Flags

\section*{Pipeline}

Example

See also

\section*{Set OPACK bit in the VSTATUS Register}
none

LSW: 1110010100100011
Set the OPACK bit in the VSTATUS register. This bit affects the packing order of the traceback output bits (using the instructions). When the bit is set to 1 it forces the bits generated from the traceback operation to be loaded through the MSb of the destination register (or memory location) with the older bits being right-shifted. This instruction sets the OPACK bit in the VSTATUS register.

This instruction sets the OPACK bit in the VSTATUS register.
This is a single-cycle instruction.
```

VSETOPACK ; VSTATUS.OPACK = 1, start packing the decoded
; bits from trace back into VT1 starting from the
; MSb, this obviates the need to manually flip the
; result each time
; etc..

```

VCLROPACK

\section*{VSETSHL \#5-bit}

Operands

Opcode

\section*{Description}

Flags

Pipeline

\section*{Example}

See also

\section*{VSETSHR \#5-bit Initialize the Left Shift Value}

Operands

Opcode

\section*{Description}

Flags

Pipeline

\section*{Example}

See also
VSETSHL \#5-bit

\section*{VSWAP32 VRb, VRa 32-bit Register Swap}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Operands} & \\
\hline & VRb General purpose register VR0...VR8 \\
\hline & VRab General purpose register VR0...VR8 \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110011011110010 \\
\hline & MSW: 00000011 bbbb aaaa \\
\hline Description & Swap the contents of the 32-bit general purpose VCU registers VRa and VRb. \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{2}{*}{Example} & ; Swap VRO and VR1 using VSWAP32 instruction \\
\hline & ; \\
\hline \multirow[t]{5}{*}{See also} & VMOV32 mem32, VSTATUS \\
\hline & VMOV32 mem32, VTa \\
\hline & VMOV32 VRa, mem32 \\
\hline & VMOV32VRbVRa \\
\hline & VMOV32VTamem32 \\
\hline
\end{tabular}

\section*{VXORMOV32 VRa, mem32 32-bit Load and XOR From Memory}

\section*{Operands}
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VRa & General purpose register VR0...VR8 \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}

Opcode

Description
```

LSW: 1110 0011 11110000
MSW: 0000 aaaa MMMM MMMM

```

XOR the contents of the VRa register with a long word from memory and store the result back into VRa
\[
\mathrm{VRa}=\mathrm{VRa}{ }^{\wedge} \operatorname{mem} 32
\]

Flags
This instruction does not affect any flags in the VSTATUS register.
Pipeline
This is a single-cycle instruction.
```

Example
VXORMOV32 VR0, *+XAR4[0] ;VR0=VR0 ^ *XAR4[0]

```

See also

\subsection*{2.5.3 Arithmetic Math Instructions}

The instructions are listed alphabetically, preceded by a summary.

\section*{Table 2-12. Arithmetic Math Instructions}
Title Page
VASHL32 VRa << \#5-bit —Arithmetic Shift Left ..... 206
VASHR32 VRa >> \#5-bit —Arithmetic Shift Right ..... 207
VBITFLIP VRa —Bit Flip. ..... 208
VLSHL32 VRa << \#5-bit —Logical Shift Left ..... 209
VLSHR32 VRa >> \#5-bit —Logical Shift Right ..... 210
VNEG VRa - Two's Complement Negate. ..... 211

\section*{VASHL32 VRa << \#5-bit Arithmetic Shift Left}

Operands
\begin{tabular}{ll}
\hline VRa & VRa can be VR0 - VR7. VRa can not be VR8. \\
\#5-bit & 5-bit unsigned immediate value \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011011110010 \\
& MSW: 00000111 IIII Iaaa
\end{tabular}

Description Arithmetic left shift of VRa
```

If(VSTATUS[SAT] == 1){
VRa = sat(VRa << \#5-bit Immediate)
}else {
VRa = VRa << \#5-bit Immediate
}

```

Flags

Pipeline
Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the 32 -bit signed result after the shift left operation overflows

This is a single-cycle instruction

VASHL32 VR4 << \#16 ; VR4 := VR4 << 16
VASHR32 VRa>> \#5-bit

\section*{VASHR32 VRa >> \#5-bit Arithmetic Shift Right}

Operands
\begin{tabular}{ll}
\hline VRa & VRa can be VR0 - VR7. VRa can not be VR8. \\
\#5-bit & 5-bit unsigned immediate value \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Opcode & LSW: 1110011011110010 \\
\hline & MSW: 00001000 IIII Iaaa \\
\hline Description & Arithmetic right shift of VRa \\
\hline & ```
If(VSTATUS[RND] == 1){
    VRa = rnd(VRa >> #5-bit Immediate)
``` \\
\hline & VRa = VRa >> \#5-bit Immediate \\
\hline & \(\}\) ) \\
\hline
\end{tabular}

Flags

\section*{Pipeline}

Example
See also

This instruction does not affect any flags in the VSTATUS register
This is a single-cycle instruction
VASHR32 VR1 >> \#16 ; VR1 := VR1 >> 16 (sign extended)
VASHL32 VRa\#5-bit

\section*{VBITFLIP VRa}

Bit Flip
Operands
VRa General purpose register VR0...VR8

\section*{Opcode}

LSW: 101000010010 aaaa
Description
Reverse the bit order of VRa register VRa[31:0] = VRa[0:31]

Flags \(\quad\) This instruction does not affect any flags in the VSTATUS register
Pipeline
This is a single-cycle instruction
Example VBITfLIP VR1 ; VR1 (31:0) := VR1 (0:31)
See also

\section*{VLSHL32 VRa << \#5-bit Logical Shift Left}

Operands
\begin{tabular}{ll}
\hline VRa & VRa can be VR0 - VR7. VRa can not be VR8. \\
\(\# 5\)-bit & 5-bit unsigned immediate value \\
\hline
\end{tabular}

Opcode
LSW: 1110011011110010
MSW: 00000101 IIII Iaaa
Description Logical right shift of VRa
VRa \(=\) VRa \(\ll\) \#-bit Immediate
Flags This instruction does not affect any flags in the VSTATUS register
Pipeline
Example
See also
This is a single-cycle instruction
VLSHL32 VR0 << \#16 ; VR0 := VR0 << 16
VLSHL32 VRa>> \#5-bit

\section*{VLSHR32 VRa >> \#5-bit Logical Shift Right}

Operands
\begin{tabular}{ll}
\hline VRa & VRa can be VR0 - VR7. VRa can not be VR8. \\
\(\# 5\)-bit & 5-bit unsigned immediate value \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Opcode & ```
LSW: 1110 0110 1111 0010
MSW: 0000 0110 IIII Iaaa
``` \\
\hline Description & Logical right shift of VRa \\
\hline & VRa = VRa >> \#5-bit Immediate \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register \\
\hline Pipeline & This is a single-cycle instruction \\
\hline Example & VLSHR32 VRO >> \#16 ; VRO := VR0 >> 16 (no sign extension) \\
\hline See also & VLSHL32 VRa\#5-bit \\
\hline
\end{tabular}

\section*{VNEG VRa}

\section*{Operands}

\section*{Opcode}

\section*{Description}

Flags

Pipeline
Example
See also

Two's Complement Negate

VRa VRa can be VR0 - VR7. VRa can not be VR8.

LSW: 111001010001 aaaa
Complex add operation.
```

// SAT is vSTATUS[SAT]
//
if (VRa == 0x800000000)
{
if(SAT == 1)
{
VRa = 0x7FFFFFFF;
}
else
{
VRa = 0x80000000;
}
}
else
{
VRa = - VRa
}

```

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the input to the operation is \(0 \times 80000000\).

This is a single-cycle instruction.

VCLROVFR
VSATON
VSATOFF

\subsection*{2.5.4 Complex Math Instructions}

The instructions are listed alphabetically, preceded by a summary.

\section*{Table 2-13. Complex Math Instructions}
Title Page
VCADD VR5, VR4, VR3, VR2 -Complex 32 + 32 = 32 Addition ..... 213
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 - Complex 32+32 = 32 Add with Parallel Load ..... 215
VCADD VR7, VR6, VR5, VR4 —Complex \(32+32=32\) - Addition ..... 217
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 —Complex Conjugate Multiply and Accumulate ..... 219
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 -: Complex Conjugate Multiply and Accumulate with Parallel Load ..... 221
VCCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ —Complex Conjugate Multiply and Accumulate ..... 223
VCCMPY VR3, VR2, VR1, VR0 - Complex Conjugate Multiply ..... 226
VCCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa - Complex Conjugate Multiply with Parallel Store ..... 228
VCCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 —Complex Conjugate Multiply with Parallel Load ..... 230
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 - Complex Conjugate Multiply with Parallel Load ..... 232
VCCON VRa - Complex Conjugate ..... 234
VCDADD16 VR5, VR4, VR3, VR2 -Complex 16 + 32 = 16 Addition ..... 235
VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 —Complex Double Add with Parallel Load ..... 239
VCDSUB16 VR6, VR4, VR3, VR2 -Complex 16-32 = 16 Subtract ..... 242
VCDSUB16 VR6, VR4, VR3, VR2 || VMOV32 VRa, mem32 -Complex 16-32 = 16 Subtract with Parallel Load ..... 246
VCFLIP VRa - Swap Upper and Lower Half of VCU Register ..... 249
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 - Complex Multiply and Accumulate ..... 250
VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ —Complex Multiply and Accumulate ..... 252
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 - Complex Multiply and Accumulate with Parallel Load ..... 256
VCMAG VRb, VRa - Magnitude of a Complex Number ..... 258
VCMPY VR3, VR2, VR1, VR0 -Complex Multiply ..... 259
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa — Complex Multiply with Parallel Store ..... 261
VCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 - Complex Multiply with Parallel Load ..... 263
VCSHL16 VRa << \#4-bit —Complex Shift Left ..... 265
VCSHR16 VRa >> \#4-bit —Complex Shift Right ..... 266
VCSUB VR5, VR4, VR3, VR2 —Complex 32-32 = 32 Subtraction ..... 267
VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 —Complex Subtraction ..... 269

\section*{VCADD VR5, VR4, VR3, VR2 Complex 32 + 32 = 32 Addition}

Before the operation, the inputs should be loaded into registers as shown below. Each operand for this instruction includes a 32-bit real and a 32-bit imaginary part.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR5 & 32-bit integer representing the real part of the first input: \(\operatorname{Re}(\mathrm{X})\) \\
VR4 & 32-bit integer representing the imaginary part of the first input: \(\operatorname{Im}(\mathrm{X})\) \\
VR3 & 32-bit integer representing the real part of the 2nd input: \(\operatorname{Re}(\mathrm{Y})\) \\
VR2 & 32-bit integer representing the imaginary part of the 2nd input: \(\operatorname{Im}(\mathrm{Y})\) \\
\hline
\end{tabular}

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR5 & \begin{tabular}{l}
32 -bit integer representing the real part of the result: \\
\(\operatorname{Re}(Z)=\operatorname{Re}(X)+(\operatorname{Re}(Y) \gg\) SHIFTR \()\)
\end{tabular} \\
VR4 & \begin{tabular}{l}
32 -bit integer representing the imaginary part of the result: \\
\(\operatorname{Im}(Z)=\operatorname{Im}(X)+(\operatorname{Im}(Y) \gg\) SHIFTR \()\)
\end{tabular} \\
\hline
\end{tabular}

\section*{Opcode}

\section*{Description}

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR4 computation (imaginary part) overflows or underflows.

Pipeline
This is a single-cycle instruction.

\section*{Example}

See also VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR \#5-bit

\section*{VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 Complex 32+32 = 32 Add with Parallel Load}

Operands Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR5 & 32-bit integer representing the real part of the first input: \(\operatorname{Re}(X)\) \\
VR4 & 32-bit integer representing the imaginary part of the first input: \(\operatorname{Im}(X)\) \\
VR3 & 32-bit integer representing the real part of the 2nd input: \(\operatorname{Re}(Y)\) \\
VR2 & \begin{tabular}{l} 
32-bit integer representing the imaginary part of the 2nd input: \(\operatorname{Im}(Y)\) \\
penter to a 32-bit memory location
\end{tabular}
\end{tabular}

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR5 and VR4 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR5 & \begin{tabular}{l} 
32-bit integer representing the real part of the result: \\
\(\operatorname{Re}(Z)=\operatorname{Re}(\mathrm{X})+(\operatorname{Re}(\mathrm{Y}) \gg\) SHIFTR)
\end{tabular} \\
VR4 & \begin{tabular}{l}
32 -bit integer representing the imaginary part of the result: \\
\(\operatorname{lm}(\mathrm{Z})=\operatorname{Im}(\mathrm{X})+(\operatorname{lm}(\mathrm{Y}) \gg\) SHIFTR \()\)
\end{tabular} \\
VRa & contents of the memory pointed to by [mem32]. VRa can not be VR5, VR4 or VR8. \\
\hline
\end{tabular}

Opcode

Description

LSW: 1110001111111000
MSW: 0000 aaaa mem32
Complex \(32+32=32\)-bit addition operation with parallel register load.
The second input operand (stored in VR3 and VR2) is shifted right by VSTATUS[SHIFR] bits before the addition. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 2.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of an overflow or underflow.
In parallel with the addition, VRa is loaded with the contents of memory pointed to by mem32.
```

// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
//
// VR5 = Re(X) VR4 = Im(X)
// VR3 = Re(Y) VR2 = Im(Y)
//
// Z = X + Y
//
if (RND == 1)
{
VR5 = VR5 + round(VR3 >> SHIFTR); // Re(Z)
VR4 = VR4 + round(VR2 >> SHIFTR); // Im(Z)
}
else
{
VR5 = VR5 + (VR3 >> SHIFTR); // Re(Z)
VR4 = VR4 + (VR2 >> SHIFTR); // Im(Z)
}
if (SAT == 1)
{
sat32(VR5);
sat32(VR4);
}
VRa = [mem32];

```

Flags

\section*{Pipeline}

\section*{Example}

See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR5 computation (real part) overflows.
- OVFI is set if the VR4 computation (imaginary part) overflows.

Both operations complete in a single cycle ( \(1 / 1\) cycles).

VCADD VR7, VR6, VR5, VR4
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR \#5-bit

InsTRUMENTS

\section*{VCADD VR7, VR6, VR5, VR4 Complex 32 + 32 = 32- Addition}

Before the operation, the inputs should be loaded into registers as shown below. Each complex number includes a 32-bit real and a 32-bit imaginary part.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR7 & 32-bit integer representing the real part of the first input: \(\operatorname{Re}(\mathrm{X})\) \\
VR6 & 32-bit integer representing the imaginary part of the first input: \(\operatorname{Im}(\mathrm{X})\) \\
VR5 & 32-bit integer representing the real part of the 2nd input: \(\operatorname{Re}(\mathrm{Y})\) \\
VR4 & 32-bit integer representing the imaginary part of the 2nd input: \(\operatorname{Im}(\mathrm{Y})\) \\
\hline
\end{tabular}

The result is also a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR7 and VR6 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR6 & \begin{tabular}{l} 
32-bit integer representing the real part of the result: \\
\(\operatorname{Re}(Z)=\operatorname{Re}(X)+(\operatorname{Re}(Y) \gg\) SHIFTR \()\)
\end{tabular} \\
VR7 & \begin{tabular}{l} 
32-bit integer representing the imaginary part of the result: \\
\(\operatorname{Im}(Z)=\operatorname{Im}(X)+(\operatorname{Im}(Y) \gg\) SHIFTR \()\)
\end{tabular} \\
\hline
\end{tabular}

\section*{Opcode}

\section*{Description}

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR7 computation (real part) overflows.
- OVFI is set if the VR6 computation (imaginary part) overflows.

Pipeline
This is a single-cycle instruction.

\section*{Example}

See also VCADD VR5, VR4, VR3, VR2
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCLROVFI
VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR \#5-bit

InsTRUMENTS

\section*{VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 Complex Conjugate Multiply and Accumulate}
```

Operands
Opcode
Description Complex Conjugate Multiply Operation
// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VR0 = X + jX: VR0[31:16] = X, VRO[15:0] = jX
// VR1 = Y + jY: VR1[31:16] = Y, VR1[15:0] = jY
//
// Perform add
//
if (RND == 1)
{
VR5 = VR5 + round(VR3 >> SHIFTR);
VR4 = VR4 + round(VR2 >> SHIFTR);
}
else
{
VR5 = VR5 + (VR3 >> SHIFTR);
VR4 = VR4 + (VR2 >> SHIFTR);
}
//
// Perform multiply (X + jX) * (Y - jY)
//
If(VSTATUS[CPACK] == 0){
VR3 = VR0H * VR1H + VROL * VR1L; Real result
VR2 = VROH * VR1L - VROL * VR1H; Imaginary result
}
else
{
VR3 = VR0L * VR1L + VR0H * VR1H; Real result
VR2 = VROL * VR1H - VR0H * VR1L; Imaginary result
}
if(SAT == 1)
{
sat32(VR3);
sat32(VR2);
}

```

Flags \(\quad\) This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Pipeline This is a \(2 p\)-cycle instruction.
See also VCLROVFI

\section*{VCLROVFR}

VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VSATON
VSATOFF

\section*{VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 : Complex Conjugate Multiply and Accumulate with Parallel Load}

\section*{Operands}
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR0 & First Complex Operand \\
VR1 & Second Complex Operand \\
VR2 & Imaginary part of the Result \\
VR3 & Real part of the Result \\
VR4 & Imaginary part of the accumulation \\
VR5 & Real part of the accumulation \\
VRa & Contents of the memory pointed to by mem32. VRa cannot be VR5, VR4 or VR8 \\
mem32 & Pointer to 32-bit memory location \\
Note: The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and \\
Imaginary-VR2) into the result registers. \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 0011 1111 0111
MSW: 0001 aaaa mem32

```

Description Complex Conjugate Multiply Operation with parallel load.
```

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VR0 = X + jX: VR0[31:16] = X, VRO[15:0] = jX
// VR1 = Y + jY: VR1[31:16] = Y, VR1[15:0] = jY
//
// Perform add
//
if (RND == 1)
{
VR5 = VR5 + round(VR3 >> SHIFTR);
VR4 = VR4 + round(VR2 >> SHIFTR);
}
else
{
VR5 = VR5 + (VR3 >> SHIFTR);
VR4 = VR4 + (VR2 >> SHIFTR);
}
//
// Perform multiply (X + jX) * (Y - jY)
//
If(VSTATUS[CPACK] == 0) {
VR3 = VR0H * VR1H + VROL * VR1L; Real result
VR2 = VR0H * VR1L - VR0L * VR1H; Imaginary result
}
else
{
VR3 = VROL * VR1L + VR0H * VR1H; Real result
VR2 = VR0L * VR1H - VR0H * VR1L; Imaginary result
}
if(SAT == 1)
{
sat32(VR3);
sat32(VR2);
}
VRa = [mem32];

```

VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 - : Complex Conjugate Multiply and Accumulate with Parallel Load
\begin{tabular}{ll} 
Flags & This instruction modifies the following bits in \\
& - OVFR is set if the VR3 computation (real \\
Pipeline & - OVFI is set if the VR2 computation (imagin \\
See also & This is a 2p-cycle instruction. \\
& VCLROVFI \\
& VCLROVFR \\
& VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 \\
& VSATON \\
& VSATOFF
\end{tabular}

\section*{VCCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ Complex Conjugate Multiply and Accumulate}

\section*{Operands}

The VMAC alternates which registers are used between each cycle. For odd cycles (1, 3,5 , and so on) the following registers are used:
\begin{tabular}{|c|c|}
\hline Odd Cycle Input & Value \\
\hline VR5 & Previous real-part total accumulation: Re(odd_sum) \\
\hline VR4 & Previous imaginary-part total accumulation: Im(odd-sum) \\
\hline VR1 & Previous real result from the multiply: Re(odd-mpy) \\
\hline VR0 & Previous imaginary result from the multiply Im(odd-mpy) \\
\hline [mem32] & Pointer to a 32-bit memory location representing the first input to the multiply
\[
\begin{aligned}
& \text { If(VSTATUS[CPACK] }==0) \\
& \text { [mem32][32:16] }=\operatorname{Re}(X) \\
& {[\text { (mem32][15:0] }=\operatorname{Im}(X)} \\
& \text { If }(\text { VSTATUS } \\
& [\text { CPPACK } 32][32: 16]==1) \\
& \text { mem32] }[15: 0]=\operatorname{Re}(X)
\end{aligned}
\] \\
\hline XAR7 & Pointer to a 32-bit memory location representing the second input to the multiply
\[
\begin{aligned}
& \text { If(VSTATUS[CPACK] }==0) \\
& \text { *XAR7[32:16] }=\operatorname{Re}(X) \\
& \text { *XAR7[15:0] }=\operatorname{Im}(X) \\
& \text { If(VSTATUS[CPACK] }==1) \\
& \text { *XAR7[32:16] }=\operatorname{Im}(X) \\
& \text { *XAR7 }[15: 0]=\operatorname{Re}(X)
\end{aligned}
\] \\
\hline
\end{tabular}

The result from the odd cycle is stored as shown below:
\begin{tabular}{|c|c|}
\hline Odd Cycle Output & Value \\
\hline VR5 & 32-bit real part of the total accumulation \(\operatorname{Re}(\) odd_sum) \(=\operatorname{Re}\) (odd_sum) \(+\operatorname{Re}\) (odd_mpy) \\
\hline VR4 & 32-bit imaginary part of the total accumulation Im(odd_sum) \(=\) Im(odd_sum) + Im(odd_mpy) \\
\hline VR1 & 32-bit real result from the multiplication:
\[
\operatorname{Re}(Z)=\operatorname{Re}(X)^{\star} \operatorname{Re}(Y)+\operatorname{Im}(X) * \operatorname{Im}(Y)
\] \\
\hline VR0 & 32-bit imaginary result from the multiplication:
\[
\operatorname{Im}(\mathrm{Z})=\operatorname{Re}(\mathrm{X})^{\star} \operatorname{Im}(\mathrm{Y})-\operatorname{Re}(\mathrm{Y})^{\star} \operatorname{Im}(\mathrm{X})
\] \\
\hline
\end{tabular}

For even cycles ( \(2,4,6\), and so on) the following registers are used:
\begin{tabular}{|c|c|}
\hline Even Cycle Input & Value \\
\hline VR7 & Previous real-part total accumulation: \(\operatorname{Re}(\) even_sum) \\
\hline VR6 & Previous imaginary-part total accumulation: Im(even-sum) \\
\hline VR3 & Previous real result from the multiply: \(\operatorname{Re}(\) even \(-m p y\) ) \\
\hline VR2 & Previous imaginary result from the multiply Im(even-mpy) \\
\hline \multirow[t]{6}{*}{[mem32]} & Pointer to a 32-bit memory location representing the first input to the multiply \\
\hline & If(VSTATUS[CPACK] == 0) \\
\hline & [mem32][32:16] \(=\operatorname{Re}(\mathrm{X})\) \\
\hline & [mem32][15:0] = Im(X) \\
\hline & If(VSTATUS[CPACK] == 1) \\
\hline & [mem32][32:16] \(=\operatorname{Im}(\mathrm{X})\) \\
\hline
\end{tabular}

\section*{Even Cycle Input Value}
\[
\text { mem32][15:0] = } \operatorname{Re}(X)
\]

XAR7 Pointer to a 32-bit memory location representing the second input to the multiply
If(VSTATUS[CPACK] == 0)
*XAR7[32:16] = Re(X)
*XAR7[15:0] \(=\operatorname{Im}(X)\)
If(VSTATUS[CPACK] ==1)
*XAR7[32:16] = Im(X)
*XAR7 [15:0] = Re(X)

The result from even cycles is stored as shown below:
\begin{tabular}{|c|c|}
\hline Even Cycle Output & Value \\
\hline VR7 & 32-bit real part of the total accumulation \(\operatorname{Re}(\) even_sum \()=\operatorname{Re}(\) even_sum \()+\operatorname{Re}(\) even_mpy \()\) \\
\hline VR6 & 32-bit imaginary part of the total accumulation \(\operatorname{Im}(\) even_sum \()=\operatorname{Im}(\) even_sum \()+\operatorname{Im}(\) even_mpy \()\) \\
\hline VR3 & 32-bit real result from the multiplication:
\[
\operatorname{Re}(Z)=\operatorname{Re}(X)^{*} \operatorname{Re}(Y)+\operatorname{Im}(X)^{*} \operatorname{Im}(Y)
\] \\
\hline VR2 & 32-bit imaginary result from the multiplication:
\[
\operatorname{Im}(\mathrm{Z})=\operatorname{Re}(\mathrm{X})^{*} \operatorname{Im}(\mathrm{Y})-\operatorname{Re}(\mathrm{Y})^{*} \operatorname{Im}(\mathrm{X})
\] \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 00100101 0001
MSW: 0010 1111 mem32

```

Perform a repeated complex conjugate multiply and accumulate operation. This instruction must be used with the single repeat instruction (RPT \|). The destination of the accumulate will alternate between VR7/VR6 and VR5/VR4 on each cycle.
```

// Cycle 1:
//
// Perform accumulate
//
if(RND == 1)
{
VR5 = VR5 + round(VR1 >> SHIFTR)
VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
VR5 = VR5 + (VR1 >> SHIFTR)
VR4 = VR4 + (VR0 >> SHIFTR)
}
//
// X and Y array element 0
//
VR1 = Re(X)*Re(Y) + Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) - Re(Y)*Im(X)
//
// Cycle 2:
//
// Perform accumulate
//
if(RND == 1)
{
VR7 = VR7 + round(VR3 >> SHIFTR)
VR6 = VR6 + round(VR2 >> SHIFTR)
}

```
```

    else
    {
    VR7 = VR7 + (VR3 >> SHIFTR)
    VR6 = VR6 + (VR2 >> SHIFTR)
    }
    //
// X and Y array element 1
//
VR3 = Re(X)*Re(Y) + Im(X)*Im(Y)
VR2 = Re(X)*Im(Y) - Re(Y)*Im(X)
//
// Cycle 3:
//
// Perform accumulate
//
if(RND == 1)
{
VR5 = VR5 + round(VR1 >> SHIFTR)
VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
VR5 = VR5 + (VR1 >> SHIFTR)
VR4 = VR4 + (VRO >> SHIFTR)
}
//
// X and Y array element 2
//
VR1 = Re(X)*Re(Y) + Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) - Re(Y)*Im(X)
etc...

```

Restrictions VR0, VR1, VR2, and VR3 will be used as temporary storage by this instruction.
Flags
The VSTATUS register flags are modified as follows:
- OVFR is set in the case of an overflow or underflow of the addition or subtraction operations.
- OVFI is set in the case an overflow or underflow of the imaginary part of the addition or subtraction operations.

\section*{Pipeline}

See also
The VCCMAC takes \(2 p+N\) cycles where \(N\) is the number of times the instruction is repeated. This instruction has the following pipeline restrictions:


\section*{VCCMPY VR3, VR2, VR1, VR0 Complex Conjugate Multiply}

Operands

\section*{Opcode}

\section*{Description}

\section*{Flags}

\section*{Pipeline}

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

This is a \(2 p\)-cycle instruction. The instruction following this one should not use VR3 or VR2.
\begin{tabular}{|c|c|c|}
\hline VCLRCPACK & & ; cpack = 0 real part in high word \\
\hline VMOV32 & VR0, *XAR4++ & ; load 1st complex input \({ }^{\text {jb }}+\mathrm{a}\) \\
\hline VMOV32 & VR1, *XAR4++ & ; load second complex input | jd + c \\
\hline VCCMPY & VR3, VR2, VR1, VR0 & \[
\begin{aligned}
& \text {; complex conjugate multiply } \\
& ;(j b+a) *(j d+c)=(a c+b d)+j(b c-a d)
\end{aligned}
\] \\
\hline NOP & & \\
\hline VMOV32 & *XAR5++, VR3 & ; store real part first \\
\hline VMOV32 & *XAR5++, VR2 & ; store imag part next \\
\hline VSETCPACK & & ; cpack \(=1\) imag part in low word \\
\hline VMOV32 & VR0, *XAR4++ & ; load 1st complex input | a + jb \\
\hline VMOV32 & VR1, *XAR4++ & ; load second complex input | c + jd \\
\hline VCCMPY & VR3, VR2, VR1, VR0 & \[
\begin{aligned}
& \text {; complex conjugate multiply } \\
& ;(a+j b) *(c+j d)=(a c+b d)+j(b c-a d)
\end{aligned}
\] \\
\hline NOP & & \\
\hline VMOV32 & *XAR5++, VR3 & ; store real part first \\
\hline VMOV32 & *XAR5++, VR2 & ; store imag part next \\
\hline
\end{tabular}

\section*{Example}

See also
VCLROVFI
VCLROVFR
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0

VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSETCPACK
VCLRCPACK
VSATON
VSATOFF

\section*{VCCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa Complex Conjugate Multiply with Parallel Store}
\begin{tabular}{|c|c|}
\hline \multirow[t]{9}{*}{Operands} & Both inputs are complex numbers with a 16 -bit real and 16-bit imaginary part. is a complex number with a 32-bit real and a 32-bit imaginary part. The result VR2 and VR3 as shown below: \\
\hline & Input Register Value \\
\hline & VR0 First Complex Operand \\
\hline & VR1 Second Complex Operand \\
\hline & VRa Value to be stored \\
\hline & VR2 Imaginary part of the Result \\
\hline & VR3 Real part of the Result \\
\hline & mem32 Pointer to 32-bit memory location \\
\hline & The result is a complex number with a 16-bit real and a 16-bit imaginary part. is stored in VR5 as shown below: \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001100000111 \\
\hline & MSW: 0001 aaaa mem32 \\
\hline \multirow[t]{7}{*}{Description} & Complex Conjugate \(16 \times 16=32\)-bit multiply operation. \\
\hline & If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, result will be saturated in the event of a 32-bit overflow or underflow. The follo operation is carried out: \\
\hline & \begin{tabular}{l}
VR3 = VROH * VR1H + VR0L * VR1L; //Re(Z) = Re(X)*Re(Y) + Im(X)*Im(Y) \\
VR2 = VROH * VR1L - VROL * VR1H; // Im(Z) = \(\operatorname{Re}(X) * \operatorname{Im}(Y)-\operatorname{Im}(X) * R e(Y)\)
\end{tabular} \\
\hline & \}else\{ \\
\hline & \multirow[t]{2}{*}{\begin{tabular}{l}
VR3 = VROL * VR1L + VR0H * VR1H; // Re(Z) \(=\operatorname{Re}(X) * \operatorname{Re}(Y)+\operatorname{Im}(X) * \operatorname{Im}(Y)\) \\
VR2 = VROL * VR1H - VR0H * VR1L; // \(\operatorname{Im}(\mathrm{Z})=\operatorname{Re}(\mathrm{X}) * \operatorname{Im}(\mathrm{Y})-\operatorname{Im}(\mathrm{X}) * \operatorname{Re}(\mathrm{Y})\)
\end{tabular}} \\
\hline & \\
\hline & \({ }_{\text {[mem32] }}\) = VRa; \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Flags & \begin{tabular}{l}
This instruction modifies the following bits in the VSTATUS register: \\
- OVFR is set if the VR3 computation (real part) overflows or unde \\
- OVFI is set if the VR2 computation (imaginary part) overflows or
\end{tabular} \\
\hline Pipeline & This is a \(2 p / 1\)-cycle instruction. The multply operation takes \(2 p\) cycl operation completes in a single cycle. The instruction following this VR3 or VR2. \\
\hline \multirow[t]{17}{*}{Example} & VCLRCPACK \(\quad\); cpack \(=0\) real part in high word \\
\hline & VMOV32 VR0, *XAR4++ ; load 1st complex input | jb + \\
\hline & VMOV32 VR1, *XAR4++ ; load second complex input | jd + c \\
\hline & VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply \\
\hline & ||VMOV32 VR0, *XAR4++ ; \((j b+a) *(j d+c)=(a c+b d)+j(b c-a d)\) \\
\hline & ; load 1st complex input | a + jb \\
\hline & NOP ; for next VCCMPY instr \\
\hline & VMOV32 *XAR5++, VR3 ; store real part first \\
\hline & VSETCPACK ; cpack = 1 imag part in low word \\
\hline & VMOV32 VR1, *XAR4++ ; load second complex input | c + jd \\
\hline & VCCMPY VR3, VR2, VR1, VR0 ; complex conjugate multiply \\
\hline & ||VMOV32 *XAR5++, VR2 ; \((\mathrm{a}+\mathrm{jb}) *(\mathrm{c}+\mathrm{jd})=(\mathrm{ac}+\mathrm{bd})+j(\mathrm{bc}-\mathrm{ad})\) \\
\hline & ; store imag part of first | \\
\hline & NOP ; VCCMPY instruction \\
\hline & VMOV32 *XAR5++, VR3 ; store real part first \\
\hline & VMOV32 *XAR5++, VR2 ; store imag part next \\
\hline & VCLRCPACK \\
\hline
\end{tabular}
www.ti.com VCCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa - Complex Conjugate Multiply with Parallel Store
See also \begin{tabular}{ll} 
VCLROVFI \\
& VCLROVFR \\
& VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 \\
& VCCMAC VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 \\
& VSETCPACK \\
& VCLRCPACK \\
& VSATON \\
& VSATOFF
\end{tabular}

\section*{VCCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 Complex Conjugate Multiply with Parallel Load}

\section*{Operands}

\section*{Opcode}

\section*{Description}

Flags

Pipeline

\section*{Example}
See also \begin{tabular}{ll} 
& VCLROVFI \\
& VCLROVFR \\
& VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 \\
& VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 \\
& VSETCPACK \\
& VCLRCPACK \\
& VSATON \\
& VSATOFF
\end{tabular}

\section*{VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 Complex Conjugate Multiply with Parallel Load}

\section*{Opcode}

\section*{Description}

Flags

Pipeline

\section*{Example}
\begin{tabular}{|c|c|}
\hline VCLRCPACK & ; cpack \(=0\) real part in high word \\
\hline VMOV32 & VR0, *XAR4++ ; load 1st complex input \({ }^{\text {jb }}+\mathrm{a}\) \\
\hline VMOV32 & VR1, *XAR4++ ; load second complex input | jd + c \\
\hline VCCMPY & VR3, VR2, VR1, VR0 ; complex conjugate multiply| \\
\hline | VMOV32 & VR0, *XAR4++ ; \((j b+a) *(j d+c)=(a c+b d)+j(b c-a d)\) \\
\hline & ; load 1st complex input | a + jb \\
\hline NOP & for next VCCMPY instr \\
\hline VMOV32 & *XAR5++, VR3 ; store real part first \\
\hline VSETCPACK & ; cpack = 1 imag part in low word \\
\hline VMOV32 & VR1, *XAR4++ ; load second complex input | c + jd \\
\hline VCCMPY & VR3, VR2, VR1, VR0 ; complex conjugate multiply| \\
\hline | VMOV32 & *XAR5++, VR2 ; \((\mathrm{a}+\mathrm{jb}) *(\mathrm{c}+\mathrm{jd})=(\mathrm{ac}+\mathrm{bd})+\mathrm{j}(\mathrm{bc}-\mathrm{ad})\) \\
\hline & ; store imag part of first \\
\hline NOP & ; VCCMPY instruction \\
\hline VMOV32 & *XAR5++, VR3 ; store real part first \\
\hline VMOV32 & *XAR5++, VR2 ; store imag part next \\
\hline VCLRCPACK & \\
\hline
\end{tabular}

\section*{See also}

VCLROVFI

VCCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSETCPACK
VCLRCPACK
VSATON
VSATOFF

\section*{VCCON VRa}

\section*{Complex Conjugate}

Operands
VRa General purpose register: VR0, VR1...VR7. Cannot be VR8.
\begin{tabular}{|c|c|}
\hline Opcode & LSW: 111000010001 aaaa \\
\hline \multirow[t]{10}{*}{Description} & if (VSTATUS[CPACK] == 0) \{ \\
\hline & if (VSTATUS[SAT] == 1) \{ \\
\hline & VRaL = sat (- VraL) \\
\hline & ```
}else {
    VRaL = - VRaL
``` \\
\hline & \} \\
\hline & \}else \{ \\
\hline & ```
if(VSTATUS[SAT] == 1){
    VRaH = sat(- VraH)
``` \\
\hline & ```
}else {
    VRaH = - VRaH
``` \\
\hline & \} \\
\hline & \} \\
\hline
\end{tabular}

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFI is set in the case an overflow or underflow of the imaginary part of the conjugate operation.

Pipeline This is a single-cycle instruction.
Example
VCCON
VR1
; VR1 := VR1^*

\section*{See also}

Instruments

\section*{VCDADD16 VR5, VR4, VR3, VR2 Complex 16 + 32 = 16 Addition}

Operands
Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16 -bit real and 16 -bit imaginary part. The second operand has a 32 -bit real and a 32 -bit imaginary part.


The result is a complex number with a 16 -bit real and a 16 -bit imaginary part. The result is stored in VR5 as shown below:


\section*{Opcode}

\section*{Description}

LSW: 1110010100000100
Complex \(16+32=16\)-bit operation. This operation is useful for algorithms similar to a complex FFT. The first operand is a complex number with a 16 -bit real and 16 -bit imaginary part. The second operand has a 32 -bit real and a 32-bit imaginary part.
Before the addition, the first input is sign extended to 32 -bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the addition is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 2.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.
```

// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

```
```

//
// Calculate Z = X + Y
//

```
```

temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)
temp1 = (temp1 << SHIFTL) + VR3; // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) + VR2; // Im(Z) intermediate
if (RND == 1)
{
temp1 = round(temp1 >> SHIFTR);
temp2 = round(temp2 >> SHIFTR);
}
else
{
temp1 = truncate(temp1 >> SHIFTR);
temp2 = truncate(temp2 >> SHIFTR);
}
if (SAT == 1)
{
VR5H = sat16(temp1);
VR5L = sat16(temp2);
}
else
{
VR5H = temp1[15:0];
VR5L = temp2[15:0];
}

```

Flags

\section*{Pipeline}

\section*{Example}

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the real-part computation (VR5H) overflows or underflows.
- OVFI is set if the imaginary-part computation (VR5L) overflows or underflows.

This is a single-cycle instruction.
```

;
; Example: Z = X + Y
X = 4 + 3j (16-bit real + 16-bit imaginary)
Y = 13 + 12j (32-bit real + 32-bit imaginary)
Real:
temp1 = 0x00000004 + 0x0000000D = 0\times00000011
VR5H = temp1[15:0] = 0x0011 = 17
Imaginary:
temp2 = 0x00000003 + 0x0000000C = 0x0000000F
VR5L = temp2[15:0]= 0x000F=15
VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR \#0 ; VSTATUS[SHIFTR] = 0
VSETSHL \#0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, \#13 ; VR3 = Re(Y) = 13
VMOVXI VR2, \#12 ; VR2 = Im(Y) = 12
VMOVXI VR4, \#3
VMOVIX VR4, \#4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x0011000F = 17 + 15j

```

The next example illustrates the operation with a right shift value defined.
;
Example: \(\mathrm{Z}=\mathrm{X}+\mathrm{Y}\) with Right Shift
```

X = 4 + 3j (16-bit real + 16-bit imaginary)
Y = 13 + 12j (32-bit real + 32-bit imaginary)
Real:
temp1 = (0x00000004 + 0x0000000D ) >> 1
temp1 = (0x00000011) >> 1 = 0x0000008.8
VR5H = temp1[15:0] = 0x0008=8
Imaginary:
temp2 = (0x00000003 + 0x0000000C ) >> 1
temp2 = (0x0000000F) >> 1 = 0x0000007.8
VR5L = temp2[15:0] = 0x0007 = 7
VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR \#1 ; VSTATUS[SHIFTR] = 1
VSETSHL \#0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, \#13 ; VR3 = Re(Y) = 13
VMOVXI VR2, \#12 ; VR2 = Im(Y) = 12
VMOVXI VR4, \#3
VMOVIX VR4, \#4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00080007 = 8 + 7j

```

The next example illustrates the operation with a right shift value defined as well as rounding.
```

Example: Z = X + Y with Right Shift and Rounding
X = 4 + 3j (16-bit real + 16-bit imaginary)
Y = 13 + 12j (32-bit real + 32-bit imaginary)
Real:
temp1 = round((0x00000004 + 0x0000000D ) >> 1)
temp1 = round(0x00000011 >> 1)
temp1 = round (0\times0000008.8) = 0\times00000009
VR5H = temp1[15:0] = 0x0011 = 8
Imaginary:
temp2 = round (0x00000003 + 0x0000000C ) >> 1)
temp2 = round(0x0000000F >> 1)
temp2 = round (0\times0000007.8) = 0x00000008
VR5L = temp2[15:0] = 0x0008=8
VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR \#1 ; VSTATUS[SHIFTR] = 1
VSETSHL \#0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, \#13 ; VR3 = Re(Y) = 13
VMOVXI VR2, \#12 ; VR2 = Im(Y) = 12
VMOVXI
VMOVIX VR4, \#4 ; VR4 = X = 0x00040003 = 4 + 3j
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0x00090008 = 9 + 8j

```

The next example illustrates the operation with both a right and left shift value defined along with rounding.
```

Example: Z = X + Y with Right Shift, Left Shift and Rounding
X = -4 + 3j (16-bit real + 16-bit imaginary)
Y = 13 - 9j (32-bit real + 32-bit imaginary)
Real:
temp1 = 0xFFFFFFFC << 2 + 0x0000000D
temp1 = 0xFFFFFFF0 + 0x0000000D = 0xFFFFFFFD
temp1 = 0xFFFFFFFD >> 1 = 0xFFFFFFFE.8

```
```

temp1 $=$ round (0xFFFFFFFFE. 8$)=0 \times \mathrm{xFFFFFFF}$
VR5H = temp1[15:0] 0xFFFF = -1 ;
Imaginary:
temp2 $=0 \times 00000003 \ll 2+0 \times 5 F F F F F F 7$
temp2 $=0 \times 0000000 \mathrm{C}+0 \times F F F F F F F 7=0 \times 00000003$
temp2 $=0 \times 00000003 \gg 1=0 \times 00000001.8$
temp1 $=$ round $(0 \times 000000001.8=0 \times 000000002$
VR5L $=$ temp2[15:0] $0 \times 0002=2$
VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] $=1$
VSETSHR \#1 ; VSTATUS[SHIFTR] = 1
VSETSHL \#2 ; VSTATUS[SHIFTL] = 2
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, \#13 ; VR3 $=\operatorname{Re}(\mathrm{Y})=13=0 \times 0000000 \mathrm{D}$
VMOVXI VR2, \#-9 $\quad$ VR2 $=\operatorname{Im}(\mathrm{Y})=-9$
VMOVIX VR2, \#0xFFFF ; sign extend VR2 = 0xFFFFFFF7
VMOVXI VR4, \#3
VMOVIX VR4, \#-4 ; VR4 $=\mathrm{X}=0 \times \mathrm{xFFFC0003}=-4+3 j$
VCDADD16 VR5, VR4, VR3, VR2 ; VR5 = Z = 0xFFFF0002 = -1 + 2 j

```

\section*{See also}

VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL \#5-bit
VSETSHR \#5-bit

InsTRUMENTS

\section*{VCDADD16 VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 Complex Double Add with Parallel Load}

Operands

Opcode

Description Complex \(16+32=16\)-bit operation with parallel register load. This operation is useful for algorithms similar to a complex FFT.

The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.
Before the addition, the first input is sign extended to 32 -bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the addition is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 2.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.
// RND is VSTATUS[RND]
```

// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit

```
```

    temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
    ```
    temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
    temp2 = sign_extend(VR4L); // 32-bit extended Im(X)
    temp2 = sign_extend(VR4L); // 32-bit extended Im(X)
    temp1 = (temp1 << SHIFTL) + VR3; // Re(Z) intermediate
    temp1 = (temp1 << SHIFTL) + VR3; // Re(Z) intermediate
    temp2 = (temp2 << SHIFTL) + VR2; // Im(Z) intermediate
    temp2 = (temp2 << SHIFTL) + VR2; // Im(Z) intermediate
    if (RND == 1)
    if (RND == 1)
    {
    {
        temp1 = round(temp1 >> SHIFTR);
        temp1 = round(temp1 >> SHIFTR);
        temp2 = round(temp2 >> SHIFTR);
        temp2 = round(temp2 >> SHIFTR);
}
}
    else
    else
    {
    {
        temp1 = truncate(temp1 >> SHIFTR);
        temp1 = truncate(temp1 >> SHIFTR);
        temp2 = truncate(temp2 >> SHIFTR);
        temp2 = truncate(temp2 >> SHIFTR);
    }
    }
    if (SAT == 1)
    if (SAT == 1)
{
{
    VR5H = sat16(temp1);
    VR5H = sat16(temp1);
    VR5L = sat16(temp2);
    VR5L = sat16(temp2);
}
}
    else
    else
    {
    {
    VR5H = temp1[15:0];
    VR5H = temp1[15:0];
    VR5L = temp2[15:0];
    VR5L = temp2[15:0];
}
}
    VRa = [mem32];
```

    VRa = [mem32];
    ```

Flags

\section*{Pipeline}

\section*{Example}

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the real-part (VR5H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR5L) computation overflows or underflows.

Both operations complete in a single cycle.
For more information regarding the addition operation, see the examples for the VCDADD16 VR5, VR4, VR3, VR2 instruction.
;
; Example: Right Shift, Left Shift and Rounding
;
\(\mathrm{X}=-4+3 j \quad(16-b i t\) real +16 -bit imaginary)
\(Y=13-9 j \quad\) (32-bit real +32 -bit imaginary)

Real:
temp1 \(=0 \times\) xFFFFFFFC \(\ll 2+0 \times 0000000 \mathrm{D}\)
temp \(1=0 \times\) PFFFFFF \(0 \quad+0 \times 0000000 \mathrm{D}=0 \times F F F F F F F D\)
temp1 \(=0 \times 5 F F F F F F D \gg 1=0 \times F F F F F F F E .8\)
temp1 \(=\) round (0xFFFFFFFFE. 8) \(=0 \times\) FFFFFFFF
VR5H = temp1[15:0] 0xFFFF \(=-1\);
Imaginary:
temp2 \(=0 \times 00000003 \ll 2+0 \times F F F F F F F 7\)
temp2 \(=0 \times 0000000 \mathrm{C} \quad+0 \times F F F F F F F 7=0 \times 00000003\)
temp2 \(=0 \times 00000003 \gg 1=0 \times 00000001.8\)
temp1 \(=\) round \((0 \times 000000001.8=0 \times 000000002\)
VR5L \(=\) temp2[15:0] \(0 \times 0002=2\)
\begin{tabular}{|c|c|c|c|}
\hline www.ti.com & VCDADD16 VR5, VR4 & 4, VR3, VR2 || VMOV32 & VRa, mem32 - Complex Double Add w \\
\hline \multirow[t]{13}{*}{} & ; & & \\
\hline & VSATOFF & & ; VSTATUS[SAT] = 0 \\
\hline & VRNDON & & ; VSTATUS[RND] = 1 \\
\hline & vSETSHR & \#1 & ; VSTATUS[SHIFTR] = 1 \\
\hline & VSETSHL & \#2 & ; VSTATUS[SHIFTL] = 2 \\
\hline & VCLEARALL & & ; VR0, VR1...VR8 == 0 \\
\hline & vmovxi & VR3, \#13 & ; VR3 \(=\operatorname{Re}(\mathrm{Y})=13=0 \times 0000000 \mathrm{D}\) \\
\hline & vmovxi & VR2, \#-9 & ; VR2 \(=\operatorname{Im}(\mathrm{Y})=-9\) \\
\hline & vmovix & VR2, \#0xFFFF & ; sign extend VR2 \(=0 \times \mathrm{FFFFFFF} 7\) \\
\hline & vmovxi & VR4, \#3 & \\
\hline & vmovix & VR4, \#-4 & ; VR4 = \(\mathrm{X}=0 \times \mathrm{xFFFCO} 003=-4+3 \mathrm{j}\) \\
\hline & VCDADD16 & VR5, VR4, VR3, VR2 & ; VR5 = z = 0xFFFF0002 = -1 + 2 j \\
\hline & || vcmov32 & VR2, *XAR7 & ; VR2 = value pointed to by XAR7 \\
\hline
\end{tabular}

See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL \#5-bit
VSETSHR \#5-bit

\section*{VCDSUB16 VR6, VR4, VR3, VR2 Complex 16-32 = 16 Subtract}

Operands

\section*{Opcode}

Description

Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16 -bit real and 16 -bit imaginary part. The second operand has a 32 -bit real and a 32 -bit imaginary part.
\begin{tabular}{|c|c|}
\hline Input Register & Value \\
\hline \multirow[t]{5}{*}{VR4H} & 16-bit integer: \\
\hline & if (VSTATUS [CPACK] = = ) \\
\hline & \(\operatorname{Re}(\mathrm{X})\) \\
\hline & else \\
\hline & Im (X) \\
\hline \multirow[t]{5}{*}{VR4L} & 16-bit integer: \\
\hline & if VSTATUS[CPACK]==0) \\
\hline & \(\operatorname{Im}(\mathrm{X})\) \\
\hline & else \\
\hline & \(\operatorname{Re}(\mathrm{X})\) \\
\hline VR3 & 32-bit integer representing the real part of the 2nd input: \(\operatorname{Re}(Y)\) \\
\hline VR2 & 32-bit integer representing the imaginary part of the 2nd input: \(\operatorname{Im}(\mathrm{Y})\) \\
\hline
\end{tabular}

The result is a complex number with a 16 -bit real and a 16-bit imaginary part. The result is stored in VR6 as shown below:


Complex 16-32=16-bit operation. This operation is useful for algorithms similar to a complex FFT.
The first operand is a complex number with a 16 -bit real and 16 -bit imaginary part. The second operand has a 32 -bit real and a 32 -bit imaginary part.
Before the addition, the first input is sign extended to 32 -bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the subtraction is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 2.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.
```

// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]
// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit

```
```

// VR2 = Im(Y) 32-bit
temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)
temp1 = (temp1 << SHIFTL) - VR3; // Re(Z) intermediate
temp2 = (temp2 << SHIFTL) - VR2; // Im(Z) intermediate
if (RND == 1)
{
temp1 = round(temp1 >> SHIFTR);
temp2 = round(temp2 >> SHIFTR);
}
else
{
temp1 = truncate(temp1 >> SHIFTR);
temp2 = truncate(temp2 >> SHIFTR);
}
if (SAT == 1)
{
VR5H = sat16(temp1);
VR5L = sat16(temp2);
}
else
{
VR5H = temp1[15:0];
VR5L = temp2[15:0];
}

```

Flags

\section*{Pipeline}

\section*{Example}

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the real-part (VR6H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR6L) computation overflows or underflows.

This is a single-cycle instruction.
```

;
Example: Z = X - Y
X = 4 + 6j (16-bit real + 16-bit imaginary)
Y = 13 + 22j (32-bit real + 32-bit imaginary)
z=(4-13)+(6-22)j=-9 - 16j
VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR \#0 ; VSTATUS[SHIFTR] = 0
VSETSHL \#0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 = 0
VMOVXI VR3, \#13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, \#22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, \#6
VMOVIX VR4, \#4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = z = 0xFFF7FFF0 = -9 + -16j

```

The next example illustrates the operation with a right shift value defined.
```

Example: Z = X - Y with Right Shift
Y = 4 + 6j (16-bit real + 16-bit imaginary)
X = 13 + 22j (32-bit real + 32-bit imaginary)
Real:
temp1 = (0x00000004-0x0000000D) >> 1
temp1 = (0xFFFFFFF7) >> 1

```
```

    temp1 = 0xFFFFFFFFB
    VR5H = temp1[15:0] = 0xFFFB = -5
    Imaginary:
temp2 = (0x00000006 - 0x00000016) >> 1
temp2 = (0xFFFFFFF0) >> 1
temp2 = 0xFFFFFFF8
VR5L = temp2[15:0] = 0xFFF8 = -8
VSATOFF ; VSTATUS[SAT] = 0
VRNDOFF ; VSTATUS[RND] = 0
VSETSHR \#1 ; VSTATUS[SHIFTR] = 1
VSETSHL \#0 ; VSTATUS[SHIFTL] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VR3, \#13 ; VR3 = Re(Y) = 13 = 0x0000000D
VMOVXI VR2, \#22 ; VR2 = Im(Y) = 22j = 0x00000016
VMOVXI VR4, \#6 , VR4 = X = 0x00040006 = 4 + 6j
VMOVIX VR4, \#4 ; VR4 = X = 0x00040006 = 4 + 6j
VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 = Z = 0xFFFBFFF8 = -5 + -8j

```

The next example illustrates rounding with a right shift value defined.
```

Example: Z = X-Y with Rounding and Right Shift

```
\(X=4+6 j \quad(16\)-bit real +16 -bit imaginary)
\(Y=-13+22 j \quad\) (32-bit real +32 -bit imaginary)
Real:
    temp1 \(=\) round \(((0 \times 00000004-0 \times F F F F F F F 3) \gg 1)\)
    temp1 \(=\) round \((0 \times 00000011) \gg 1)\)
    temp1 \(=\) round \((0 \times 000000008.8)=0 \times 000000009\)
    VR5H = temp1[15:0] \(=0 \times 0009=9\)
Imaginary:
    temp2 \(=\) round \(((0 \times 00000006-0 \times 00000016) \gg 1)\)
    temp2 \(=\) round ( \(0 \times \mathrm{xFFFFFF} 0\) ) >> 1)
    temp2 \(=\) round (0xFFFFFFF8.0) \(=0 \times\) PFFFFFFF8
    VR5L \(=\) temp2[15:0] \(=0 \times F F F 8=-8\)
    VSATOFF ; VSTATUS[SAT] \(=0\)
    VRNDON ; VSTATUS[RND] = 1
    VSETSHR \#1 ; VSTATUS[SHIFTR] = 1
    VSETSHL \#0 ; VSTATUS[SHIFTL] = 0
    VCLEARALL ; VR0, VR1...VR8 == 0
    VMOVXI VR3, \#-13 ; VR3 = \(\operatorname{Re}(\mathrm{Y})\)
    VMOVIX VR3, \#0xFFFF ; sign extend VR3 \(=-13=0 \times F F F F F F F 3\)
    VMOVXI VR2, \#22 ; VR2 \(=\operatorname{Im}(Y)=22 j=0 \times 00000016\)
    VMOVXI VR4, \#6
    VMOVIX VR4, \#4 ; VR4 \(=\mathrm{X}=0 \times 00040006=4+6 j\)
    VCDSUB16 VR6, VR4, VR3, VR2 ; VR5 \(=\mathrm{Z}=0 \times 0009 \mathrm{FFF} 8=9+-8 \mathrm{j}\)

The next example illustrates rounding with both a left and a right shift value defined.
```

Example: Z = X-Y with Rounding and both Left and Right Shift
X = 4 + 6j (16-bit real + 16-bit imaginary)
Y = -13 + 22j (32-bit real + 32-bit imaginary)
Real:
temp1 = round((0x00000004 << 2 - 0xFFFFFFF3) >> 1)
temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
temp1 = round( 0x0000001D >> 1)
temp1 = round( 0x0000000E.8) = 0x0000000F
VR5H = temp1[15:0] = 0x000F = 15
Imaginary:
temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)

```


\section*{VCDSUB16 VR6, VR4, VR3, VR2 || VMOV32 VRa, mem32 Complex 16-32 = 16 Subtract with Parallel Load}

\section*{Operands}

Before the operation, the inputs should be loaded into registers as shown below. The first operand is a complex number with a 16 -bit real and 16 -bit imaginary part. The second operand has a 32 -bit real and a 32-bit imaginary part.
\begin{tabular}{|c|c|}
\hline Input Register & Value \\
\hline \multirow[t]{5}{*}{VR4H} & 16-bit integer: \\
\hline & if (VSTATUS [CPACK] = = ) \\
\hline & \(\mathrm{Re}(\mathrm{X})\) \\
\hline & else \\
\hline & \(\operatorname{Im}(\mathrm{X})\) \\
\hline \multirow[t]{5}{*}{VR4L} & 16-bit integer: \\
\hline & if (VSTATUS [CPACK] = = ) \\
\hline & \(\operatorname{Im}(\mathrm{X})\) \\
\hline & else \\
\hline & \(\operatorname{Re}\) (X) \\
\hline VR3 & 32-bit integer representing the real part of the 2nd input: \(\operatorname{Re}(Y)\) \\
\hline VR2 & 32 -bit integer representing the imaginary part of the 2nd input: \(\operatorname{Im}(\mathrm{Y})\) \\
\hline mem32 & pointer to a 32-bit memory location. \\
\hline
\end{tabular}

The result is a complex number with a 16 -bit real and a 16-bit imaginary part. The result is stored in VR6 as shown below:
\begin{tabular}{|c|c|}
\hline Output Register & Value \\
\hline \multirow[t]{4}{*}{VR6H} & 16-bit integer: \\
\hline & ```
if (VSTATUS[CPACK]==0) {
    Re(Z) = (Re(X) << SHIFTL) - (Re(Y) ) >> SHIFTR
``` \\
\hline & ```
} else {
    Im(Z) = (Im(X) << SHIFTL) - (Im(Y) ) >> SHIFTR
``` \\
\hline & \} \\
\hline \multirow[t]{4}{*}{VR6L} & 16-bit integer: \\
\hline & ```
if(VSTATUS[CPACK]==0) {
    Im(Z) = (Im(X) << SHIFTL) - (Im(Y)) >> SHIFTR
``` \\
\hline & ```
} else {
    Re(Z) = (Re(X) << SHIFTL) - (Re(Y)) >> SHIFTR
``` \\
\hline & \} \\
\hline VRa & Contents of the memory pointed to by [mem32]. VRa cannot be VR6 or VR8. \\
\hline
\end{tabular}

Opcode
LSW: 1110001111111011
MSW: 0000 aaaa mem32
Description
Complex 16-32=16-bit operation with parallel load. This operation is useful for algorithms similar to a complex FFT.
The first operand is a complex number with a 16-bit real and 16-bit imaginary part. The second operand has a 32-bit real and a 32-bit imaginary part.
Before the addition, the first input is sign extended to 32 -bits and shifted left by VSTATUS[VSHIFTL] bits. The result of the subtraction is left shifted by VSTATUS[VSHIFTR] before it is stored in VR5H and VR5L. If VSTATUS[RND] is set, then bits shifted out to the right are rounded, otherwise these bits are truncated. The rounding operation is described in Section 2.3.2. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 16-bit overflow or underflow.
// RND is VSTATUS[RND]
// SAT is VSTATUS[SAT]
// SHIFTR is VSTATUS[SHIFTR]

InsTruments
www.ti.com VCDSUB16 VR6, VR4, VR3, VR2 || VMOV32 VRa, mem32 - Complex 16-32 = 16 Subtract with Parallel Load
```

// SHIFTL is VSTATUS[SHIFTL]
//
// VSTATUS[CPACK] = 0
// VR4H = Re(X) 16-bit
// VR4L = Im(X) 16-bit
// VR3 = Re(Y) 32-bit
// VR2 = Im(Y) 32-bit
temp1 = sign_extend(VR4H); // 32-bit extended Re(X)
temp2 = sign_extend(VR4L); // 32-bit extended Im(X)
if (RND == 1)
{
temp1 = round(temp1 >> SHIFTR);
temp2 = round(temp2 >> SHIFTR);
}
else
{
temp1 = truncate(temp1 >> SHIFTR);
temp2 = truncate(temp2 >> SHIFTR);
}
if (SAT == 1)
{
VR5H = sat16(temp1);
VR5L = sat16(temp2);
}
else
{
VR5H = temp1[15:0];
VR5L = temp2[15:0];
}
VRa = [mem32];

```

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the real-part (VR6H) computation overflows or underflows.
- OVFI is set if the imaginary-part (VR6I) computation overflows or underflows.

\section*{Pipeline}

Both operations complete in a single cycle.

\section*{Example}

For more information regarding the subtraction operation, please refer to VCDSUB16 VR6, VR4, VR3, VR2.
```

Example: Z = X-Y with Rounding and both Left and Right Shift
X = 4 + 6j (16-bit real + 16-bit imaginary)
Y = -13 + 22j (32-bit real + 32-bit imaginary)
Real:
temp1 = round((0x00000004 << 2 - 0xFFFFFFF3) >> 1)
temp1 = round((0x00000010 - 0xFFFFFFF3) >> 1)
temp1 = round( 0x0000001D >> 1)
temp1 = round( 0x0000000E.8) = 0x0000000F
VR5H = temp1[15:0] = 0x000F = 15
Imaginary:
temp2 = round((0x00000006 << 2 - 0x00000016) >> 1)
temp2 = round((0x00000018 - 0x00000016) >> 1)
temp2 = round( 0x00000002 >> 1)
temp1 = round( 0x00000001.0) = 0x00000001
VR5L = temp2[15:0] = 0x0001 = 1
VSATOFF ; VSTATUS[SAT] = 0
VRNDON ; VSTATUS[RND] = 1
VSETSHR \#1 ; VSTATUS[SHIFTR] = 1
VSETSHL \#2 ; VSTATUS[SHIFTL] = 2

```

VCDSUB16 VR6, VR4, VR3, VR2 || VMOV32 VRa, mem32 - Complex 16-32 = 16 Subtract with Parallel Load www.ti.com


See also
VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32
VCADD VR7, VR6, VR5, VR4
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHL \#5-bit VSETSHR \#5-bit

\section*{VCFLIP VRa}

Operands
VRa General purpose register: VR0, VR1....VR7. Cannot be VR8.
Opcode LSW: 101000010000 aaaa
Description Swap VRaL and VRaH

Flags This instruction does not affect any flags in the VSTATUS register
Pipeline
This is a single-cycle instruction.
Example
```

VCFLIP VR7 ; VR7H := VR7L | VR7L := VR7H

```

See also

\section*{VCMAC VR5, VR4, VR3, VR2, VR1, VR0 Complex Multiply and Accumulate}

\section*{Operands}

\section*{Opcode}

\section*{Description}

Flags

\section*{Pipeline}
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR5 & Real part of the accumulation \\
VR4 & Imaginary part of the accumulation \\
VR3 & Real part of the product \\
VR2 & Imaginary part of the product \\
VR1 & Second Complex Operand \\
VR0 & First Complex Operand \\
\hline
\end{tabular}

NOTE: The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.
```

LSW: 1110 0101 0000 0001

```

Complex multiply operation.
```

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VR0 = X + jX: VR0[31:16] = X, VR0[15:0] = jX
// VR1 = Y + jY: VR1[31:16] = Y, VR1[15:0] = jY
//
// Perform add
//
if (RND == 1)
{
VR5 = VR5 + round(VR3 >> SHIFTR);
VR4 = VR4 + round(VR2 >> SHIFTR);
}
else
{
VR5 = VR5 + (VR3 >> SHIFTR);
VR4 = VR4 + (VR2 >> SHIFTR);
}
//
// Perform multiply (X + jX) * (Y + jY)
//
if(VSTATUS[CPACK] == 0) {
VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}else{
VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}
if(SAT == 1)
{
sat32(VR3);
sat32(VR2);
}

```

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

Example
See also \(\quad\)\begin{tabular}{ll} 
& VCLROVFI \\
& VCLROVFR \\
& VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 \\
& VSATON \\
& VSATOFF
\end{tabular}

\section*{VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ Complex Multiply and Accumulate}

Operands The VMAC alternates which registers are used between each cycle. For odd cycles (1, 3,5 , and so on) the following registers are used:
\begin{tabular}{|c|c|}
\hline Odd Cycle Input & Value \\
\hline VR5 & Previous real-part total accumulation: Re(odd_sum) \\
\hline VR4 & Previous imaginary-part total accumulation: Im(odd-sum) \\
\hline VR1 & Previous real result from the multiply: Re(odd-mpy) \\
\hline VR0 & Previous imaginary result from the multiply Im(odd-mpy) \\
\hline [mem32] & Pointer to a 32-bit memory location representing the first input to the multiply
\[
\begin{aligned}
& \text { If }(\text { VSTATUS[CPACK }]==0) \\
& {[\text { mem32][32:16] } \operatorname{Re}(X)} \\
& \text { [mem32][15:0] }=\operatorname{Im}(X) \\
& \text { If }(\mathrm{VSTATUS[CPACK]}==1) \\
& {[\text { mem32][32:16] }=\operatorname{Im}(X)} \\
& \text { mem32][15:0] }=\operatorname{Re}(X)
\end{aligned}
\] \\
\hline XAR7 & Pointer to a 32-bit memory location representing the second input to the multiply
\[
\begin{aligned}
& \text { If(VSTATUS[CPACK] }==0) \\
& \text { *XAR7[32:16] }=\operatorname{Re}(X) \\
& \text { *XAR7[15:0] }=\operatorname{Im}(X) \\
& \text { If(VSTATUS[CPACK] }==1) \\
& \text { *XAR7[32:16] }=\operatorname{Im}(X) \\
& \text { *XAR7 }[15: 0]=\operatorname{Re}(X)
\end{aligned}
\] \\
\hline
\end{tabular}

The result from odd cycle is stored as shown below:
\begin{tabular}{|c|c|}
\hline Odd Cycle Output & Value \\
\hline VR5 & 32-bit real part of the total accumulation \(\operatorname{Re}(\) odd_sum \()=\operatorname{Re}\left(o d d \_s u m\right)+\operatorname{Re}\left(o d d \_m p y\right)\) \\
\hline VR4 & 32-bit imaginary part of the total accumulation \(\operatorname{Im}(\) odd_sum \()=\operatorname{Im}\) (odd_sum) \(+\operatorname{Im}\) (odd_mpy) \\
\hline VR1 & 32-bit real result from the multiplication:
\[
\operatorname{Re}(Z)=\operatorname{Re}(X)^{*} \operatorname{Re}(Y)-\operatorname{Im}(X)^{*} \operatorname{Im}(Y)
\] \\
\hline VRO & 32-bit imaginary result from the multiplication:
\[
\operatorname{Im}(\mathrm{Z})=\operatorname{Re}(\mathrm{X})^{*} \operatorname{Im}(\mathrm{Y})+\operatorname{Re}(\mathrm{Y})^{*} \operatorname{Im}(\mathrm{X})
\] \\
\hline
\end{tabular}

For even cycles (2, 4, 6, and so on) the following registers are used:
\begin{tabular}{|c|c|}
\hline Even Cycle Input & Value \\
\hline VR7 & Previous real-part total accumulation: Re (even_sum) \\
\hline VR6 & Previous imaginary-part total accumulation: Im(even-sum) \\
\hline VR3 & Previous real result from the multiply: Re (even-mpy) \\
\hline VR2 & Previous imaginary result from the multiply Im(even-mpy) \\
\hline [mem32] & Pointer to a 32-bit memory location representing the first input to the multiply If(VSTATUS[CPACK] == 0) \\
\hline & [mem32][32:16] \(=\operatorname{Re}(X)\) \\
\hline & [mem32][15:0] = Im(X) \\
\hline & If(VSTATUS[CPACK] == 1 ) \\
\hline & [mem32][32:16] \(=\operatorname{Im}(\mathrm{X})\) \\
\hline
\end{tabular}

InSTRUMENTS

VCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++ - Complex Multiply and Accumulate

\section*{Even Cycle Input Value}
\[
\text { mem32][15:0] }=\operatorname{Re}(X)
\]
\begin{tabular}{|c|c|}
\hline XAR7 & Pointer to a 32-bit memory location representing the second input to the multiply \\
\hline & \(\mathrm{If}(\mathrm{VSTATUS}[\mathrm{CPACK}]==0)\) \\
\hline & *XAR7[32:16] = Re(X) \\
\hline & *XAR7[15:0] = Im( X ) \\
\hline & If(VSTATUS[CPACK] == 1) \\
\hline & *XAR7[32:16] \(=\operatorname{Im}(\mathrm{X})\) \\
\hline & *XAR7 [15:0] \(=\operatorname{Re}(\mathrm{X})\) \\
\hline
\end{tabular}

The result from even cycles is stored as shown below:
\begin{tabular}{|c|c|}
\hline Even Cycle Output & Value \\
\hline VR7 & 32-bit real part of the total accumulation \(\operatorname{Re}(\) even_sum \()=\operatorname{Re}(\) even_sum \()+\operatorname{Re}(\) even_mpy \()\) \\
\hline VR6 & 32-bit imaginary part of the total accumulation Im(even_sum) = Im(even_sum) + Im(even_mpy) \\
\hline VR3 & 32-bit real result from the multiplication:
\[
\operatorname{Re}(\mathrm{Z})=\operatorname{Re}(\mathrm{X})^{*} \operatorname{Re}(\mathrm{Y})-\operatorname{Im}(\mathrm{X})^{*} \operatorname{Im}(\mathrm{Y})
\] \\
\hline VR2 & 32-bit imaginary result from the multiplication:
\[
\operatorname{Im}(Z)=\operatorname{Re}(X)^{*} \operatorname{lm}(Y)+\operatorname{Re}(Y)^{*} \operatorname{Im}(X)
\] \\
\hline
\end{tabular}

\section*{Opcode}

\section*{Description}
```

LSW: 1110 0010 0101 0001
MSW: 0000 0000 mem32

```

Perform a repeated multiply and accumulate operation. This instruction must be used with the repeat instruction (RPT\|). The destination of the accumulate will alternate between VR7/VR6 and VR5/VR4 on each cycle.
```

// Cycle 1:
//
// Perform accumulate
//
if(RND == 1)
{
VR5 = VR5 + round(VR1 >> SHIFTR)
VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
VR5 = VR5 + (VR1 >> SHIFTR)
VR4 = VR4 + (VR0 >> SHIFTR)
}
//
// X and Y array element 0
//
VR1 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) + Re(Y)*Im(X)
//
// Cycle 2:
//
// Perform accumulate
//
if(RND == 1)
{
VR7 = VR7 + round(VR3 >> SHIFTR)
VR6 = VR6 + round(VR2 >> SHIFTR)
}
else
{

```
```

        VR7 = VR7 + (VR3 >> SHIFTR)
        VR6 = VR6 + (VR2 >> SHIFTR)
    }
    //
    // X and Y array element 1
    //
    VR3 = Re(X)*Re(Y) - Im(X)*Im(Y)
    VR2 = Re(X)*Im(Y) + Re(Y)*Im(X)
    //
// Cycle 3:
//
// Perform accumulate
//
if(RND == 1)
{
VR5 = VR5 + round(VR1 >> SHIFTR)
VR4 = VR4 + round(VR0 >> SHIFTR)
}
else
{
VR5 = VR5 + (VR1 >> SHIFTR)
VR4 = VR4 + (VR0 >> SHIFTR)
}
//
// X and Y array element 2
//
VR1 = Re(X)*Re(Y) - Im(X)*Im(Y)
VR0 = Re(X)*Im(Y) + Re(Y)*Im(X)
etc...

```

\section*{Restrictions}

Flags

\section*{Pipeline}

\section*{Example}

VR0, VR1, VR2, and VR3 will be used as temporary storage by this instruction.
The VSTATUS register flags are modified as follows:
- OVFR is set in the case of an overflow or underflow of the addition or subtraction operations.
- OVFI is set in the case an overflow or underflow of the imaginary part of the addition or subtraction operations.

The VCCMAC takes \(2 p+N\) cycles where \(N\) is the number of times the instruction is repeated. This instruction has the following pipeline restrictions:
```

<<instruction1>> ; No restrictions
<<instruction2>> ; Cannot be a 2p instruction that writes
; to VR0, VR1...VR7 registers
; Execute N times, where N is even
RPT \#(N-1) ; Execute N times,
|| VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
; Can read VR0, VR1...VR8

```

Cascading of RPT || VCMAC 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:
```

Example of cascaded VMAC instructions

```
    VCLEARALL ; Zero the accumulation registers
Execute MACF32 N+1 (4) times
    RPT \#3
|| VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
;
Execute MACF32 N+1 (6) times
|| VCMAC VR7, VR6, VR5, VR4, *XAR6++, *XAR7++
;
; Repeat MACF32 N+1 times where \(N+1\) is even
;
    RPT \#N
|| MACF32 R7H, R3H, *XAR6++, *XAR7++
    ADDF32 VR7, VR6, VR5, VR4

See also VCCMAC VR7, VR6, VR5, VR4, mem32, *XAR7++
\begin{tabular}{lll}
\hline VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 Complex Multiply and Accumulate \\
with Parallel Load
\end{tabular}

NOTE: The user will need to do one final addition to accumulate the final multiplications (Real-VR3 and Imaginary-VR2) into the result registers.

\section*{Opcode}

\section*{Description}
```

LSW: 1110 0011 1111 0111
MSW: 0000 aaaa mem32

```

Complex multiply operation.
```

// VR5 = Accumulation of the real part
// VR4 = Accumulation of the imaginary part
//
// VRO = X + Xj: VR0[31:16] = Re(X), VR0[15:0] = Im(X)
// VR1 = Y + Yj: VR1[31:16] = Re(Y), VR1[15:0] = Im(Y)
//
// Perform add
//
if (RND == 1)
{
VR5 = VR5 + round(VR3 >> SHIFTR);
VR4 = VR4 + round(VR2 >> SHIFTR);
}
else
{
VR5 = VR5 + (VR3 >> SHIFTR);
VR4 = VR4 + (VR2 >> SHIFTR);
}
//
// Perform multiply Z = (X + Xj) * (Y + Yj)
//
if(VSTATUS[CPACK] == 0) {
VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0H * VR1L + VROL * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}else{
VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0L * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
})
if(SAT == 1)
{
sat32(VR3);
sat32(VR2);
}
VRa = [mem32];

```

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
\begin{tabular}{ll}
\hline & - OVFIIS set it the VR2 computation (imaginary part) overtiows or undertiows. \\
Pipeline & \begin{tabular}{l} 
This is a \(2 p / 1-\) cycle instruction. The multiply and accumulate is a \(2 p-c y c l e ~ o p e r a t i o n ~ a n d ~\) \\
the VMOV32 is a single-cycle operation.
\end{tabular} \\
Example \\
See also & \\
& VCLROVFI \\
& VCLROVFR \\
& VCMAC VR5, VR4, VR3, VR2, VR1, VR0 \\
& VSATON \\
& VSATOFF
\end{tabular}

\section*{VCMAG VRb, VRa}

\section*{Magnitude of a Complex Number}

Operands
Opcode
Description

Flags

\section*{Pipeline}

Example

VRb General purpose register VR0...VR8
VRa General purpose register VR0...VR8
```

LSW: 1110 0110 1111 0010
MSW: 0000 0100 bbbb aaaa

```

Compute the magnitude of the Complex value in VRa
If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.
```

If(VSTATUS[SAT] == 1){
If(VSTATUS[RND] == 1){
VRb = rnd(sat(VRaH*VRaH + VRaL*VRaL) >>VSTATUS[SHIFTR])
}else {
VRb = sat(VRaH*VRaH + VRaL*VRaL)>>VSTATUS[SHIFTR]
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1){
VRb = rnd((VRaH*VRaH + VRaL*VRaL)>>VSTATUS[SHIFTR])
}else {
VRb = (VRaH*VRaH + VRaL*VRaL) >>VVSTATUS[SHIFTR]
}
}

```

Sign-Extension is automatically done for the shift right operations
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if overflow is detected in the complex magnitude operation of the real 32-bit result

This is a 2 cycle instruction
```

VMOV32 VR1, VR0 ; VR1 := VR0
VCCON VR1 ; VR1 := VR1^*
VCMAG VR2 , VR0 ; VR2 := magnitude(VR0)
and so forth

```

\section*{See also}

InsTRUMENTS

\section*{VCMPY VR3, VR2, VR1, VR0 Complex Multiply}

\section*{Opcode}

Description

Flags

\section*{Pipeline}

\section*{Example}

Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR3 & Real part of the Result \\
VR2 & Imaginary part of the Result \\
VR1 & Second Complex Operand \\
VR0 & First Complex Operand \\
\hline
\end{tabular}

LSW: 1110010100000000
Complex \(16 \times 16=32\)-bit multiply operation.
If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, the result will be saturated in the event of a 32-bit overflow or underflow.
```

// Calculate: Z = (X + jX) * (Y + jY)
//
if(VSTATUS[CPACK] == 0){
VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}else{
VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VROL * VR1H + VROH * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}
if(SAT == 1)
{
sat32(VR3);
sat32(VR2);
}

```

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

This is a 2 p-cycle instruction. The instruction following this one should not use VR3 or VR2.
```

Example 1
X = 4 + 6j
Y = 12 + 9j
Z = X * Y
Re(Z)=4*12-6*9 = -6
; Im(Z) = 4*9 + 6*12 = 108
VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVXI VRO,\#6
VMOVXI VR1, \#9 年
VMOVIX VR1, \#12 ; VR1 = Y = 0x000C0009 = 12 + 9j
VCMPY VR3, VR2, VR1, VR0 ; VR3 = Re(Z) = 0xFFFFFFFA = - 6
; VR2 = Im(Z) = 0x0000006C = 108
; <- Must not use VR2, VR3
; <- VCMPY completes, VR2, VR3 valid
; Can use VR2, VR3

```

\author{
See also \\ VCLROVFI \\ VCLROVFR \\ VCMAC VR5, VR4, VR3, VR2, VR1, VR0 \\ VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 \\ VSATON \\ VSATOFF
}

InsTRUMENTS
www.ti.com
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa - Complex Multiply with Parallel Store
VCMPY VR3, VR2, VR1, VR0 || VMOV32 mem32, VRa Complex Multiply with Parallel Store
Operands

Opcode

\section*{Description}
```

LSW: 1110 0010 1100 1010
MSW: 0000 aaaa mem16

```

Complex \(16 \times 16=32\)-bit multiply operation with parallel register load.
If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.
```

// Calculate: Z = (X + jX) * (Y + jY)
//
if(VSTATUS[CPACK] == 0){
VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}else{
VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VROL * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}
if(SAT == 1)
{
sat32(VR3);
sat32(VR2);
}
VRa = [mem32];

```



See also
VCLROVFI
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSATON
VSATOFF

InsTRUMENTS

\section*{VCMPY VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32 Complex Multiply with Parallel Load}

Operands

\section*{Opcode}

\section*{Description}

Both inputs are complex numbers with a 16-bit real and 16-bit imaginary part. The result is a complex number with a 32-bit real and a 32-bit imaginary part. The result is stored in VR2 and VR3 as shown below:
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR3 & Real part of the Result \\
VR2 & Imaginary part of the Result \\
VR1 & Second Complex Operand \\
VR0 & First Complex Operand \\
VRa & 32-bit value pointed to by mem32. VRa can not be VR2, VR3 or VR8. \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
```

LSW: 1110 0011 1111 0110
MSW: 0000 aaaa mem32

```

Complex \(16 \times 16=32\)-bit multiply operation with parallel register load.
If the VSTATUS[CPACK] bit is set, the low word of the input is treated as the real part while the upper word is treated as imaginary. If the VSTATUS[SAT] bit is set, then the result will be saturated in the event of a 32-bit overflow or underflow.
```

// Calculate: z = (X + jX) * (Y + jY)
//
if(VSTATUS[CPACK] == 0){
VR3 = VR0H * VR1H - VR0L * VR1L; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VR0H * VR1L + VR0L * VR1H; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}else{
VR3 = VR0L * VR1L - VR0H * VR1H; // Re(Z) = Re(X)*Re(Y) - Im(X)*Im(Y)
VR2 = VROL * VR1H + VR0H * VR1L; // Im(Z) = Re(X)*Im(Y) + Im(X)*Re(Y)
}
if(SAT == 1)
{
sat32 (VR3);
sat32 (VR2);
}
VRa = [mem32];

```

- OVFR is set if the VR3 computation (real part) overflows or underflows.
- OVFI is set if the VR2 computation (imaginary part) overflows or underflows.

This is a \(2 p / 1\)-cycle instruction. The multiply operation takes \(2 p\) cycles and the VMOV operation completes in a single cycle. The instruction following this one must not use VR2 or VR3.
```

Example 1
X = 4 + 6j
Y = 12 + 9j
Z = X * Y
Re(Z) = 4*12 - 6*9 = -6
VSATOFF ; VSTATUS[SAT] = 0
VCLEARALL ; VR0, VR1...VR8 == 0
VMOVIX VRO, \#4 ; VRO = X = 0x000400006 = 4 + 6j
VMOVXI VR1, \#9
VR1, \#12 ; VR1 = Y = 0x000C0009 = 12 + 9j
; VR3 = Re(Z) = 0xFFFFFFFA = -6

```


See also
VCLROVFI
VCLROVFR
VCMAC VR5, VR4, VR3, VR2, VR1, VR0
VCMAC VR5, VR4, VR3, VR2, VR1, VR0 || VMOV32 VRa, mem32
VSATON
VSATOFF

InsTRUMENTS

\section*{VCSHL16 VRa << \#4-bit Complex Shift Left}

\section*{Operands}

\section*{Opcode}

\section*{Description}
```

LSW: 1110 0110 1111 0010
MSW: 0000 0000 IIII aaaa

```

Left Shift the Real and Imaginary parts of the complex value in VRa.
```

if(VSTATUS[CPACK] == 0){
if(VSTATUS[SAT] == 1){
VRaL = sat(VRaL <<\#4-bit Immediate) (imaginary result)
VRaH = sat(VRaH << \#4-bit Immediate) (real result)
}else {
VRaL = VRaL << \#4-bit Immediate (imaginary result)
VRaH = VRaH << \#4-bit Immediate (real result)
}
}else {
If(VSTATUS[SAT] == 1){
VRaL = sat(VRaL << \#4-bit Immediate) (real result)
VRaH = sat(VRaH << \#4-bit Immediate) (imaginary result)
}else {
VRaL = VRaL << \#4-bit Immediate (real result)
VRaH = VRaH << \#4-bit Immediate (imaginary result)
}
}

```

Flags

Pipeline This is a single-cycle instruction.
Example \(\quad \begin{aligned} & \text { VSATOFF } \\ & \text { VCSHL16 }\end{aligned}\) VR5 \(\ll \# 8\); turn off saturation \(\quad\) VR5L \(:=\) VR5L \(\ll 8 \mid\) VR5H \(:=\) VR5H \(\ll 8\)

\section*{See also}

\section*{VCSHR16 VRa >> \#4-bit Complex Shift Right}

\section*{Operands}

\section*{Opcode}

\section*{Description}
```

if(VSTATUS[CPACK] == 0){
if(VSTATUS[RND] == 1) {
VRaL = rnd(VRaL >> \#4-bit Immediate) (imaginary result)
VRaH = rnd(VRaH >> \#4-bit Immediate) (real result)
}else {
VRaL = VRaL >> \#4-bit Immediate (imaginary result)
VRaH = VRaH >> \#4-bit Immediate (real result)
}
}else {
If(VSTATUS[RND] == 1){
VRaL = rnd(VRaL >> \#4-bit Immediate) (real result)
VRaH = rnd(VRaH >> \#4-bit Immediate) (imaginary result)
}else {
VRaL = VRaL >> \#4-bit Immediate (real result)
VRaH = VRaH >> \#4-bit Immediate (imaginary result)
}
}

```

Sign-Extension is automatically done for the shift right operations

Flags
Pipeline
Example

This instruction does not affect any flags in the VSTATUS register.
This is a single-cycle instruction.
```

VSATOFF ; turn off saturation
VCSHR16 VR6 >> \#8 ; VR6L := VR6L >> 8 | VR6H := VR6H >> 8

```

\section*{See also}

\section*{VCSUB VR5, VR4, VR3, VR2 Complex 32-32 = 32 Subtraction}

\section*{Opcode}

\section*{Description}

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if the VR5 computation (real part) overflows or underflows.
- OVFI is set if the VR6 computation (imaginary part) overflows or underflows.

Pipeline
This is a single-cycle instruction.

\section*{Example}
\begin{tabular}{ll} 
See also & VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 \\
& VCADD VR7, VR6, VR5, VR4 \\
& VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 \\
& VCLROVFI
\end{tabular}

VCLROVFR
VRNDOFF
VRNDON
VSATON
VSATOFF
VSETSHR \#5-bit

\section*{VCSUB VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 Complex Subtraction}

Opcode

Description

Flags

\section*{Pipeline}

\section*{Example}
\begin{tabular}{ll} 
See also & VCADD VR5, VR4, VR3, VR2 || VMOV32 VRa, mem32 \\
& VCADD VR7, VR6, VR5, VR4 \\
& VCSUB VR5, VR4, VR3, VR2 \\
& VCLROVFI \\
& VCLROVFR \\
& VRNDOFF \\
& VRNDON \\
& VSATON \\
& VSATOFF \\
& VSETSHR \#5-bit
\end{tabular}

Instruments

\subsection*{2.5.5 Cyclic Redundancy Check (CRC) Instructions}

The instructions are listed alphabetically, preceded by a summary.

\section*{Table 2-14. CRC Instructions}
Title Page
VCRC8H_1 mem16 -CRC8, High Byte ..... 272
VCRC8L_1 mem16 —CRC8, Low Byte ..... 273
VCRC16P1H_1 mem16 — CRC16, Polynomial 1, High Byte ..... 274
VCRC16P1L_1 mem16 — CRC16, Polynomial 1, Low Byte ..... 275
VCRC16P2H_1 mem16 — CRC16, Polynomial 2, High Byte ..... 276
VCRC16P2L_1 mem16 — CRC16, Polynomial 2, Low Byte ..... 277
VCRC24H_1 mem16 — CRC24, High Byte ..... 278
VCRC24L_1 mem16 —CRC24, Low Byte ..... 279
VCRC32H_1 mem16 —CRC32, High Byte ..... 280
VCRC32L_1 mem16 - CRC32, Low Byte ..... 281
VCRC32P2H_1 mem16 - CRC32, Polynomial 2, High Byte ..... 282
VCRC32P2L_1 mem16 —CRC32, Low Byte ..... 283
VCRCCLR —Clear CRC Result Register ..... 284
VMOV32 mem32, VCRC — Store the CRC Result Register ..... 285
VMOV32 VCRC, mem32 —Load the CRC Result Register ..... 286

\section*{VCRC8H_1 mem16 CRC8, High Byte}

Operands

Opcode

\section*{Description}

Flags
Pipeline
Example
See also
mem16 16-bit memory location
\begin{tabular}{lll} 
LSW: 1110 & 0010 & 11001100 \\
MSW: 0000 & 0000 & mem16
\end{tabular}

This instruction uses CRC8 polynomial \(==0 \times 07\).
Calculate the CRC8 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP == 0) {
temp[7:0] = [mem16][15:8];
}else {
temp[7:0] = [mem16][8:15];
}
VCRC[7:0] = CRC8 (VCRC[7:0], temp[7:0])

```

InsTRUMENTS

\section*{VCRC8L_1 mem16 CRC8, Low Byte}

\section*{Operands}

\section*{Opcode}

\section*{Description}

\section*{Flags}

Pipeline

\section*{Example}

\section*{See also VCRC8H_1 mem16}

\section*{VCRC16P1H_1 mem16 CRC16, Polynomial 1, High Byte}

Operands

Opcode

\section*{Description}

Flags
Pipeline
Example
See also
mem16 16-bit memory location
```

LSW: 1110 0010 1100 1111
MSW: 0000 0000 mem16

```

This instruction uses CRC16 polynomial \(1==0 \times 8005\).
Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP] == 0) {
temp[7:0] = [mem16][15:8];
}else {
temp[7:0] = [mem16][8:15];
}
VCRC[15:0] = CRC16(VCRC[15:0], temp[7:0])

```

InsTRUMENTS

\section*{VCRC16P1L_1 mem16 CRC16, Polynomial 1, Low Byte}


\section*{VCRC16P2H_1 mem16 CRC16, Polynomial 2, High Byte}

Operands

Opcode

\section*{Description}

Flags
Pipeline
Example
See also
mem16 \(\quad\) 16-bit memory location
```

LSW: 1110 0010 1100 1111
MSW: 0001 0000 mem16

```

This instruction uses CRC16 polynomial \(2=0 \times 1021\).
Calculate the CRC16 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP] == 0){
temp[7:0] = [mem16][15:8];
}else {
temp[7:0] = [mem16][8:15];
}
VCRC[15:0] = CRC16(VCRC[15:0], temp[7:0])

```

InsTRUMENTS

\section*{VCRC16P2L_1 mem16 CRC16, Polynomial 2, Low Byte}


\section*{VCRC24H_1 mem16 CRC24, High Byte}

Operands

Opcode

\section*{Description}

\section*{Flags}

Pipeline
Example
```

LSW: 1110001011001011
MSW: 00000010 mem16

```

This instruction uses CRC24 polynomial \(==0 \times 5\) D6DCB
Calculate the CRC24 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP] == 0){
temp[7:0] = [mem16][15:8];
}else {
temp[7:0] = [mem16][8:15];
}
VCRC[23:0] = CRC24 (VCRC[23:0], temp[7:0])

```

See also
This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VCRC24L_1 mem16.
VCRC24L_1 mem16

InsTRUMENTS

\section*{VCRC24L_1 mem16 CRC24, Low Byte}

\section*{Operands}

\section*{Opcode}

\section*{Description}

\section*{Flags}

Pipeline

\section*{Example}

\section*{See also}

\section*{VCRC32H_1 mem16 CRC32, High Byte}

Operands

Opcode

Description

Flags
Pipeline
Example
See also
mem16 16-bit memory location
\begin{tabular}{ccccc} 
LSW: & 1110 & 0010 & 1100 & 0010 \\
MSW: 0000 & 0000 & mem16
\end{tabular}

This instruction uses CRC32 polynomial \(1==0 \times 04 \mathrm{C} 11\) DB7
Calculate the CRC32 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP] == 0){
temp[7:0] = [mem16][15:8];
}else {
temp[7:0] = [mem16][8:15];
}
VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])

```

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VCRC32L_1 mem16.
VCRC32L_1 mem16

\section*{VCRC32L_1 mem16 CRC32, Low Byte}

\section*{Operands}

\section*{Opcode}

\section*{Description}

Flags

\section*{Pipeline}
```

LSW:1110 0010 1100 0001
MSW: 0000 0000 mem16

```

This instruction uses CRC32 polynomial \(1==0 \times 04 \mathrm{C} 11\) DB7
Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP] == 0){
temp[7:0] = [mem16][7:0];
}else {
temp[7:0] = [mem16][0:7];
}
VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])

```

This instruction does not modify any flags in the VSTATUS register.

\section*{Example}

\section*{See also}

VCRC32H_1 mem16

\section*{VCRC32P2H_1 mem16 CRC32, Polynomial 2, High Byte}

Operands

Opcode

\section*{Description}

This instruction uses CRC32 polynomial \(==0 \times 1\) EDC6F41
Calculate the CRC32 of the most significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC.
```

if (VSTATUS[CRCMSGFLIP] == 0) {
temp[7:0] = [mem16][15:8];
}else {
temp[7:0] = [mem16][8:15];
}
VCRC[31:0] = CRC32 (VCRC[31:0], temp[7:0])

```

Flags
Pipeline
Example
See also

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VCRC32P2L_1 mem16.
VCRC32L_1 mem16

InsTRUMENTS

\section*{VCRC32P2L_1 mem16 CRC32, Low Byte}
\begin{tabular}{|c|c|}
\hline \multirow[t]{2}{*}{Operands} & \\
\hline & mem16 16-bit memory location \\
\hline Opcode & \begin{tabular}{l}
LSW: 1110001011001011 \\
MSW: 00000011 mem16
\end{tabular} \\
\hline \multirow[t]{4}{*}{Description} & This instruction uses CRC32 polynomial \(==0 \times 04 \mathrm{C} 11 \mathrm{DB7}\) \\
\hline & Calculate the CRC32 of the least significant byte pointed to by mem16 and accumulate it with the value in the VCRC register. Store the result in VCRC. \\
\hline & ```
if (VSTATUS[CRCMSGFLIP] == 0){
    temp[7:0] = [mem16][7:0];
    }else {
        temp[7:0] = [mem16][0:7];
    }
``` \\
\hline & \(\operatorname{VCRC}[31: 0]=\operatorname{CRC} 32(\operatorname{VCRC}[31: 0]\), temp [7:0]) \\
\hline Flags & This instruction does not modify any flags in the VSTATUS register. \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{19}{*}{Example} & ```
typedef struct {
    uint32_t *CRCResult; // Address where result should be stored
    uint16_t *CRCData; // Start of data
    uint16_t CRCLen; // Length of data in bytes
    }CRC_CALC;
``` \\
\hline & CRC_CALC mycrc; \\
\hline & CRC32P2(\&mycrc); \\
\hline & . . \\
\hline & \begin{tabular}{l}
; Calculate the CRC of a block of data \\
; This function assumes the block is a multiple of 216 -bit words
\end{tabular} \\
\hline & \[
\begin{aligned}
& \text {.global _CRC32P2 } \\
& \text { _CRC32P2 }
\end{aligned}
\] \\
\hline & VCRCCLR ; Clear the result register \\
\hline & MOV AL, *+XAR4[4] ; AL = CRCLen \\
\hline & ASR AL, 2 ; AL = CRCLen/4 \\
\hline & SUBB AL, \#1 ; AL = CRCLen/4 - 1 \\
\hline & MOVL XAR7, *+XAR4[2] ; XAR7 = \&CRCData .align 2 \\
\hline & NOP ; Align RPtB to an odd address \\
\hline & RPTB _CRC32P2_done, AL ; Execute block of code AL + 1 times VCRC32P2L_1 *XAR7 ; Calculate CRC for 4 bytes \\
\hline & VCRC32P2H_1 *XAR7++ ; ... \\
\hline & VCRC32P2L_1 *XAR7 ; \\
\hline & \begin{tabular}{l}
VCRC32P2H_1 *XAR7++ ; ... \\
CRC32P2 done
\end{tabular} \\
\hline & MOVL XAR7, *_+XAR4[0] ; XAR7 = \&CRCResult \\
\hline & VMOV32 *+XAR7[0], VCRC ; Store the result \\
\hline & LRETR ; return to caller \\
\hline See also & VCRC32P2H_1 mem16 \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|}
\hline \multicolumn{2}{|l|}{vCRCCLR - Clear CRC Result Register} & www.ti.com \\
\hline VCRCCLR & Clear CRC Result Register & \\
\hline \multicolumn{3}{|l|}{Operands} \\
\hline & mem16 16-bit memory location & \\
\hline Opcode & LSW: 1110010100100100 & \\
\hline Description & Clear the VCRC register.
\[
\mathrm{VCRC}=0 \times 0000
\] & \\
\hline Flags & This instruction does not modify any flags in the VSTATUS register. & \\
\hline Pipeline & This is a single-cycle instruction. & \\
\hline Example & Refer to the example for VCRC32L_1 mem16. & \\
\hline See also & \begin{tabular}{l}
VMOV32 mem32, VCRC \\
VMOV32 VCRC, mem32
\end{tabular} & \\
\hline
\end{tabular}

\section*{VMOV32 mem32, VCRC Store the CRC Result Register}
\begin{tabular}{|c|c|}
\hline \multirow[t]{3}{*}{Operands} & \\
\hline & mem32 32-bit memory destination \\
\hline & VCRC CRC result register \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001000000110 \\
\hline & MSW: 00000000 mem32 \\
\hline \multirow[t]{2}{*}{Description} & Store the VCRC register. \\
\hline & [mem32] = VCRC \\
\hline Flags & This instruction does not modify any flags in the VSTATUS register. \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multicolumn{2}{|l|}{Example} \\
\hline See also & \begin{tabular}{l}
VCRCCLR \\
VMOV32 VCRC, mem32
\end{tabular} \\
\hline
\end{tabular}

\section*{VMOV32 VCRC, mem32 Load the CRC Result Register}
\begin{tabular}{ll} 
Operands & \begin{tabular}{l} 
mem32 \\
VCRC
\end{tabular} \\
\begin{tabular}{ll} 
32-bit memory source \\
CRC result register
\end{tabular} \\
Opcode & \begin{tabular}{l} 
LSW: 1110001111110110 \\
MSW: \(00000000 \quad\) mem32
\end{tabular} \\
Description & \begin{tabular}{l} 
Load the VCRC register. \\
VCRC \(=[\) mem32 \(]\)
\end{tabular} \\
Flags & This instruction does not modify any flags in the VSTATUS register. \\
Pipeline & This is a single-cycle instruction. \\
Example & \\
See also & \begin{tabular}{l} 
VCRCCLR \\
\end{tabular}
\end{tabular}

TEXAS INSTRUMENTS

\subsection*{2.5.6 Deinterleaver Instructions}

The instructions are listed alphabetically, preceded by a summary.
Table 2-15. Deinterleaver Instructions
Title Page
VCLRDIVE —Clear DIVE bit in the VSTATUS Register ..... 288
VDEC VRaL - 16-bit Decrement ..... 289
VDEC VRaL || VMOV32 VRb, mem32 - 16-bit Decrement with Parallel Load ..... 290
VINC VRaL - 16-bit Increment ..... 291
VINC VRaL || VMOV32 VRb, mem32 - 16-bit Increment with Parallel Load ..... 292
VMOD32 VRaH, VRb, VRcH — Modulo Operation ..... 293
VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, VRe — Modulo Operation with Parallel Move ..... 294
VMOD32 VRaH, VRb, VRcL -Modulo Operation ..... 295
VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe —Modulo Operation with Parallel Move ..... 296
VMOV16 VRaL, VRbH — 16-bit Register Move ..... 297
VMOV16 VRaH, VRbL — 16-Bit Register Move ..... 298
VMOV16 VRaH, VRbH — 16-Bit Register Move ..... 299
VMOV16 VRaL, VRbL — 16-Bit Register Move ..... 300
VMPYADD VRa, VRaL, VRaH, VRbH — Multiply Add 16-Bit ..... 301
VMPYADD VRa, VRaL, VRaH, VRbL — Multiply Add 16-bit ..... 302

\section*{VCLRDIVE}

Operands
Opcode
Description
Flags
Pipeline
Example
See also

\section*{Clear DIVE bit in the VSTATUS Register}
none
LSW: 1110010100100000
Clear the DIVE (Divide by zero error) bit in the VSTATUS register.
This instruction clears the DIVE bit in the VSTATUS register
This is a single-cycle operation

\section*{VDEC VRaL}

\section*{16-bit Decrement}

Operands
VRaL Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L
\begin{tabular}{ll} 
Opcode & \begin{tabular}{l} 
LSW: 1110011011110010 \\
MSW: 000010110000 1aaa
\end{tabular} \\
Description & \begin{tabular}{l} 
16-bit Increment \\
VRaL \(=\) VRaL - 1
\end{tabular} \\
Flags & \begin{tabular}{l} 
This instruction does not affect any flags in the VSTATUS register
\end{tabular} \\
Pipeline & This is a single-cycle instruction \\
Example & \begin{tabular}{l} 
VDEC VROL ; VROL = VROL - 1
\end{tabular} \\
See also & \begin{tabular}{l} 
VINC VRaL || VMOV32 VRb, mem32 \\
\\
\\
\\
\\
\\
VINC VRaL \\
VDEC VRaL || VMOV32 VRb, mem32
\end{tabular}
\end{tabular}

\section*{VDEC VRaL || VMOV32 VRb, mem32 16-bit Decrement with Parallel Load}

\section*{Operands}
\begin{tabular}{ll}
\hline VRaL & Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L \\
VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110001010000001 \\
& MSW: 01bb baaa mem32
\end{tabular}
Description 16-bit Decrement with Parallel Load
```

VRaL = VRaL - 1
VRb = [mem32]

```

Flags
Pipeline
This instruction does not affect any flags in the VSTATUS register
This is a single-cycle instruction
Example
vDEC VROL || vMOV32 VR1, *+XAR3[4]
See also
VINC VRaL
VDEC VRaL
VINC VRaL || VMOV32 VRb, mem32

\section*{VINC VRaL}

16-bit Increment
Operands
VRaL Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L
\begin{tabular}{|c|c|}
\hline Opcode & LSW: 1110011011110010 \\
\hline & MSW: 000010110000 Oaaa \\
\hline Description & 16-bit Increment VRaL \(=\) VRaL +1 \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register \\
\hline Pipeline & This is a single-cycle instruction \\
\hline Example & vinc Vrol ; Vrol = Vrol + 1 \\
\hline See also & \begin{tabular}{l}
VINC VRaL || VMOV32 VRb, mem32 \\
VDEC VRaL \\
VDEC VRaL || VMOV32 VRb, mem32
\end{tabular} \\
\hline
\end{tabular}

\section*{VINC VRaL || VMOV32 VRb, mem32 16-bit Increment with Parallel Load}

\section*{Operands}
\begin{tabular}{ll}
\hline VRaL & Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L \\
VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
Opcode LSW: 1110001010000001

Description

\section*{Flags}

Pipeline
Example
See also

MSW: 00bb baaa mem32
16-bit Increment with parallel load
```

VRaL = VRaL +1
VRb}= [mem32

```

This instruction does not affect any flags in the VSTATUS register
This is a single-cycle instruction
vinc vrol || vMOV32 VR1, *+XAR3[4]
VINC VRaL
VDEC VRaL
VDEC VRaL || VMOV32 VRb, mem32

\section*{VMOD32 VRaH, VRb, VRcH Modulo Operation}

\section*{Operands}
\begin{tabular}{ll}
\hline VRaH & High word of a general purpose register: VR0H, VR1H...VR7H. Cannot be VR8H \\
VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRcH & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
\hline
\end{tabular}

Opcode LSW: 1110011010000000
MSW: 0010 100a aabb bccc

\section*{Description}

Modulo operation: 32-bit signed \%16 bit unsigned
```

if(VRcH == 0x0){
VSTATUS[DIVE] = 1
}else{
VRaH = VRb % VRcH
}

```

Flags This instruction modifies the following bits in the VSTATUS register:
- DIVE is set if VRcH is 0 i.e. a divide by zero error.

Pipeline \(\quad\) This is a \(9 p\) cycle instruction. No VMOD32 related instruction can be present in the delay slot of this instruction.

\section*{Example}

See also


VMOD32 VRaH, VRb, VRcL
VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, Vre VCLRDIVE

\section*{VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, VRe Modulo Operation with Parallel Move}

\section*{Operands}
\begin{tabular}{ll}
\hline VRaH & High word of a general purpose register: VR0H, VR1H...VR7H. Cannot be VR8H \\
VRb & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRcH & Low word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRd & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRe & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011011110011 \\
& MSW: 1eee dddc ccbb baaa
\end{tabular}

Description Modulo operation: 32-bit signed \% 16 bit unsigned
```

    if(VRcL == 0x0) \{
        VSTATUS[DIVE] = 1
    \}else\{
        \(\mathrm{VRaH}=\mathrm{VRb} \% \mathrm{VRcH}\)
    \}
    $\mathrm{VRd}=\mathrm{VRe}$

```

Flags This instruction modifies the following bits in the VSTATUS register:
- DIVE is set if VRcH is 0 , that is, a divide by zero error.

Pipeline This is a 9p/1 cycle instruction. The VMOD32 instruction takes 9p cycles while the VMOV32 operation completes in a single cycle. No VMOD32 related instruction can be present in the delay slot of this instruction.

\section*{Example}
```

VMOD32 VR5H, VR3, VR4H ; VR5H = VR3%VR4H = j; VRO = {J,I}
|| VMOV32 VR0, VR6 ; compute j = (b * J - v * i) % n;
; load back saved J,I
VINC VROL ; D1 VR1H = u, VR1L = a
|| VMOV32 VR1, *+XAR3[4] ; increment I; load u, a
MOV *+XAR1[AR0], AL ; D2 Save previous Y(i+j*m)
VCMPY VR3, VR2, VR1, VR0 ; D3 VR3 = a*I - u*J
; compute a * I - u * J
VMOV32 VR1, *+XAR3[2] ; D4/D1 VR1H = v, VR1L = b load v,b
MOV AL, *XAR4++ ; D5 AL = X(I) load X(I)
NOP ; D6
VMOV32 VR6, VR0 ; D7 VR6 = {J,I} save current {J,I}
VMOV16 VR0L, *+XAR5[0] ; D8 VR0L = J load J
VMOD32 VR0H, VR3, VR4H ; VR0H = (VR3 % VR4H) = i
; compute i = (a * I - u* J) % m;

```
\begin{tabular}{ll} 
See also & VMOD32 VRaH, VRb, VRcH \\
& VMOD32 VRaH, VRb, VRcL \\
& VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe \\
& VCLRDIVE
\end{tabular}

\section*{VMOD32 VRaH, VRb, VRcL Modulo Operation}

\section*{Operands}
\begin{tabular}{ll}
\hline VRaH & High word of a general purpose register: VR0H, VR1H...VR7H. Cannot be VR8H \\
VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRcL & Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L \\
\hline
\end{tabular}
Opcode LSW: 1110011010000000

\section*{Description}

Modulo operation: 32-bit signed \%16 bit unsigned
```

if(VRcL == 0x0){
VSTATUS[DIVE] = 1
}else{
VRaH = VRb % VRcL
}

```

Flags
This instruction modifies the following bits in the VSTATUS register:
- DIVE is set if VRcL is 0 , that is, a divide by zero error.

Pipeline This is a 9p cycle instruction. No VMOD32 related instruction can be present in the delay slot of this instruction.

\section*{Example}

See also
VMOD32 VRaH, VRb, VRcH
VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe
VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, Vre
VCLRDIVE

Texas
InsTruments

\section*{VMOD32 VRaH, VRb, VRcL || VMOV32 VRd, VRe Modulo Operation with Parallel Move}

\section*{Operands}
\begin{tabular}{ll}
\hline VRaH & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRcL & Low word of a general purpose register: VR0L, VR1L...VR7L. Cannot be VR8L \\
VRd & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRe & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\hline
\end{tabular}
Opcode \begin{tabular}{lllll} 
LSW: 1110 & 0110 & 1111 & 0011 \\
& MSW: 0eee dddc ccbb baaa
\end{tabular}

Description Modulo operation: 32-bit signed \%16 bit unsigned
```

    if(VRCL == 0x0) {
        VSTATUS[DIVE] = 1
    }else{
        VRaH = VRb % VRcL
    }
    VRd = VRe

```

Flags This instruction modifies the following bits in the VSTATUS register:
- DIVE is set if VRcH is 0 , that is, a divide by zero error.

Pipeline This is a 9p/1 cycle instruction. The VMOD32 instruction takes 9p cycles while the VMOV32 operation completes in a single cycle. No VMOD32 related instruction can be present in the delay slot of this instruction.

\section*{Example}
```

VMOD32 VR5H, VR3, VR4L ; VR5H = VR3%VR4L = j; VR0 = {J,I}
|| VMOV32 VR0, VR6 ; compute j = (b * J - v * i) % n;
; load back saved J,I
VINC VROL
|| VMOV32 VR1, *+XAR3[4] ; increment I; load u, a
MOV *+XAR1[AR0], AL ; D2 Save previous Y(i+j*m)
VCMPY VR3, VR2, VR1, VR0 ; D3 VR3 = a*I - u*J
; compute a * I - u * J
VMOV32 VR1, *+XAR3[2] ; D4/D1 VR1H = v, VR1L = b load v,b
MOV AL, *XAR4++ ; D5 AL = X(I) load X(I)
NOP ; D6
VMOV32 VR6, VR0 ; D7 VR6 = {J,I} save current {J,I}
VMOV16 VR0L, *+XAR5[0] ; D8 VR0L = J load J
VMOD32 VR0H, VR3, VR4H ; VR0H = (VR3 % VR4H) = i
; compute i = (a * I - u* J) % m;

```
\begin{tabular}{ll} 
See also & VMOD32 VRaH, VRb, VRcH \\
& VMOD32 VRaH, VRb, VRcL \\
& VMOD32 VRaH, VRb, VRcH || VMOV32 VRd, Vre \\
& VCLRDIVE
\end{tabular}

\section*{VMOV16 VRaL, VRbH 16-bit Register Move}

Operands
\begin{tabular}{ll}
\hline VRbH & High word of a general purpose register: VR0H, VR1H...VR7H. Cannot be VR8H \\
VRaL & Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011011110010 \\
MSW: 0000 1010 00bb baaa \\
Description & 16-bit Register Move \\
& VRaL \(=\) VRbH
\end{tabular}
Flags This instruction does not affect any flags in the VSTATUS register

\section*{Pipeline}

This is a single-cycle instruction
Example
VMOV16 VR5L, VROH ; VR5L = VROH
See also
VMOV16 VRaH, VRbL
VMOV16 VRaH, VRbH
VMOV16 VRaL, VRbL

\section*{VMOV16 VRaH, VRbL 16-Bit Register Move}

Operands
\begin{tabular}{ll}
\hline VRbL & Low word of a general purpose register: VR0L, VR1L...VR7L. Cannot be VR8L \\
VRaH & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & \begin{tabular}{l} 
LSW: 111001101111 O010 \\
MSW: 0000 1010 01bb baaa
\end{tabular} \\
Description & 16-bit Register Move \\
& VRaH \(=\) VRbL
\end{tabular}

Flags This instruction does not affect any flags in the VSTATUS register
Pipeline
This is a single-cycle instruction
Example
See also
vmov16 VR5H, VROL ; VR5H = VROL
VMOV16 VRaL, VRbH
VMOV16 VRaH, VRbH
VMOV16 VRaL, VRbL

\section*{VMOV16 VRaH, VRbH 16-Bit Register Move}

Operands
\begin{tabular}{ll}
\hline VRbH & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRaH & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011011110010 \\
MSW: 0000 1010 10bb baaa \\
Description & 16-bit Register Move \\
& VRaH \(=\) VRbH
\end{tabular}
Flags This instruction does not affect any flags in the VSTATUS register
\begin{tabular}{ll} 
Pipeline & This is a single-cycle ins \\
Example & VMOV16 VR5H, VRoH ; VR \\
See also & VMOV16 VRaL, VRbH \\
& VMOV16 VRaH, VRbL \\
& VMOV16 VRaL, VRbL
\end{tabular}

\section*{VMOV16 VRaL, VRbL 16-Bit Register Move}

Operands
\begin{tabular}{ll}
\hline VRbL & Low word of a general purpose register: VR0L, VR1L...VR7L. Cannot be VR8L \\
VRaL & Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011011110010 \\
MSW: 00001010 11bb baaa \\
Description & 16-bit Register Move \\
& VRaL \(=\) VRbL
\end{tabular}

Flags This instruction does not affect any flags in the VSTATUS register
Pipeline
This is a single-cycle instruction
Example
VMOV16 VR5L, VROL ; VR5L = VROL
See also
VMOV16 VRaL, VRbH
VMOV16 VRaH, VRbL
VMOV16 VRaH, VRbH

\section*{VMPYADD VRa, VRaL, VRaH, VRbH Multiply Add 16-Bit}

\section*{Operands}
\begin{tabular}{ll}
\hline VRbH & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRaH & Low word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRaL & Low word of a general purpose register: VR0L, VR1L....VR7L. Cannot be VR8L \\
VRa & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\hline
\end{tabular}
Opcode LSW: 1110011011110010

Description Performs \(p+q^{*} r\), where \(p, q\), and \(r\) are 16-bit values
If (VSTATUS[SAT] == 1) \{ If (VSTATUS[RND] == 1) \{
            VRa \(=\) rnd(sat(VRaL + VRaH * VRbH) >>VSTATUS[SHIFTR]);
        \}else \{
            VRa \(=\) sat (VRaL + VRaH * VRbH) >>VSTATUS[SHIFTR];
        \}
\}else \{ //VSTATUS[SAT] \(=0\)
    If (VSTATUS[RND] == 1) \{
            VRa \(=\) rnd((VRaL + VRaH * VRbH) >>VSTATUS[SHIFTR]);
    \}else \{
            VRa \(=(V R a L+V R a H * V R b H) \gg V S T A T U S[S H I F T R] ;\)
    \}
\}

It should be noted that:
- VRaH VRbH is represented as 32 -bit temp value
- VRaL should be sign extended to 32-bit before performing add
- The add operation is a 32 -bit operation

Flags This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow if 32-bit signed overflow is detected in the add operation.

\section*{Pipeline}

This is a \(2 p\) cycle operation

\section*{Example}
```

VMPYADD VR5, VR5L, VR5H, VR4H ; VR5 = VR5L + VR5H*VR4H
; = i + j*m compute i + j*m

```

See also VMPYADD VRa, VRaL, VRaH, VRbL

\section*{VMPYADD VRa, VRaL, VRaH, VRbL Multiply Add 16-bit}

\section*{Operands}
\begin{tabular}{ll}
\hline VRbL & High word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRaH & Low word of a general purpose register: VR0H, VR1H....VR7H. Cannot be VR8H \\
VRaL & Low word of a general purpose register: VR0L, VR1L...VR7L. Cannot be VR8L \\
VRa & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 0110 1111 0010
MSW: 0000 1100 01bb baaa

```

Description Performs \(p+q^{*} r\), where \(p, q\), and \(r\) are 16-bit values
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1) {
VRa = rnd(sat(VRaL + VRaH * VRbL) >>VSTATUS[SHIFTR]);
}else {
VRa = sat(VRaL + VRaH * VRbL)>>VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1){
VRa = rnd((VRaL + VRaH * VRbL) >>VSTATUS[SHIFTR]);
}else {
VRa = (VRaL + VRaH * VRbL) >>VSTATUS[SHIFTR];
}
}

```

It should be noted that:
- VRaH* VRbL is represented as 32 -bit temp value
- VRaL should be sign extended to 32-bit before performing add
- The add operation is a 32-bit operation

Flags
This instruction modifies the following bits in the VSTATUS register:
- - OVFR is set if signed overflow if 32-bit signed overflow is detected in the add operation.

Pipeline
This is a \(2 p\) cycle operation
Example
```

VMPYADD VR5, VR5L, VR5H, VR4L ; VR5 = VR5L + VR5H*VR4L
= i + j*m compute i + j*m
NOP ; D1

```

See also VMPYADD VRa, VRaL, VRaH, VRbH

INSTRUMENTS

\subsection*{2.5.7 FFT Instructions}

The instructions are listed alphabetically, preceded by a summary.

\section*{Table 2-16. FFT Instructions}
Title Page
VCFFT1 VR2, VR5, VR4 - Complex FFT calculation instruction ..... 304
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, \#1-bit -Complex FFT calculation instruction ..... 305
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, \#1-bit || VMOV32 mem32, VR1 —Complex FFT calculation instruction with Parallel Store ..... 307
VCFFT3 VR5, VR4, VR3, VR2, VR0, \#1-bit -Complex FFT calculation instruction ..... 309
VCFFT3 VR5, VR4, VR3, VR2, VR0, \#1-bit || VMOV32 VR5, mem32 - Complex FFT calculation instruction with Parallel Load ..... 311
VCFFT4 VR4, VR2, VR1, VR0, \#1-bit —Complex FFT calculation instruction ..... 313
VCFFT4 VR4, VR2, VR1, VR0, \#1-bit || VMOV32 VR7, mem32 - Complex FFT calculation instruction with Parallel Load ..... 315
VCFFT5 VR5, VR4, VR3, VR2, VR1, VR0, \#1-bit || VMOV32 mem32, VR1 - Complex FFT calculation instruction with Parallel Load ..... 317
VCFFT6 VR3, VR2, VR1, VR0, \#1-bit || VMOV32 mem32, VR1 —Complex FFT calculation instruction with Parallel Load ..... 319
VCFFT7 VR1, VR0, \#1-bit || VMOV32 VR2, mem32 - Complex FFT calculation instruction with Parallel Load ..... 320
VCFFT8 VR3, VR2, \#1-bit -Complex FFT calculation instruction ..... 321
VCFFT8 VR3, VR2, \#1-bit || VOMV32 mem32, VR4 -Complex FFT calculation instruction with Parallel Store ..... 322
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 \#1-bit -Complex FFT calculation instruction ..... 323
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 \#1-bit || VMOVE32 mem32, VR5 - Complex FFT calculation instruction with Parallel Store ..... 324
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 \#1-bit -Complex FFT calculation instruction ..... 326
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 \#1-bit || VMOV32 VR0, mem32 -Complex FFT calculation instruction with Parallel Load ..... 330

\section*{VCFFT1 VR2, VR5, VR4 Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part \(\operatorname{VRa}\) [15:0] \(=\) Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR4 & First Complex Input \\
VR5 & Second Complex Input \\
VR2 & Complex Output \\
\hline
\end{tabular}

\section*{Opcode LSW: 1110010100101011}

Description

Flags

Pipeline
Example

\section*{See also}

This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1) {
VR2H = rnd(sat(VR5H*VR4L - VR5L*VR4H) >>VSTATUS[SHIFTR])
VR2L = rnd(sat(VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR])
}else {
VR2H = sat(VR5H*VR4L - VR5L*VR4H) >>VSTATUS[SHIFTR]
VR2H = sat(VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR]
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VR2H = rnd((VR5H*VR4L - VR5L*VR4H) >>VSTATUS[SHIFTR])
VR2H = rnd((VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR])
}else {
VR2H = (VR5H*VR4L - VR5L*VR4H)>>VSTATUS[SHIFTR]
VR2L = (VR5L*VR4L + VR5H*VR4H)>>VSTATUS[SHIFTR]
}
}

```

Sign-Extension is automatically done for the shift right operations
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

This is a two cycle instruction
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

InsTRUMENTS

\section*{VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, \#1-bit Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part \(\operatorname{VRa}[15: 0]=\) Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR7 & Complex Input \\
VR6 & Complex Input \\
VR4 & Complex Input \\
VR2 & Complex Output \\
VR1 & Complex Output \\
VR0 & Complex Output \\
\#1-bit & 1-bit immediate value \\
\hline
\end{tabular}

\section*{Opcode}

Description
This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1){
VROH = rnd(sat(VR7H + VR2H)>>\#1-bit);
VROL = rnd(sat(VR7L + VR2L)>>\#1-bit);
VR1L = rnd(sat(VR7L - VR2L)>>\#1-bit);
VR1H = rnd(sat(VR7H - VR2H)>>\#1-bit);
VR2H = rnd(sat(VR6H * VR4L - VR6L * VR4H) >> VSTATUS[SHIFTR]);
VR2L = rnd(sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
}else {
VROH = sat(VR7H + VR2H)>>\#1-bit;
VROL = sat(VR7L + VR2L) >>\#1-bit;
VR1L = sat(VR7L - VR2L) >>\#1-bit;
VR1H = sat(VR7H - VR2H) >>\#1-bit;
VR2H = sat(VR6H * VR4L - VR6L * VR4H) >> VSTATUS[SHIFTR];
VR2L = sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1){
VR0H = rnd((VR7H + VR2H)>>\#1-bit);
VROL = rnd((VR7L + VR2L)>>\#1-bit);
VR1L = rnd((VR7L - VR2L)>>\#1-bit);
VR1H = rnd((VR7H - VR2H)>>\#1-bit);
VR2H = rnd((VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
VR2L = rnd((VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
}else {
VROH = (VR7H + VR2H)>>\#1-bit;
VROL = (VR7L + VR2L) >>\#1-bit;
VR1L = (VR7L - VR2L)>>\#1-bit;
VR1H = (VR7H - VR2H) >>\#1-bit;
VR2H = (VR6H * VR4L - VR6L * VR4H) >> VSTATUS[SHIFTR];
VR2L = (VR6L * VR4L + VR6H * VR4H) >> VSTATUS[SHIFTR];
}
}

```

Sign-Extension is automatically done for the shift right operations
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit
temporary result can't fit in 16-bit destination

Pipeline
Example
See also

\section*{VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, \#1-bit || VMOV32 mem32, VR1 Complex FFT calculation instruction with Parallel Store \\ Operands \\ This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part \\ VRa[15:0] = Real Part \\ It ignores the VSTATUS[CPACK] bit.}
\begin{tabular}{ll}
\hline VR7 & Complex Input \\
VR6 & Complex Input \\
VR4 & Complex Input \\
VR2 & Complex Output \\
VR1 & Complex Output \\
VR0 & Complex Output \\
\(\# 1-\) bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
Opcode \begin{tabular}{lll} 
LSW: 1110001000000111 \\
MSW: 0010 000I mem32
\end{tabular}

\section*{Description \\ This operation is used in the butterfly operation of the FFT:}
```

If(VSTATUS[SAT] == 1){
If(VSTATUS[RND] == 1){
VROH = rnd(sat(VR7H + VR2H)>>\#1-bit);
VROL = rnd(sat(VR7L + VR2L)>>\#1-bit);
VR1L = rnd(sat(VR7L - VR2L)>>\#1-bit);
VR1H = rnd(sat(VR7H - VR2H)>>\#1-bit);
VR2H = rnd(sat(VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
VR2L = rnd(sat(VR6L * VR4L + VR6H * VR4H) >> VSTATUS[SHIFTR]);
}else {
VROH = sat(VR7H + VR2H)>>\#1-bit;
VROL = sat(VR7L + VR2L) >>\#1-bit;
VR1L = sat(VR7L - VR2L)>>\#1-bit;
VR1H = sat(VR7H - VR2H) >>\#1-bit;
VR2H = sat(VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR];
VR2L = sat(VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VR0H = rnd((VR7H + VR2H)>>\#1-bit);
VROL = rnd((VR7L + VR2L)>>\#1-bit);
VR1L = rnd((VR7L - VR2L)>>\#1-bit);
VR1H = rnd((VR7H - VR2H) >>\#1-bit);
VR2H = rnd((VR6H * VR4L - VR6L * VR4H)>> VSTATUS[SHIFTR]);
VR2L = rnd((VR6L * VR4L + VR6H * VR4H)>> VSTATUS[SHIFTR]);
}else {
VROH = (VR7H + VR2H)>>\#1-bit;
VROL = (VR7L + VR2L)>>\#1-bit;
VR1L = (VR7L - VR2L) >>\#1-bit;
VR1H = (VR7H - VR2H) >>\#1-bit;
VR2H = (VR6H * VR4L - VR6L * VR4H) >> VSTATUS[SHIFTR];
VR2L = (VR6L * VR4L + VR6H * VR4H) >> VSTATUS[SHIFTR];
}
}
[mem32] = VR1;

```

Sign-Extension is automatically done for the shift right operations

Flags

Pipeline

Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

This is a \(2 p / 1\)-cycle instruction. The VCFFT operation takes \(2 p\) cycles and the VMOV operation completes in a single cycle.

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

INSTRUMENTS

\section*{VCFFT3 VR5, VR4, VR3, VR2, VR0, \#1-bit Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part VRa[15:0] = Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR5 & Complex Input \\
VR4 & Complex Input \\
VR3 & Complex Output \\
VR2 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\(\# 1\)-bit & 1-bit immediate value \\
\hline
\end{tabular}

\section*{Opcode}

LSW: \(101000010011001 I\)
Description
This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1) {
VROH = rnd(sat(VR5H + VR2H)>>\#1-bit);
VROL = rnd(sat(VR5L + VR2L)>>\#1-bit);
VR3H = rnd(sat(VR5H - VR2H)>>\#1-bit);
VR3L = rnd(sat(VR5L - VR2L)>>\#1-bit);
VR2H = rnd(sat(VROH * VR4L - VROL * VR4H)>>VSTATUS[SHIFTR]);
VR2L = rnd(sat(VROL * VR4L + VR0H * VR4H)>>VSTATUS[SHIFTR]);
}else {
VROH = sat(VR5H + VR2H)>>\#1-bit;
VROL = sat(VR5L + VR2L)>>\#1-bit;
VR3H = sat(VR5H - VR2H) >>\#1-bit;
VR3L = sat(VR5L - VR2L) >>\#1-bit;
VR2H = sat(VR0H * VR4L - VROL * VR4H) >>VSTATUS[SHIFTR];
VR2L = sat(VR0L * VR4L + VR0H * VR4H) >>VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VROH = rnd((VR5H + VR2H)>>\#1-bit);
VROL = rnd((VR5L + VR2L)>>\#1-bit);
VR3H = rnd((VR5H - VR2H)>>\#1-bit);
VR3L = rnd((VR5L - VR2L) >>\#1-bit);
VR2H = rnd((VR0H * VR4L - VROL * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd((VR0L * VR4L + VR0H * VR4H) >>VSTATUS[SHIFTR]);
}else {
VROH = (VR5H + VR2H)>>\#1-bit;
VROL = (VR5L + VR2L) >>\#1-bit;
VR3H = (VR5H - VR2H)>>\#1-bit;
VR3L = (VR5L - VR2L) >>\#1-bit;
VR2H = (VROH * VR4L - VROL * VR4H) >>VSTATUS[SHIFTR];
VR2L = (VROL * VR4L + VROH * VR4H) >>VSTATUS[SHIFTR];
}
}

```

Sign-Extension is automatically done for the shift right operations

Flags \(\quad\) This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

Pipeline

Example
See also

\section*{See also}

This is a \(2 p / 1\)-cycle instruction. The VCFFT operation takes \(2 p\) cycles and the VMOV operation completes in a single cycle.

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

Instruments

\section*{VCFFT3 VR5, VR4, VR3, VR2, VR0, \#1-bit || VMOV32 VR5, mem32 Complex FFT calculation instruction with Parallel Load}

\section*{Operands}

This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part VRa[15:0] = Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR5 & Complex Input \\
VR4 & Complex Input \\
VR3 & Complex Output \\
VR2 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110001010110000 \\
MSW: 0000 001I mem32
\end{tabular}

\section*{Description}

This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1){
If(VSTATUS[RND] == 1){
VROH = rnd(sat(VR5H + VR2H)>>\#1-bit);
VROL = rnd(sat(VR5L + VR2L)>>\#1-bit);
VR3H = rnd(sat(VR5H - VR2H)>>\#1-bit);
VR3L = rnd(sat(VR5L - VR2L)>>\#1-bit);
VR2H = rnd(sat(VROH * VR4L - VROL * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd(sat(VROL * VR4L + VROH * VR4H) >>VSTATUS[SHIFTR]);
}else {
VR0H = sat(VR5H + VR2H) >>\#1-bit;
VROL = sat(VR5L + VR2L)>>\#1-bit;
VR3H = sat(VR5H - VR2H) >>\#1-bit;
VR3L = sat(VR5L - VR2L) >>\#1-bit;
VR2H = sat(VROH * VR4L - VROL * VR4H) >>VSTATUS[SHIFTR];
VR2L = sat(VR0L * VR4L + VROH * VR4H) >>VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VROH = rnd((VR5H + VR2H)>>\#1-bit);
VROL = rnd((VR5L + VR2L)>>\#1-bit);
VR3H = rnd((VR5H - VR2H) >>\#1-bit);
VR3L = rnd((VR5L - VR2L) >>\#1-bit);
VR2H = rnd((VR0H * VR4L - VR0L * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd((VR0L * VR4L + VROH * VR4H) >>VSTATUS[SHIFTR]);
}else {
VROH = (VR5H + VR2H)>>\#1-bit;
VROL = (VR5L + VR2L)>>\#1-bit;
VR3H = (VR5H - VR2H) >>\#1-bit;
VR3L = (VR5L - VR2L) >>\#1-bit;
VR2H = (VR0H * VR4L - VROL * VR4H) >>VSTATUS[SHIFTR];
VR2L = (VROL * VR4L + VROH * VR4H) >>VSTATUS[SHIFTR];
}
}
VR5 = [mem32];

```

Sign-Extension is automatically done for the shift right operations

\section*{Flags}

Pipeline
Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

This is a \(2 p\) cycle instruction.
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

InsTRUMENTS

\section*{VCFFT4 VR4, VR2, VR1, VR0, \#1-bit Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part VRa[15:0] = Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR4 & Complex Input \\
VR2 & Complex Output/Complex Input from previous operation \\
VR1 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
\hline
\end{tabular}

\section*{Opcode}

LSW: 101000010011010 I

\section*{Description}

This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1){
VROH = rnd(sat(VROH + VR2H)>>\#1-bit);
VROL = rnd(sat(VROL + VR2L)>>\#1-bit);
VR1H = rnd(sat(VROH - VR2H)>>\#1-bit);
VR1L = rnd(sat(VROL - VR2L)>>\#1-bit);
VR2H = rnd(sat(VR1L * VR4L + VR1H * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd(sat(VR1H * VR4L - VR1L * VR4H)>>VSTATUS[SHIFTR]);
}else {
VROH = sat(VROH + VR2H)>>\#1-bit;
VROL = sat(VROL + VR2L) >>\#1-bit;
VR1H = sat(VROH - VR2H) >>\#1-bit;
VR1L = sat(VROL - VR2L) >>\#1-bit;
VR2H = sat(VR1L * VR4L + VR1H * VR4H) >>VSTATUS[SHIFTR];
VR2L = sat(VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VROH = rnd((VROH + VR2H)>>\#1-bit);
VROL = rnd((VROL + VR2L)>>\#1-bit);
VR1H = rnd((VROH - VR2H)>>\#1-bit);
VR1L = rnd((VROL - VR2L)>>\#1-bit);
VR2H = rnd((VR1L * VR4L + VR1H * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd((VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR]);
}else {
VROH = (VROH + VR2H)>>\#1-bit;
VROL = (VROL + VR2L)>>\#1-bit;
VR1H = (VROH - VR2H) >>\#1-bit;
VR1L = (VROL - VR2L) >>\#1-bit;
VR2H = (VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR];
VR2L = (VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR];
}
}

```

Sign-Extension is automatically done for the shift right operations

Flags

Pipeline
Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

This is a \(2 p\) cycle instruction.
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

InSTRUMENTS
www.ti.com
VCFFT4 VR4, VR2, VR1, VR0, \#1-bit || VMOV32 VR7, mem32 - Complex FFT calculation instruction with
Parallel Load

\section*{VCFFT4 VR4, VR2, VR1, VR0, \#1-bit || VMOV32 VR7, mem32 Complex FFT calculation instruction} with Parallel Load

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR4 & Complex Input \\
VR2 & Complex Output/Complex Input from previous operation \\
VR1 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
Opcode \begin{tabular}{ll} 
LSW: 1110001010110000 \\
MSW: 0000 010I mem32
\end{tabular}

\section*{Description}

This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1) {
VROH = rnd(sat(VROH + VR2H)>>\#1-bit);
VROL = rnd(sat(VROL + VR2L)>>\#1-bit);
VR1H = rnd(sat(VR0H - VR2H)>>\#1-bit);
VR1L = rnd(sat(VROL - VR2L)>>\#1-bit);
VR2H = rnd(sat(VR1L * VR4L + VR1H * VR4H)>>VSTATUS[SHIFTR]);
VR2L = rnd(sat(VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR]);
}else {
VROH = sat(VROH + VR2H) >>\#1-bit;
VROL = sat(VROL + VR2L) >>\#1-bit;
VR1H = sat(VROH - VR2H) >>\#1-bit;
VR1L = sat(VROL - VR2L) >>\#1-bit;
VR2H = sat(VR1L * VR4L + VR1H * VR4H) >>VSTATUS[SHIFTR];
VR2L = sat(VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1){
VROH = rnd((VROH + VR2H) >>\#1-bit);
VROL = rnd((VROL + VR2L)>>\#1-bit);
VR1H = rnd((VROH - VR2H)>>\#1-bit);
VR1L = rnd((VROL - VR2L)>>\#1-bit);
VR2H = rnd((VR1L * VR4L + VR1H * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd((VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR]);
}else {
VROH = (VROH + VR2H)>>\#1-bit;
VROL = (VROL + VR2L)>>\#1-bit;
VR1H = (VR0H - VR2H)>>\#1-bit;
VR1L = (VROL - VR2L) >>\#1-bit;
VR2H = (VR1L * VR4L + VR1H * VR4H) >>VSTATUS[SHIFTR];
VR2L = (VR1H * VR4L - VR1L * VR4H) >>VSTATUS[SHIFTR];
}
}
VR7 = [mem32];

```

Sign-Extension is automatically done for the shift right operations

VCFFT4 VR4, VR2, VR1, VR0, \#1-bit || VMOV32 VR7, mem32 - Complex FFT calculation instruction with Parallel Load www.ti.com

\section*{Flags}

Pipeline
Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

This is a \(2 p\) cycle instruction.
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

InSTRUMENTS

\section*{VCFFT5 VR5, VR4, VR3, VR2, VR1, VR0, \#1-bit || VMOV32 mem32, VR1 Complex FFT calculation instruction with Parallel Load \\ Operands \\ This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part VRa[15:0] = Real Part It ignores the VSTATUS[CPACK] bit.}
\begin{tabular}{ll}
\hline VR5 & Complex Input \\
VR4 & Complex Input \\
VR3 & Complex Input \\
VR2 & Complex Output/Complex Input from previous operation \\
VR1 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110001000000111 \\
MSW: 0010001 I mem32
\end{tabular}

\section*{Description \\ This operation is used in the butterfly operation of the FFT:}
```

If(VSTATUS[SAT] == 1){
If(VSTATUS[RND] == 1) {
VROH = rnd(sat(VR3H - VR2H)>>\#1-bit);
VROL = rnd(sat(VR3L + VR2L)>>\#1-bit);
VR1H = rnd(sat(VR3H + VR2H)>>\#1-bit);
VR1L = rnd(sat(VR3L - VR2L)>>\#1-bit);
VR2H = rnd(sat(VR5H * VR4L - VR5L * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd(sat(VR5L * VR4L + VR5H * VR4H)>>VSTATUS[SHIFTR]);
}else {
VR0H = sat(VR3H - VR2H) >>\#1-bit;
VROL = sat(VR3L + VR2L) >>\#1-bit;
VR1H = sat(VR3H + VR2H)>>\#1-bit;
VR1L = sat(VR3L - VR2L) >>\#1-bit;
VR2H = sat(VR5H * VR4L - VR5L * VR4H) >>VSTATUS[SHIFTR];
VR2L = sat(VR5L * VR4L + VR5H * VR4H) >>VSTATUS[SHIFTR];
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VROH = rnd((VR3H - VR2H)>>\#1-bit);
VROL = rnd((VR3L + VR2L)>>\#1-bit);
VR1H = rnd((VR3H + VR2H)>>\#1-bit);
VR1L = rnd((VR3L - VR2L) >>\#1-bit);
VR2H = rnd((VR5H * VR4L - VR5L * VR4H) >>VSTATUS[SHIFTR]);
VR2L = rnd((VR5L * VR4L + VR5H * VR4H)>>VSTATUS[SHIFTR]);
}else {
VROH = (VR3H - VR2H)>>\#1-bit;
VROL = (VR3L + VR2L)>>\#1-bit;
VR1H = (VR3H + VR2H)>>\#1-bit;
VR1L = (VR3L - VR2L) >>\#1-bit;
VR2H = (VR5H * VR4L - VR5L * VR4H) >>VSTATUS[SHIFTR];
VR2L = (VR5L * VR4L + VR5H * VR4H) >>VSTATUS[SHIFTR];
}
}
[mem32] = VR1;

```

Sign-Extension is automatically done for the shift right operations

\section*{Flags}

Pipeline
Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH
- The OVFR and OVFI flags are also set if, after shift right operation, the 32-bit temporary result can't fit in 16-bit destination

This is a \(2 p\) cycle instruction.
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\section*{VCFFT6 VR3, VR2, VR1, VR0, \#1-bit || VMOV32 mem32, VR1 Complex FFT calculation instruction with Parallel Load}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR3 & Complex Input \\
VR2 & Complex Output/Complex Input from previous operation \\
VR1 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Opcode & \begin{tabular}{l}
LSW: 1110001000000111 \\
MSW: 0010 O10I mem32
\end{tabular} \\
\hline \multirow[t]{27}{*}{Description} & This operation is used in the butterfly operation \\
\hline & If (VSTATUS[SAT] == 1) \{ \\
\hline & If (VSTATUS [RND] == 1) \(\{\) \\
\hline & VROH = rnd (sat (VR3H - VR2H) >>\#1-bit); \\
\hline & VROL = rnd(sat (VR3L + VR2L) >>\#1-bit); \\
\hline & VR1H = rnd (sat (VR3H + VR2H) >>\#1-bit); \\
\hline & VR1L = rnd(sat (VR3L - VR2L) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VROH = sat (VR3H - VR2H) >>\#1-bit; \\
\hline & VROL = sat (VR3L + VR2L) >>\#1-bit; \\
\hline & VR1H = sat (VR3H + VR2H) >>\#1-bit; \\
\hline & VR1L = sat (VR3L - VR2L) >>\#1-bit; \\
\hline & \} \\
\hline & \}else \{ //vSTATUS[SAT] = 0 \\
\hline & If (VSTATUS[RND] == 1) \{ \\
\hline & VROH = rnd ( (VR3H - VR2H) >>\#1-bit); \\
\hline & VROL = rnd ((VR3L + VR2L) >>\#1-bit); \\
\hline & VR1H = rnd ( \(\mathrm{VR} 3 \mathrm{H}+\mathrm{VR} 2 \mathrm{H}\) ) >>\#1-bit) ; \\
\hline & VR1L = rnd((VR3L - VR2L) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VROH = (VR3H - VR2H) >>\#1-bit; \\
\hline & VROL \(=(\) VR3L + VR2L) >>\#1-bit; \\
\hline & VR1H = (VR3H + VR2H) >>\#1-bit; \\
\hline & VR1L \(=(\) VR3L - VR2L) >>\#1-bit; \\
\hline & \(\}\) ) \\
\hline & \} \\
\hline & [mem32] = VR1; \\
\hline
\end{tabular}

Flags

Pipeline

Example See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\section*{See also}

\section*{VCFFT7 VR1, VR0, \#1-bit || VMOV32 VR2, mem32 Complex FFT calculation instruction with Parallel Load}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part VRa[15:0] = Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR3 & Complex Input \\
VR2 & Complex Output/Complex Input from previous operation \\
VR1 & Complex Output/Complex Input from previous operation \\
VR0 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001010110000 \\
\hline & MSW: 0000 011I mem32 \\
\hline \multirow[t]{27}{*}{Description} & This operation is used in the butterfly operation \\
\hline & If (VSTATUS[SAT] = = 1) \{ \\
\hline & If (VSTATUS[RND] == 1) \(\{\) \\
\hline & VROL = rnd (sat (VROL + VR1L)>>\#1-bit); \\
\hline & VROH = rnd (sat (VROL - VR1L) >>\#1-bit); \\
\hline & VR1L = rnd (sat (VROH + VR1H) >>\#1-bit); \\
\hline & VR1H = rnd (sat (VR0H - VR1H) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VROL = sat (VROL + VR1L) >>\#1-bit; \\
\hline & VROH = sat (VROL - VR1L) >>\#1-bit; \\
\hline & VR1L = sat (VR0H + VR1H) >>\#1-bit; \\
\hline & VR1H \(=\) sat(VR0H - VR1H) >>\#1-bit; \\
\hline & \} \\
\hline & \}else \{ //vSTATUS[SAT] = 0 \\
\hline & If (VSTATUS[RND] == 1) \{ \\
\hline & VROL = rnd ((VROL + VR1L) >>\#1-bit); \\
\hline & VROH = rnd ( (VROL - VR1L) >>\#1-bit); \\
\hline & VR1L \(=\) rnd((VR0H + VR1H) >>\#1-bit); \\
\hline & VR1H = rnd((VR0H - VR1H)>>\#1-bit); \\
\hline & \}else \{ \\
\hline & VROL = (VROL + VR1L)>>\#1-bit; \\
\hline & VROH = (VROL - VR1L)>>\#1-bit; \\
\hline & VR1L \(=\) (VR0H + VR1H)>>\#1-bit; \\
\hline & VR1H = (VROH - VR1H)>>\#1-bit; \\
\hline & \} VR1н \\
\hline & \} \\
\hline & VR2 = [mem32]; \\
\hline
\end{tabular}

Flags

Pipeline

Example

\section*{See also}

INSTRUMENTS

\section*{VCFFT8 VR3, VR2, \#1-bit Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part \(\operatorname{VRa}\) [15:0] \(=\) Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR2 & Complex Output/Complex Input from previous operation \\
VR3 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
\hline
\end{tabular}
Opcode LSW: 101000010011011 I

Description
This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1){
If(VSTATUS[RND] == 1) {
VR2L = rnd(sat(VR2L + VR3L)>>\#1-bit);
VR2H = rnd(sat(VR2L - VR3L)>>\#1-bit);
VR3L = rnd(sat(VR2H + VR3H)>>\#1-bit);
VR3H = rnd(sat(VR2H - VR3H)>>\#1-bit);
}else {
VR2L = sat(VR2L + VR3L) >>\#1-bit;
VR2H = sat(VR2L - VR3L) >>\#1-bit;
VR3L = sat(VR2H + VR3H) >>\#1-bit;
VR3H = sat(VR2H - VR3H) >>\#1-bit;
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VR2L = rnd((VR2L + VR3L)>>\#1-bit);
VR2H = rnd((VR2L - VR3L) >>\#1-bit);
VR3L = rnd((VR2H + VR3H)>>\#1-bit);
VR3H = rnd((VR2H - VR3H)>>\#1-bit);
}else {
VR2L = (VR2L + VR3L)>>\#1-bit;
VR2H = (VR2L - VR3L) >>\#1-bit;
VR3L = (VR2H + VR3H)>>\#1-bit;
VR3H = (VR2H - VR3H) >>\#1-bit;
}
}

```

Sign-Extension is automatically done for the shift right operations
This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline
This is a single cycle instruction.
Example
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\section*{See also}

\section*{VCFFT8 VR3, VR2, \#1-bit || VOMV32 mem32, VR4 Complex FFT calculation instruction with Parallel Store}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR4 & Complex Input from previous operation \\
VR2 & Complex Output/Complex Input from previous operation \\
VR3 & Complex Output/Complex Input from previous operation \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001000000111 \\
\hline & MSW: 0010 011I mem32 \\
\hline \multirow[t]{27}{*}{Description} & This operation is used in the butterfly operation \\
\hline & If (VSTATUS[SAT] = = 1) \{ \\
\hline & If (VSTATUS[RND] == 1) \{ \\
\hline & VR2L = rnd(sat (VR2L + VR3L)>>\#1-bit); \\
\hline & VR2H = rnd (sat (VR2L - VR3L) >>\#1-bit); \\
\hline & VR3L = rnd(sat (VR2H + VR3H) >>\#1-bit); \\
\hline & VR3H = rnd(sat (VR2H - VR3H) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VR2L = sat (VR2L + VR3L) >>\#1-bit; \\
\hline & VR2H = sat (VR2L - VR3L) >>\#1-bit; \\
\hline & VR3L = sat (VR2H + VR3H) >>\#1-bit; \\
\hline & VR3H = sat (VR2H - VR3H) >>\#1-bit; \\
\hline & \} \\
\hline & \}else \{ //vSTATUS[SAT] = 0 \\
\hline & If (VSTATUS[RND] == 1) \{ \\
\hline & VR2L = rnd((VR2L + VR3L)>>\#1-bit); \\
\hline & VR2H = rnd ((VR2L - VR3L)>>\#1-bit); \\
\hline & VR3L = rnd((VR2H + VR3H) >>\#1-bit); \\
\hline & VR3H = rnd((VR2H - VR3H)>>\#1-bit); \\
\hline & \}else \{ \\
\hline & VR2L = (VR2L + VR3L) >>\#1-bit; \\
\hline & VR2H = (VR2L - VR3L) >>\#1-bit; \\
\hline & VR3L = (VR2H + VR3H) >>\#1-bit; \\
\hline & VR3H = (VR2H - VR3H) >>\#1-bit; \\
\hline & \} VR3н \({ }^{\text {a }}\) \\
\hline & \} \\
\hline & [mem32] = VR4; \\
\hline
\end{tabular}

Sign-Extension is automatically done for the shift right operations
Flags \(\quad\) This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

\section*{Pipeline}

Example
This is a single cycle instruction.
See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\section*{See also}

INSTRUMENTS

\section*{VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 \#1-bit Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part \(\operatorname{VRa}[15: 0]=\) Real Part It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR0 & Complex Input \\
VR1 & Complex Input \\
VR2 & Complex Input \\
VR3 & Complex Input \\
VR4 & Complex Output \\
VR5 & Complex Output \\
\(\# 1-\) bit & 1-bit immediate value \\
\hline
\end{tabular}

\section*{Opcode}

Description This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1){
VR4L = rnd(sat(VROL + VR2L)>>\#1-bit);
VR4H = rnd(sat(VR1L + VR3L)>>\#1-bit);
VR5L = rnd(sat(VROL - VR2L)>>\#1-bit);
VR5H = rnd(sat(VR1L - VR3L)>>\#1-bit);
}else {
VR4L = sat(VROL + VR2L) >>\#1-bit;
VR4H = sat(VR1L + VR3L) >>\#1-bit;
VR5L = sat(VROL - VR2L) >>\#1-bit;
VR5H = sat(VR1L - VR3L) >>\#1-bit;
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VR4L = rnd((VROL + VR2L)>>\#1-bit);
VR4H = rnd((VR1L + VR3L)>>\#1-bit);
VR5L = rnd((VROL - VR2L)>>\#1-bit);
VR5H = rnd((VR1L - VR3L) >>\#1-bit);
}else {
VR4L = (VROL + VR2L)>>\#1-bit;
VR4H = (VR1L + VR3L)>>\#1-bit;
VR5L = (VROL - VR2L) >>\#1-bit;
VR5H = (VR1L - VR3L)>>\#1-bit;
}
}

```

Sign-Extension is automatically done for the shift right operations
Flags \(\quad\) This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline This is a single cycle instruction.
Example See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\section*{VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0 \#1-bit || VMOVE32 mem32, VR5 Complex FFT calculation instruction with Parallel Store}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR0 & Complex Input \\
VR1 & Complex Input \\
VR2 & Complex Input \\
VR3 & Complex Input \\
VR4 & Complex Output \\
VR5 & Complex Output \\
\#1-bit & 1-bit immediate value \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 0010 0000 0111
MSW: 0010 100I mem32

```

Description This operation is used in the butterfly operation of the FFT:
```

If(VSTATUS[SAT] == 1) {
If(VSTATUS[RND] == 1){
VR4L = rnd(sat(VROL + VR2L)>>\#1-bit);
VR4H = rnd(sat(VR1L + VR3L)>>\#1-bit);
VR5L = rnd(sat(VR0L - VR2L)>>\#1-bit);
VR5H = rnd(sat(VR1L - VR3L)>>\#1-bit);
}else {
VR4L = sat(VROL + VR2L)>>\#1-bit;
VR4H = sat(VR1L + VR3L) >>\#1-bit;
VR5L = sat(VROL - VR2L) >>\#1-bit;
VR5H = sat(VR1L - VR3L) >>\#1-bit;
}
}else { //VSTATUS[SAT] = 0
If(VSTATUS[RND] == 1) {
VR4L = rnd((VROL + VR2L) >>\#1-bit);
VR4H = rnd((VR1L + VR3L)>>\#1-bit);
VR5L = rnd((VR0L - VR2L)>>\#1-bit);
VR5H = rnd((VR1L - VR3L) >>\#1-bit);
}else {
VR4L = (VROL + VR2L)>>\#1-bit;
VR4H = (VR1L + VR3L)>>\#1-bit;
VR5L = (VR0L - VR2L) >>\#1-bit;
VR5H = (VR1L - VR3L) >>\#1-bit;
}
}
[mem32] = VR5;

```

Sign-Extension is automatically done for the shift right operations

\section*{Flags}

Pipeline

Example
See also

This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

This is a 1/1-cycle instruction. The VCFFT and VMOV operations are completed in one cycle.

See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\section*{VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 \#1-bit Complex FFT calculation instruction}

Operands
This operation assumes the following complex packing order for complex operands:
VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR0 & Complex Input \\
VR1 & Complex Input \\
VR2 & Complex Input \\
VR3 & Complex Input \\
VR6 & Complex Output \\
VR7 & Complex Output \\
\(\# 1-\) bit & 1-bit immediate value \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Opcode & LSW: 101000010011 101I \\
\hline \multirow[t]{26}{*}{Description} & This operation is used in the butterfly operation \\
\hline & If (VSTATUS[SAT] == 1) \{ \\
\hline & If (VSTATUS[RND] == 1) \{ \\
\hline & VR6L = rnd(sat (VR0H + VR3H) >>\#1-bit); \\
\hline & VR6H = rnd (sat (VR1H - VR2H) >>\#1-bit); \\
\hline & VR7L = rnd(sat (VROH - VR3H) >>\#1-bit); \\
\hline & VR7H = rnd(sat (VR1H + VR2H) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VR6L = sat (VROH + VR3H) >>\#1-bit; \\
\hline & VR6H = sat (VR1H - VR2H) >>\#1-bit; \\
\hline & VR7L = sat (VR0H - VR3H) >>\#1-bit; \\
\hline & VR7H = sat (VR1H + VR2H) >>\#1-bit; \\
\hline & \} VR7н \({ }^{\text {a }}\) \\
\hline & \}else \{ //vSTATUS[SAT] = 0 \\
\hline & If (VSTATUS[RND] == 1) \{ \\
\hline & VR6L = rnd ((VR0H + VR3H) >>\#1-bit); \\
\hline & VR6H = rnd((VR1H - VR2H) >>\#1-bit); \\
\hline & VR7L = rnd((VROH - VR3H) >>\#1-bit); \\
\hline & VR7H = rnd ((VR1H + VR2H) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VR6L \(=(\mathrm{VROH}+\mathrm{VR} 3 \mathrm{H}) \gg \# 1-\mathrm{bit}\); \\
\hline & VR6H = (VR1H - VR2H) >>\#1-bit; \\
\hline & VR7L \(=(\) VROH - VR3H \() \gg \# 1-\) bit; \\
\hline & VR7H = (VR1H + VR2H) >>\#1-bit; \\
\hline & \} \\
\hline & \} \\
\hline
\end{tabular}

Sign-Extension is automatically done for the shift right operations
Flags \(\quad\) This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

Pipeline This is a single cycle instruction.

\section*{Example}
```

_CFFT_run1024Pt:
etc ...
MOVL *-SP[ARG_OFFSET], XAR4
VSATON

```
```

_CFFT_run1024Pt_stages1and2Combined:
MOVZ AR0, *+XAR4 [NSAMPLES_OFFSET]
MOVL XAR2, *+XAR4[INBUFFER_OFFSET]
MOVL XAR1, *+XAR4[OUTBUFFER_OFFSET]
.lp_amode
SETC AMODE
NOP *,ARP2
VMOV32 VR0, *BR0++
VMOV32 VR1, *BR0++
VCFFT7 VR1, VR0, \#1
|| VMOV32 VR2, *BR0++
VMOV32 VR3, *BR0++
VCFFT8 VR3, VR2, \#1
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0, \#1
.align 2
RPTB _CFFT_run1024Pt_stages1and2CombinedLoop, \#S12_LOOP_COUNT
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1
|| VMOV32 VR0, *BR0++
VMOV32 VR1, *BR0++
VCFFT7 VR1, VR0, \#1
|| VMOV32 VR2, *BR0++
VMOV32 VR3, *BR0++
VCFFT8 VR3, VR2, \#1
|| VMOV32 *XAR1++, VR4
VMOV32 *XAR1++, VR6
VCFFT9 VR5, VR4, VR3, VR2, VR1, VR0, \#1
|| VMOV32 *XAR1++, VR5
VMOV32 *++, VR7, ARP2
_CFFT_run1024Pt_stages1and2CombinedLoop:
VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1
VMOV32 *XAR1++, VR4
VMOV32 *XAR1++, VR6
VMOV32 *XAR1++, VR5
VMOV32 *XAR1++, VR7
_CFFT_run1024Pt_stages1and2CombinedEnd:
.c28_amode
CLRC AMODE
_CFFT_run1024Pt_stages3and4Combined:
..
etc ...
...
VSETSHR \#15
VRNDON
MOVL XAR2, *+XAR4[S34_INPUT_OFFSET]
MOVL XAR1, \#S34_INSEP
MOVL XAR0, \#S34_OUTSEP
MOVL XAR6, *+XAR4[S34_OUTPUT_OFFSET]
MOVL XAR7, XAR6
ADDB XAR7, \#S34_GROUPSEP
MOVL XAR3, \#_vcu2_twiddleFactors

```
```

    MOVL *-SP[TFPTR_OFFSET], XAR3
    MOVL XAR4, XAR2
    ADDB XAR4, #S34_GROUPSEP
    MOVL XAR5, #S34_OUTER_LOOP_COUNT
    _CFFT_run1024Pt_stages3and4OuterLoop:
MOVL XAR3, *-SP[TFPTR_OFFSET]
; Inner Butterfly Loop
VMOV32 VR5, *+XAR4[AR1]
VMOV32 VR6, *+XAR2[AR1]
VMOV32 VR7, *XAR4++
VMOV32 VR4, *XAR3++
VCFFT1 VR2, VR5, VR4
VMOV32 VR5, *XAR2++
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, \#1
.align 2
RPTB _CFFT_run1024Pt_stages3and4InnerLoop, \#S34_INNER_LOOP_COUNT
VMOV32 VR4, *XAR3++
VCFFT3 VR5, VR4, VR3, VR2, VR0, \#1
|| VMOV32 VR5, *+XAR4[AR1]
VMOV32 VR6, *+XAR2[AR1]
VCFFT4 VR4, VR2, VR1, VR0, \#1
|| VMOV32 VR7, *XAR4++
VMOV32 VR4, *XAR3++
VMOV32 *XAR6++, VR0
VCFFT5 VR5, VR4, VR3, VR2, VR1, VR0, \#1
|| VMOV32 *XAR7++, VR1
VMOV32 VR5, *XAR2++
VMOV32 *+XAR6[AR0], VR0
VCFFT2 VR7, VR6, VR4, VR2, VR1, VR0, \#1
|| VMOV32 *+XAR7[AR0], VR1
_CFFT_run1024Pt_stages3and4InnerLoop:
VMOV32 VR4, *XAR3++
VCFFT3 VR5, VR4, VR3, VR2, VR0, \#1
NOP
VCFFT4 VR4, VR2, VR1, VR0, \#1
NOP
VMOV32 *XAR6++, VR0
VCFFT6 VR3, VR2, VR1, VR0, \#1
|| VMOV32 *XAR7++, VR1
NOP
VMOV32 *+XAR6[AR0], VR0
VMOV32 *+XAR7[AR0], VR1
ADDB XAR2, \#S34_POST_INCREMENT
ADDB XAR4, \#S34_POST_INCREMENT
ADDB XAR6, \#S34_POST_INCREMENT
ADDB XAR7, \#S34_POST_INCREMENT
BANZ _CFFT_run1024Pt_stages3and4OuterLoop, AR5--
_CFFT_run1024Pt_stages3and4CombinedEnd:

```

\footnotetext{
See also
The entire FFT implementation, with accompanying code comments, can be found in the VCU Library in controlSUITE.
}

VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 \#1-bit || VMOV32 VR0, mem32 - Complex FFT calculation instruction with Parallel Load
www.ti.com

\section*{VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0 \#1-bit || VMOV32 VR0, mem32 Complex FFT calculation instruction with Parallel Load}

Operands
This operation assumes the following complex packing order for complex operands: VRa[31:16] = Imaginary Part
VRa[15:0] = Real Part
It ignores the VSTATUS[CPACK] bit.
\begin{tabular}{ll}
\hline VR0 & Complex Input \\
VR1 & Complex Input \\
VR2 & Complex Input \\
VR3 & Complex Input \\
VR6 & Complex Output \\
VR7 & Complex Output \\
\#1-bit & 1-bit immediate value \\
mem32 & pointer to 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001010110000 \\
\hline & MSW: 0000 100I mem32 \\
\hline \multirow[t]{27}{*}{Description} & This operation is used in the butterfly operation \\
\hline & If (VSTATUS [SAT] == 1) \{ \\
\hline & If (VSTATUS [RND] == 1) \{ \\
\hline & VR6L = rnd(sat (VROH + VR3H) >>\#1-bit); \\
\hline & VR6H = rnd(sat (VR1H - VR2H) >>\#1-bit); \\
\hline & VR7L = rnd (sat (VROH - VR3H) >>\#1-bit); \\
\hline & VR7H = rnd(sat (VR1H + VR2H) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VR6L = sat (VR0H + VR3H) >>\#1-bit; \\
\hline & VR6H = sat (VR1H - VR2H) >>\#1-bit; \\
\hline & VR7L = sat (VROH - VR3H) >>\#1-bit; \\
\hline & VR7H = sat (VR1H + VR2H) >>\#1-bit; \\
\hline & \} \\
\hline & \}else \{ //vSTATUS[SAT] = 0 \\
\hline & If (VSTATUS [RND] == 1) \{ \\
\hline & VR6L \(=\) rnd ( (VR0H + VR3H) >>\#1-bit); \\
\hline & VR6H = rnd ((VR1H - VR2H) >>\#1-bit); \\
\hline & VR7L = rnd ((VR0H - VR3H)>>\#1-bit); \\
\hline & VR7H = rnd ( (VR1H + VR2H) >>\#1-bit); \\
\hline & \}else \{ \\
\hline & VR6L \(=\) (VROH + VR3H) >>\#1-bit; \\
\hline & VR6H = (VR1H - VR2H) >>\#1-bit; \\
\hline & VR7L \(=(\) VROH - VR3H) \(\gg \# 1-\mathrm{bit}\); \\
\hline & VR7H = (VR1H + VR2H) >>\#1-bit; \\
\hline & \} \\
\hline & \} \\
\hline & VR0 = [mem32]; \\
\hline
\end{tabular}

Flags \(\quad\) This instruction modifies the following bits in the VSTATUS register:
- OVFR is set if signed overflow is detected for add/sub calculation in which destination is VRxL
- OVFI is set if signed overflow is detected for add/sub calculation in which destination is VRxH

\section*{Pipeline}

This is a 1/1-cycle instruction. The VCFFT and VMOV operations are completed in one cycle.

Example See the example for VCFFT10 VR7, VR6, VR3, VR2, VR1, VR0, \#1-bit

\subsection*{2.5.8 Galois Instructions}

The instructions are listed alphabetically, preceded by a summary.

\section*{Table 2-17. Galois Field Instructions}
Title Page
VGFACC VRa, VRb, \#4-bit —Galois Field Instruction ..... 332
VGFACC VRa, VRb, VR7 -Galois Field Instruction ..... 333
VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32 — Galois Field Instruction with Parallel Load ..... 334
VGFADD4 VRa, VRb, VRc, \#4-bit —Galois Field Four Parallel Byte X Byte Add. ..... 335
VGFINIT mem16 - Initialize Galois Field Polynomial and Order ..... 336
VGFMAC4 VRa, VRb, VRc — Galois Field Four Parallel Byte X Byte Multiply and Accumulate ..... 337
VGFMPY4 VRa, VRb, VRc — Galois Field Four Parallel Byte X Byte Multiply ..... 338
VGFMPY4 VRa, VRb, VRc || VMOV32 VR0, mem32 - Galois Field Four Parallel Byte X Byte Multiply with Parallel Load ..... 339
VGFMAC4 VRa, VRb, VRc || PACK4 VR0, mem32, \#2-bit —Galois Field Four Parallel Byte X Byte Multiply and Accumulate with Parallel Byte Packing ..... 340
VPACK4 VRa, mem32, \#2-bit —Byte Packing ..... 341
VREVB VRa —Byte Reversal ..... 342
VSHLMB VRa, VRb —Shift Left and Merge Right Bytes ..... 343

\section*{VGFACC VRa, VRb, \#4-bit Galois Field Instruction}

Operands
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\#4-bit & 4-bit Immediate Value \\
\hline
\end{tabular}

Opcode
\[
\begin{aligned}
& \text { LSW: } 1110011010000001 \\
& \text { MSW: } 0000 \text { 00aa abbb IIII }
\end{aligned}
\]

Description Performs the following sequence of operations
```

If (I[0:0] == 1 )
VRa[7:0] = VRa[7:0] ^ VRb[7:0]
If (I[1:1] == 1 )
VRa[7:0] = VRa[7:0] ^ VRb[15:8]
If (I[2:2] == 1 )
VRa[7:0] = VRa[7:0] ^ VRb[23:16]
If (I[3:3] == 1 )
VRa[7:0] = VRa[7:0] ^ VRb[31:24]

```

Flags
Pipeline

\section*{Example}

See also

This instruction does not affect any flags in the VSTATUS register
This is a single-cycle instruction
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
VGFACC VRa, VRb, VR7
VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32

\section*{VGFACC VRa, VRb, VR7 Galois Field Instruction}

Operands
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VR7 & General purpose register: VR7 \\
\hline
\end{tabular}
\begin{tabular}{lllll} 
Opcode & LSW: 11100110 & 10000001 \\
& MSW: 0000 & 0100 & \(00 a a\) & abbb
\end{tabular}

Description
Performs the following sequence of operations
```

If (VR7[0:0] == 1 )
VRa[7:0]=VRa[7:0]^ VRb[7:0]
If (VR7[1:1] == 1 )
VRa[7:0] = VRa[7:0] ^ VRb[15:8]
If (VR7[2:2] == 1)
VRa[7:0] = VRa[7:0] ^ VRb[23:16]
If (VR7[3:3] == 1 )
VRa[7:0] = VRa[7:0] ^ VRb[31:24]

```

Flags This instruction does not affect any flags in the VSTATUS register
Pipeline
Example
See also
This is a single-cycle instruction
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
VGFACC VRa, VRb, \#4-bit
VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32

\section*{VGFACC VRa, VRb, VR7 || VMOV32 VRc, mem32 Galois Field Instruction with Parallel Load}

Operands
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRc & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VR7 & General purpose register: VR7 \\
mem32 & Pointer to a 32-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Opcode & \[
\begin{aligned}
& \text { LSW: } 111000101011 \text { 011a } \\
& \text { MSW: aabb bccc mem32 }
\end{aligned}
\] \\
\hline \multirow[t]{10}{*}{Description} & Performs the following sequence of operations \\
\hline & If (VR7[0:0] == 1 ) \\
\hline & VRa[7:0] = VRa[7:0] ^ VRb [7:0] \\
\hline & If (VR7[1:1] == 1 ) \\
\hline & \(\operatorname{VRa}[7: 0]=\operatorname{VRa}[7: 0] \wedge \operatorname{VRb}[15: 8]\) \\
\hline & If (VR7[2:2] == 1 ) \\
\hline & VRa[7:0] = VRa[7:0] ^ VRb [23:16] \\
\hline & If (VR7[3:3] == 1 ) \\
\hline & \(\operatorname{VRa}[7: 0]=\operatorname{VRa}[7: 0] \wedge \operatorname{VRb}[31: 24]\) \\
\hline & VRc \(=\) [mem32] \\
\hline Flags & This instruction does not affect any flags in the VSTATUS register \\
\hline Pipeline & This is a \(1 / 1\)-cycle instruction. Both the VGFACC and VMOV32 operation complete in a single cycle. \\
\hline Example & See the Reed-Solomon algorithm implementation in the VCU library in controISUITE \\
\hline See also & VGFACC VRa, VRb, \#4-bit VGFACC VRa, VRb, VR7 \\
\hline
\end{tabular}

VGFADD4 VRa, VRb, VRc, \#4-bit Galois Field Four Parallel Byte X Byte Add
Operands
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRc & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\#4-bit & 4-bit Immediate Value \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011010000000 \\
& MSW: \(000 a\) aabb bccc IIII
\end{tabular}

Description Performs the following sequence of operations
If ( \([0: 0]==1\) )
\(\operatorname{VRa}[7: 0]=\operatorname{VRb}[7: 0] \wedge \operatorname{VRc}[7: 0]\)
else
\(\operatorname{VRa}[7: 0]=\operatorname{VRb}[7: 0]\)

If (I[1:1] == 1 )
\(\operatorname{VRa}[15: 8]=\operatorname{VRb}[15: 8] \wedge \operatorname{VRc}[15: 8]\)
else
\(\operatorname{VRa}[15: 8]=\operatorname{VRb}[15: 8]\)
If (I[2:2] == 1 )
\(\operatorname{VRa}[23: 16]=\operatorname{VRb}[23: 16] \wedge \operatorname{VRc}[23: 16]\)
else
\(\operatorname{VRa}[23: 16]=\operatorname{VRb}[23: 16]\)
If (I[3:3] == 1 )
\(\operatorname{VRa}[31: 24]=\operatorname{VRb}[31: 24] \wedge \operatorname{VRc}[31: 24]\)
else
\(\operatorname{VRa}[31: 24]=\operatorname{VRb}[31: 24]\)
Flags This instruction does not affect any flags in the VSTATUS register

\section*{Pipeline}

This is a single cycle instruction
Example
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
See also

\section*{VGFINIT mem16 Initialize Galois Field Polynomial and Order}

Operands
\begin{tabular}{ll}
\hline mem16 & Pointer to 16-bit memory location \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & \begin{tabular}{l} 
LSW: 1110001011000101 \\
MSW: 00000000 mem16
\end{tabular} \\
Description & Initialize GF Polynomial and Order \\
& VSTATUS [GFPoLY] \(=[\) mem16] [7:0] \\
& VSTATUS [GFORDER] \(=[\) mem16] [10:8]
\end{tabular}

Flags
Pipeline
Example See the Reed-Solomon algorithm implementation in the VCU library in controISUITE

\section*{VGFMAC4 VRa, VRb, VRc Galois Field Four Parallel Byte X Byte Multiply and Accumulate}

\section*{Operands}
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRc & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011010000000 \\
MSW: 0010 001a aabb bccc
\end{tabular}

Description Performs the follow sequence of operations:
\(\operatorname{VRa}[7: 0]=(\operatorname{VRa}[7: 0] \quad * \operatorname{VRb}[7: 0]) \wedge \operatorname{VRc}[7: 0]\)
\(\operatorname{VRa}[15: 8]=(\operatorname{VRa}[15: 8] * \operatorname{VRb}[15: 8]) \wedge \operatorname{VRc}[15: 8]\)
\(\operatorname{VRa}[23: 16]=(V R a[23: 16] * \operatorname{VRb}[23: 16])\) ^ VRc[23:16]
\(\operatorname{VRa}[31: 24]=(\operatorname{VRa}[31: 24] * \operatorname{VRb}[31: 24]) \wedge \operatorname{VRc}[31: 24]\)
The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags
Pipeline
This instruction does not affect any flags in the VSTATUS register
This is a single-cycle instruction
Example
See also
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
VGFMPY4 VRa, VRb, VRc || VMOV32 VR0, mem32

\section*{VGFMPY4 VRa, VRb, VRc Galois Field Four Parallel Byte X Byte Multiply}

Operands
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRc & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011010000000 \\
MSW: 0010 000a aabb bccc
\end{tabular}

Description

Flags
Pipeline
Example
See also

Performs the following sequence of operations
VRa[7:0] = VRb[7:0] * VRc[7:0]
\(\operatorname{VRa}[15: 8]=\operatorname{VRb}[15: 8]\) * \(\operatorname{VRc}[15: 8]\)
VRa[23:16] = VRb[23:16] * VRc[23:16]
VRa[31:24] = VRb[31:24] * VRc[31:24]
The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

This instruction does not affect any flags in the VSTATUS register
This is a single cycle instruction
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
VGFMPY4 VRa, VRb, VRc || VMOV32 VR0, mem32

\section*{VGFMPY4 VRa, VRb, VRc || VMOV32 VR0, mem32 Galois Field Four Parallel Byte X Byte Multiply} with Parallel Load

\section*{Operands}
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRc & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VR0 & General purpose register: VR0 \\
mem32 & Pointer to a 32-bit memory location \\
\hline
\end{tabular}
Opcode LSW: 111000101011 010a

MSW: aabb bccc mem32
Description Performs the following sequence of operations
\(\operatorname{VRa}[7: 0]=\operatorname{VRb}[7: 0] \quad * \operatorname{VRc}[7: 0]\)
\(\operatorname{VRa}[15: 8]=\operatorname{VRb}[15: 8] * \operatorname{VRc}[15: 8]\)
VRa[23:16] = VRb[23:16] * VRc[23:16]
VRa[31:24] = VRb[31:24] * VRc[31:24]
VRO = [mem32]
The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

\section*{Pipeline}

Example
See also

Flags This instruction does not affect any flags in the VSTATUS register
This is a 1/1-cycle instruction. Both the VGFMPY4 and VMOV32 operation complete in a single cycle.
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
VGFMPY4 VRa, VRb, VRc

VGFMAC4 VRa, VRb, VRc || PACK4 VR0, mem32, \#2-bit — Galois Field Four Parallel Byte X Byte Multiply and Accumulate with Parallel Byte Packing
VGFMAC4 VRa, VRb, VRc || PACK4 VR0, mem32, \#2-bit Galois Field Four Parallel Byte X Byte Multiply and Accumulate with Parallel Byte Packing

\section*{Operands}
\begin{tabular}{ll}
\hline VRb & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRc & General purpose register: VR0, VR1....VR7. Cannot be VR8 \\
VR0 & General purpose register: VR0 \\
mem32 & Pointer to 32-bit memory location \\
\#2-bit & 2-bit Immediate Value \\
\hline
\end{tabular}
Opcode \begin{tabular}{l} 
LSW: 111000101011 1IIa \\
MSW: aabb bccc mem32
\end{tabular}

Description Performs the follow sequence of operations:
```

VRa[7:0] = (VRa[7:0] * VRb[7:0]) ^ VRc[7:0]
VRa[15:8] = (VRa[15:8] * VRb[15:8]) ^ VRc[15:8]
VRa[23:16] = (VRa[23:16] * VRb[23:16]) ^ VRc[23:16]
VRa[31:24] = (VRa[31:24] * VRb[31:24]) ^ VRc[31:24]
If (I == 0)
VR0[7:0] = [mem32][7:0]
VR0[15:8] = [mem32][7:0]
VR0[23:16] = [mem32][7:0]
VR0[31:24] = [mem32][7:0]
Else If (I == 1)
VR0[7:0] = [mem32][15:8]
VR0[15:8] = [mem32][15:8]
VR0[23:16] = [mem32][15:8]
VR0[31:24] = [mem32][15:8]
Else If (I == 2)
VR0[7:0] = [mem32][23:16]
VR0[15:8] = [mem32][23:16]
VRO[23:16] = [mem32][23:16]
VR0[31:24] = [mem32][23:16]
Else If (I == 3)
VR0[7:0] = [mem32][31:24]
VR0[15:8] = [mem32][31:24]
VR0[23:16] = [mem32][31:24]
VR0[31:24] = [mem32][31:24]

```

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags This instruction does not affect any flags in the VSTATUS register
Pipeline \(\quad\) This is a 1/1-cycle instruction. Both the VGFMAC4 and PACK4 operations complete in a single cycle.

Example See the Reed-Solomon algorithm implementation in the VCU library in controlSUITE

\section*{See also}

\section*{VPACK4 VRa, mem32, \#2-bit Byte Packing}

\section*{Operands}
\begin{tabular}{ll}
\hline VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
mem32 & Pointer to a 32-bit memory location \\
\#2-bit & 2-bit Immediate Value \\
\hline
\end{tabular}

Opcode
```

LSW: 1110 0010 1011 0001
MSW: 000a aaII mem32

```

Description
Pack lth byte from a memory location 4 times in VRa
```

If (I == 0)
VRa[7:0] = [mem32][7:0]
VRa[15:8] = [mem32][7:0]
VRa[23:16] = [mem32][7:0]
VRa[31:24] = [mem32][7:0]
Else If (I == 1)
VRa[7:0] = [mem32][15:8]
VRa[15:8] = [mem32][15:8]
VRa[23:16] = [mem32][15:8]
VRa[31:24] = [mem32][15:8]
Else If (I == 2)
VRa[7:0] = [mem32][23:16]
VRa[15:8] = [mem32][23:16]
VRa[23:16] = [mem32][23:16]
VRa[31:24] = [mem32][23:16]
Else If (I == 3)
VRa[7:0] = [mem32][31:24]
VRa[15:8] = [mem32][31:24]
VRa[23:16] = [mem32][31:24]
VRa[31:24] = [mem32][31:24]

```

The GF multiply operation is defined by VSTATUS[GFPOLY] and VSTATUS[GFORDER] bits.

Flags This instruction does not affect any flags in the VSTATUS register

\section*{Pipeline}

Example
This is a single-cycle instruction
See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
See also

\section*{VREVB VRa}

\section*{Byte Reversal}

Operands
VRa General purpose register: VR0, VR1...VR7. Cannot be VR8
\begin{tabular}{ll} 
Opcode & LSW: 1110011010000000 \\
MSW: 0010 01000000 Oaaa \\
Description & Reverse Bytes \\
& Input: VRa \(=\{\mathrm{B} 3, \mathrm{~B} 2, \mathrm{~B} 1, \mathrm{~B} 0\}\) \\
& Output: VRa \(=\{\mathrm{B} 0, \mathrm{~B} 1, \mathrm{~B} 2, \mathrm{~B} 3\}\)
\end{tabular}

\section*{Flags}

Pipeline
Example

See also

\section*{VSHLMB VRa, VRb Shift Left and Merge Right Bytes}

\section*{Operands}
\begin{tabular}{ll}
\hline VRa & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
VRb & General purpose register: VR0, VR1...VR7. Cannot be VR8 \\
\hline
\end{tabular}
\begin{tabular}{ll} 
Opcode & LSW: 1110011010000000 \\
MSW: 0010010001 aa abbb \\
Description & Shift Left and Merge Bytes \\
& Input: VRa \(=\{\mathrm{B} 7, \mathrm{~B} 6, \mathrm{~B} 5, \mathrm{~B} 4\}\) \\
& Input: \(\mathrm{VRb}=\{\mathrm{B} 3, \mathrm{~B} 2, \mathrm{~B} 1, \mathrm{~B} 0\}\) \\
& Output: \(\mathrm{VRa}=\{\mathrm{B} 6, \mathrm{~B} 5, \mathrm{~B} 4, \mathrm{~B} 3\}\) \\
& Output: VRb \(=\left\{\mathrm{B} 2, \mathrm{~B} 1, \mathrm{~B} 0,8^{\prime} \mathrm{b} 0\right\}\)
\end{tabular}

Restrictions VRa!= VRb. The source and destination registers must be different
Flags This instruction does not affect any flags in the VSTATUS register
Pipeline This is a single-cycle instruction
Example See the Reed-Solomon algorithm implementation in the VCU library in controISUITE
See also InSTRUMENTS

\subsection*{2.5.9 Viterbi Instructions}

The instructions are listed alphabetically, preceded by a summary.

\section*{Table 2-18. Viterbi Instructions}
Title Page
VITBM2 VRO —Code Rate 1:2 Branch Metric Calculation ..... 345
VITBM2 VR0, mem32 —Branch Metric Calculation CR=1/2 ..... 346
VITBM2 VR0 || VMOV32 VR2, mem32 - Code Rate 1:2 Branch Metric Calculation with Parallel Load ..... 347
VITBM3 VR0, VR1, VR2 - Code Rate 1:3 Branch Metric Calculation ..... 348
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32 —Code Rate 1:3 Branch Metric Calculation with Parallel Load ..... 349
VITBM3 VR0L, VR1L, mem16 —Branch Metric Calculation CR=1/3 ..... 350
VITDHADDSUB VR4, VR3, VR2, VRa - Viterbi Double Add and Subtract, High ..... 351
VITDHADDSUB VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Add and Subtract High with Parallel Store ..... 353
VITDHSUBADD VR4, VR3, VR2, VRa - Viterbi Add and Subtract Low ..... 354
VITDHSUBADD VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Subtract and Add, High with Parallel Store ..... 355
VITDLADDSUB VR4, VR3, VR2, VRa - Viterbi Add and Subtract Low ..... 356
VITDLADDSUB VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Add and Subtract Low with Parallel Load ..... 357
VITDLSUBADD VR4, VR3, VR2, VRa - Viterbi Subtract and Add Low ..... 358
VITDLSUBADD VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Subtract and Add, Low with Parallel Store ..... 359
VITHSEL VRa, VRb, VR4, VR3 - Viterbi Select High ..... 360
VITHSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 - Viterbi Select High with Parallel Load ..... 361
VITLSEL VRa, VRb, VR4, VR3 - Viterbi Select, Low Word ..... 362
VITLSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 - Viterbi Select Low with Parallel Load ..... 363
VITSTAGE —Parallel Butterfly Computation ..... 365
VITSTAGE || VITBM2 VR0, mem32 —Parallel Butterfly Computation with Parallel Branch Metric Calculation CR=1/2 ..... 366
VITSTAGE || VMOV16 VR0L, mem16 —Parallel Butterfly Computation with Parallel Load ..... 368
VMOV32 VSM (k+1):VSM(k), mem32 -Load Consecutive State Metrics ..... 369
VMOV32 mem32, VSM (k+1):VSM(k) —Store Consecutive State Metrics ..... 370
VSETK \#3-bit —Set Constraint Length for Viterbi Operation ..... 371
VSMINIT mem16 - State Metrics Register initialization ..... 372
VTCLEAR — Clear Transition Bit Registers ..... 373
VTRACE mem32, VR0, VT0, VT1 — Viterbi Traceback, Store to Memory ..... 374
VTRACE VR1, VR0, VT0, VT1 — Viterbi Traceback, Store to Register ..... 376
VTRACE VR1, VR0, VT0, VT1 || VMOV32 VT0, mem32 -Trace-back with Parallel Load ..... 378

\section*{VITBM2 VR0}

Operands

\section*{Opcode}

Description

Flags

\section*{Pipeline}

\section*{Example}

See also

\section*{Code Rate 1:2 Branch Metric Calculation}

Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16 -bits.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VROL & 16-bit decoder input 0 \\
VROH & 16-bit decoder input 1 \\
\hline
\end{tabular}

The result of the operation is also stored in VRO as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VROL & 16 -bit branch metric \(0=\) VROL + VROH \\
VROH & 16 -bit branch metric \(1=\) VROL - VROL \\
\hline
\end{tabular}

LSW: 1110010100001100
Branch metric calculation for code rate \(=1 / 2\).
```

// SAT is vStATUS[SAT]
// VROL is decoder input 0
// VROH is decoder input 1
//
// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction
//
VROL = VROL + VROH; // VROL = branch metric 0
VROH = VROL - VROL; // VROH = branch metric 1
if (SAT == 1)
{
sat16(VROL);
sat16(VROH);
}

```

This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

This is a single-cycle instruction.

VITBM2 VR0 || VMOV32 VR2, mem32 VITBM3 VR0, VR1, VR2

\section*{VITBM2 VR0, mem32 Branch Metric Calculation CR=1/2}
\begin{tabular}{|c|c|}
\hline Operands & Before the operation, the inputs are loaded into the registers as shown bel \\
\hline \multirow[t]{2}{*}{Opcode} & LSW: 1110001010000000 \\
\hline & MSW: 00000001 mem16 \\
\hline \multirow[t]{8}{*}{Description} & Calculates two Branch-Metrics (BMs) for CR = 1/2 \\
\hline & If (VSTATUS [SAT] == 1) \{ \\
\hline & VROL \(=\) sat ([mem32][15:0] + [mem32][31:16]); \\
\hline & \(\mathrm{VROH}=\operatorname{sat}([\) mem32][15:0] - [mem32][31:16]); \\
\hline & \}else \{ \\
\hline & VROL \(=\) [mem32][15:0] + [mem32][31:16]; \\
\hline & VROH \(=\) [mem32][15:0] - [mem32][31:16]; \\
\hline & \} \\
\hline Flags & \begin{tabular}{l}
This instruction modifies the following bits in the VSTATUS register: \\
- OVFR is set if overflow is detected in the computation of 16-bit signed
\end{tabular} \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{18}{*}{Example} & ; \\
\hline & ; Viterbi \(\mathrm{K}=4 \mathrm{CR}=1 / 2\) \\
\hline & ; \\
\hline & ; etc ... \\
\hline & ; \\
\hline & VSETK \#CONSTRAINT_LENGTH ; Set constraint length \\
\hline & MOV AR1, \#SMETRICINIT_OFFSET \\
\hline & VSMINIT *+XAR4[AR1] ; Initialize the state metrics \\
\hline & MOV AR1, \#NBITS_OFFSET \\
\hline & MOV AL, *+XAR4[AR1] \\
\hline & LSR AL, 2 \\
\hline & SUBB AL, \#2 \\
\hline & MOV AR3, AL \(\quad\); Initialize the BMSEL register \\
\hline & MOVL XAR6, *+XAR4[BMSELINIT_OFFSET] \\
\hline & VMOV32 VR2, *XAR6 \(\quad\); Initialize BMSEL for \\
\hline & VITBM2 VRO, *XARO++ \(\quad\); Calculate and store BMs in \\
\hline & ; \\
\hline & ; etc ... \\
\hline \multirow[t]{3}{*}{See also} & VITBM2 VR0 \\
\hline & VITBM2 VR0 || VMOV32 VR2, mem32 \\
\hline & VITSTAGE_VITBM2_VR0_mem32 \\
\hline
\end{tabular}

INSTRUMENTS

\section*{VITBM2 VR0 || VMOV32 VR2, mem32 Code Rate 1:2 Branch Metric Calculation with Parallel Load}

Operands Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VROL & 16-bit decoder input 0 \\
VROH & 16-bit decoder input 1 \\
[mem32] & pointer to 32-bit memory location. \\
\hline
\end{tabular}

The result of the operation is stored in VR0 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VROL & 16 -bit branch metric \(0=\) VROL + VROH \\
VROH & 16 -bit branch metric \(1=\) VROL - VROL \\
VR2 & contents of memory pointed to by [mem32] \\
\hline
\end{tabular}

Opcode
```

LSW: 1110 0011 1111 1100
MSW: 0000 0000 mem32

```

\section*{Description}

Branch metric calculation for a code rate of \(1 / 2\) with parallel register load.
```

// SAT is vSTATUS[SAT]
// VROL is decoder input 0
// VROH is decoder input 1
//
// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction
//
VROL = VROL + VROH; // VROL = branch metric 0
VROH = VROL - VROL; // VROH = branch metric 1
if (SAT == 1)
{
sat16(VROL);
sat16(VR0H);
}
VR2 = [mem32] // Load VR2L and VR2H with the next state metrics

```

Flags This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow or underflow.

Pipeline
Both operations complete in a single cycle.

\section*{Example}

See also
VITBM2 VR0
VITBM3 VR0, VR1, VR2
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32

\section*{VITBM3 VR0, VR1, VR2 Code Rate 1:3 Branch Metric Calculation}

Operands

\section*{Opcode}

Description
\begin{tabular}{ll} 
Flags & \begin{tabular}{l} 
This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow \\
or underflow.
\end{tabular} \\
Pipeline & \begin{tabular}{l} 
This is a 2p-cycle instruction. The instruction following VITBM3 must not use VR0 or \\
VR1.
\end{tabular} \\
Example & \begin{tabular}{l} 
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
\end{tabular} \\
See also & \begin{tabular}{l} 
VITBM2 VR0 \\
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32 \\
VITBM2 VRO || VMOV32 VR2, mem32
\end{tabular}
\end{tabular}
\begin{tabular}{l} 
www.ti.com VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32 - Code Rate 1:3 Branch Metric Calculation with \\
\hline VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32 Code Rate 1:3 Branch Metric Calculation with \\
\(\underset{\text { Parallel Load }}{ }\)
\end{tabular}

\section*{Operands}

Before the operation, the inputs are loaded into the registers as shown below. Each operand for the branch metric calculation is 16-bits.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR0L & 16-bit decoder input 0 \\
VR1L & 16-bit decoder input 1 \\
[mem32] & pointer to a 32-bit memory location \\
\hline
\end{tabular}

The result of the operation is stored in VR0 and VR1 and VR2 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR0L & 16-bit branch metric \(0=\) VROL + VR1L + VR2L \\
VR0H & 16-bit branch metric \(1=\) VROL + VR1L - VR2L \\
VR1L & 16-bit branch metric \(2=\) VROL - VR1L + VR2 \\
VR1H & 16-bit branch metric \(3=\) VR0L - VR1L - VR2L \\
VR2 & Contents of the memory pointed to by [mem32] \\
\hline
\end{tabular}

\section*{Opcode}

Description Calculate the four branch metrics for a code rate of \(1 / 3\) with parallel register load.
```

// SAT is VSTATUS[SAT]
// VROL is decoder input 0
// VR1L is decoder input 1
// VR2L is decoder input 2
//
// Calculate the branch metrics by performing 16-bit signed
// addition and subtraction
//
VROL = VROL + VR1L + VR2L; // VROL = branch Metric 0
VROH = VROL + VR1L - VR2L; // VROH = branch Metric 1
VR1L = VROL - VR1L + VR2L; // VR1L = branch Metric 2
VR1H = VR0L - VR1L - VR2L; // VR1H = branch Metric 3
if(SAT == 1)
{
sat16(VR0L);
sat16(VR0H);
sat16(VR1L);
sat16(VR1H);
}
VR2 = [mem32];

```
\begin{tabular}{ll} 
Flags & \begin{tabular}{l} 
This instruction sets the real overflow flag, VSTATUS[OVFR] in the event of an overflow \\
or underflow.
\end{tabular} \\
Pipeline & \begin{tabular}{l} 
This is a 2p/1-cycle instruction. The VBITM3 operation takes 2p cycles and the VMOV32 \\
completes in a single cycle. The next instruction must not use VR0 or VR1.
\end{tabular} \\
Example & \begin{tabular}{l} 
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
\end{tabular} \\
See also & \begin{tabular}{l} 
VITBM2 VR0 \\
VITBM2 VRO \(\|\) VMOV32 VR2, mem32
\end{tabular}
\end{tabular}

\section*{VITBM3 VR0L, VR1L, mem16 Branch Metric Calculation CR=1/3}

\section*{Operands}
\begin{tabular}{ll}
\hline Input & Output \\
\hline VR0L & Low word of the general purpose register VR0 \\
VR1L & Low word of the general purpose register VR1 \\
mem16 & Pointer to 16-bit memory location \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 0010 1100 0101
MSW: 0000 0010 mem16

```

Calculates four Branch-Metrics (BMs) for CR \(=1 / 3\)
```

If(VSTATUS[SAT] == 1){
VROL = sat(VROL + VR1L + [mem16]);
VROH = sat(VROL + VR1L - [mem16]);
VR1L = sat(VR0L - VR1L + [mem16]);
VR1H = sat(VR0L - VR1L - [mem16]);
}else {
VROL = VROL + VR1L + [mem16];
VR0H = VROL + VR1L - [mem16];
VR1L = VR0L - VR1L + [mem16];
VR1H = VROL - VR1L - [mem16];
}

```

Flags This instruction modifies the following bits in the VSTATUS register.
- OVFR is set if overflow is detected in the computation of a 16-bit signed result

Pipeline
Example
See also

This is a single-cycle instruction.
See the example for VITSTAGE || VMOV16 VROL, mem16
VITBM3
VITBM3 VR0, VR1, VR2 || VMOV32 VR2, mem32

InsTRUMENTS

\section*{VITDHADDSUB VR4, VR3, VR2, VRa Viterbi Double Add and Subtract, High}

Operands

\section*{Opcode}

Description

Flags
Pipeline

\section*{Example}
```

Example Viterbi decoder code fragment
Viterbi butterfly calculations
; Loop once for each decoder input pair
; Branch metrics = BMO and BM1
; XAR5 points to the input stream to the decoder

```

```

_loop:
VMOV32 VR0, *XAR5++ ; Load two inputs into VR0L, VR0H
VITBM2 VR0 ; VROL = BM0 VROH = BM1
|| VMOV32 VR2, *XAR1++ ; Load previous state metrics
;
2 cycle Viterbi butterfly
;
VITDLADDSUB VR4,VR3,VR2,VR0 ; Perform add/sub
VITLSEL VR6,VR5,VR4,VR3 ; Perform compare/select
|| VMOV32 VR2, *XAR1++ ; Load previous state metrics
2 cycle Viterbi butterfly, next stage
VITDHADDSUB VR4,VR3,VR2,VR0

```

VITHSEL VR6, VR5,VR4,VR3
|| VMOV32 VR2, *XAR1++
;
; 2 cycle Viterbi butterfly, next stage
;
VITDLADDSUB VR4,VR3,VR2,VR0
|| VMOV32 *XAR2++, VR5
. . .

See also
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa VITDLSUBADD VR4, VR3, VR2, VRa

\begin{tabular}{ll} 
See also & VITDHSUBADD VR4, VR3, VR2, VRa \\
& VITDLADDSUB VR4, VR3, VR2, VRa \\
& VITDLSUBADD VR4, VR3, VR2, VRa
\end{tabular}

\section*{VITDHSUBADD VR4, VR3, VR2, VRa Viterbi Add and Subtract Low}

Operands

\section*{Opcode}

\section*{Description}

Flags
Pipeline
Example
See also

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR2L & 16-bit state metric 0 \\
VR2H & 16-bit state metric 1 \\
VRaL & Branch metric 0. VRa must be VR0 or VR1. \\
\hline
\end{tabular}

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR3L & 16-bit path metric \(0=\) VR2L - VRaH \\
VR3H & 16-bit path metric \(1=\) VR2H + VRaH \\
VR4L & 16-bit path metric \(2=\) VR2L + VRaH \\
VR4H & 16-bit path metric \(3=\) VR2H - VRaL \\
\hline
\end{tabular}

LSW: 111001011111 aaaa
This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.
```

//
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaL with the branch metric.
//
VR3L = VR2L - VRaL // Path metric 0
VR3H = VR2H + VRaL // Path metric 1
VR4L = VR2L + VRaL // Path metric 2
VR4H = VR2H - VRaL // Path metric 3

```

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa VITDLSUBADD VR4, VR3, VR2, VRa

\section*{VITDHSUBADD VR4, VR3, VR2, VRa || VMOV32 mem32, VRb Viterbi Subtract and Add, High with Parallel Store}

Operands
Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaH.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR2L & 16-bit state metric 0 \\
VR2H & 16-bit state metric 1 \\
VRaH & Branch metric 1. VRa must be VR0 or VR1. \\
VRb & Contents to be stored. VRb can be VR5, VR6, VR7 or VR8. \\
\hline
\end{tabular}

The result of the operation is stored in VR3 and VR4 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR3L & 16-bit path metric \(0=\) VR2L -VRaH \\
VR3H & 16-bit path metric \(1=\) VR2H + VRaH \\
VR4L & 16-bit path metric \(2=\) VR2L + VRaH \\
VR4H & 16-bit path metric \(3=\) VR2H - VRaH \\
[mem32] & Contents of VRb. VRb can be VR5, VR6, VR7 or VR8. \\
\hline
\end{tabular}

\section*{Opcode}

Description Viterbi high subtract and add. This instruction is used to calculate four path metrics.
\(1 /\)
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
[mem32] = VRb // Store VRb to memory
VR3L = VR2L - VRaH // Path metric 0
VR3H \(=\) VR2H + VRaH // Path metric 1
VR4L = VR2L + VRaH // Path metric 2
VR4H \(=\) VR2H - VRaH // Path metric 3
Flags \(\quad\) This instruction does not modify any flags in the VSTATUS register.
Pipeline
This is a single-cycle instruction.

\section*{Example}

See also
VITDHADDSUB VR4, VR3, VR2, VRa
VITDLADDSUB VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa

\section*{VITDLADDSUB VR4, VR3, VR2, VRa Viterbi Add and Subtract Low}

Operands

\section*{Opcode}

\section*{Description}

\section*{Flags}

Pipeline
Example
See also

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR2L & 16-bit state metric 0 \\
VR2H & 16-bit state metric 1 \\
VRaL & Branch metric 0. VRa must be VR0 or VR1. \\
\hline
\end{tabular}

The result of the operation is 4 path metrics stored in VR3 and VR4 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR3L & 16-bit path metric \(0=\) VR2L + VRaH \\
VR3H & 16-bit path metric \(1=\) VR2H - VRaH \\
VR4L & 16-bit path metric \(2=\) VR2L - VRaH \\
VR4H & 16-bit path metric \(3=\) VR2H + VRaL \\
\hline
\end{tabular}

LSW: 111001010011 aaaa
This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.
//
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaL with the branch metric.
//
VR3L = VR2L + VRaL // Path metric 0
VR3H = VR2H - VRaL // Path metric 1
VR4L = VR2L - VRaL // Path metric 2
VR4H = VR2H + VRaL // Path metric 3
This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa
VITDLSUBADD VR4, VR3, VR2, VRa


\section*{Opcode}

\section*{Description}

Flags This instruction does not modify any flags in the VSTATUS register.

\section*{Pipeline}

This is a single-cycle instruction.
Example
See also
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITDHADDSUB VR4, VR3, VR2, VRa VITDHSUBADD VR4, VR3, VR2, VRa VITDLSUBADD VR4, VR3, VR2, VRa

\section*{VITDLSUBADD VR4, VR3, VR2, VRa Viterbi Subtract and Add Low}

Operands

\section*{Opcode}

\section*{Description}

Flags
Pipeline
Example
See also

Before the operation, the inputs are loaded into the registers as shown below. This operation uses the branch metric stored in VRaL.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR2L & 16-bit state metric 0 \\
VR2H & 16-bit state metric 1 \\
VRaL & Branch metric 0. VRa must be VR0 or VR1. \\
\hline
\end{tabular}

The result of the operation is four path metrics stored in VR3 and VR4 as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR3L & 16-bit path metric \(0=\) VR2L - VRaH \\
VR3H & 16-bit path metric \(1=\) VR2H + VRaH \\
VR4L & 16-bit path metric \(2=\) VR2L + VRaH \\
VR4H & 16-bit path metric \(3=\) VR2H - VRaL \\
\hline
\end{tabular}

LSW: 111001011110 aaaa
This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.
```

//
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
VR3L = VR2L - VRaL // Path metric 0
VR3H = VR2H + VRaL // Path metric 1
VR4L = VR2L + VRaL // Path metric 2
VR4H = VR2H - VRaL // Path metric 3

```

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITDHADDSUB VR4, VR3, VR2, VRa
VITDHSUBADD VR4, VR3, VR2, VRa VITDLADDSUB VR4, VR3, VR2, VRa
\begin{tabular}{lll} 
www.ti.com & VITDLSUBADD VR4, VR3, VR2, VRa || VMOV32 mem32, VRb — Viterbi Subtract and Add, Low with Parallel \\
Store
\end{tabular}

\section*{Opcode}

\section*{Description}
```

LSW: 1110 00100000 1010
MSW: bbbb aaaa mem32

```

This instruction is used to calculate four path metrics in the Viterbi butterfly. This operation uses the branch metric stored in VRaL.
```

//
// Calculate the four path metrics by performing 16-bit signed
// addition and subtraction
//
// Before this operation VR2L and VR2H are loaded with the state
// metrics and VRaH with the branch metric.
//
[mem32] = VRb // Store VRb into mem32
VR3L = VR2L - VRaL // Path metric 0
VR3H = VR2H + VRaL // Path metric 1
VR4L = VR2L + VRaL // Path metric 2
VR4H = VR2H - VRaL // Path metric 3

```

Flags This instruction does not modify any flags in the VSTATUS register.

\section*{Pipeline}

This is a single-cycle instruction.
Example
See also
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITDHADDSUB VR4, VR3, VR2, VRa VITDHSUBADD VR4, VR3, VR2, VRa VITDLADDSUB VR4, VR3, VR2, VRa

\section*{VITHSEL VRa, VRb, VR4, VR3 Viterbi Select High}

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR3L & 16-bit path metric 0 \\
VR3H & 16-bit path metric 1 \\
VR4L & 16-bit path metric 2 \\
VR4H & 16-bit path metric 3 \\
\hline
\end{tabular}

The result of the operation is the new state metrics stored in VRa and VRb as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VRaH & 16-bit state metric 0. VRa can be VR6 or VR8. \\
VRbH & 16-bit state metric 1. VRb can be VR5 or VR7. \\
VT0 & The transition bit is appended to the end of the register. \\
VT1 & The transition bit is appended to the end of the register. \\
\hline
\end{tabular}

Opcode

Description

Flags
Pipeline
Example
See also
```

LSW: 1110 0110 1111 0111
MSW: 0000 0000 bbbb aaaa

```

This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16 bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITLSEL instruction.
```

T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
VRbH = VR3L; // New state metric 0
TO[0:0] = 0; // Store the transition bit
}
else
{
VRbH = VR3H; // New state metric 0
TO[0:0] = 1; // Store the transition bit
}
T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
VRaH = VR4L; // New state metric 1
T1[0:0] = 0; // Store the transition bit
}
else
{
VRaH = VR4H; // New state metric 1
T1[0:0] = 1; // Store the transition bit
}

```

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
VITLSEL VRa, VRb, VR4, VR3

InSTRUMENTS

\section*{VITHSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 Viterbi Select High with Parallel Load}

Operands Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR3L & 16-bit path metric 0 \\
VR3H & 16-bit path metric 1 \\
VR4L & 16-bit path metric 2 \\
VR4H & 16-bit path metric 3 \\
[mem32] & pointer to 32-bit memory location. \\
\hline
\end{tabular}

The result of the operation is the new state metrics stored in VRa and VRb as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VRaH & 16-bit state metric 0. VRa can be VR6 or VR8. \\
VRbH & 16-bit state metric 1. VRb can be VR5 or VR7. \\
VT0 & The transition bit is appended to the end of the register. \\
VT1 & The transition bit is appended to the end of the register. \\
VR2 & Contents of the memory pointed to by [mem32]. \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 0011 1111 1111
MSW: bbbb aaaa mem32

```

\section*{Description}

Flags This instruction does not modify any flags in the VSTATUS register.
Pipeline This is a single-cycle instruction.
Example Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
See also VITLSEL VRa, VRb, VR4, VR3

\section*{VITLSEL VRa, VRb, VR4, VR3 Viterbi Select, Low Word}

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR3L & 16-bit path metric 0 \\
VR3H & 16-bit path metric 1 \\
VR4L & 16-bit path metric 2 \\
VR4H & 16-bit path metric 3 \\
\hline
\end{tabular}

The result of the operation is the new state metrics stored in VRa and VRb as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VRaL & 16-bit state metric 0. VRa can be VR6 or VR8. \\
VRbL & 16-bit state metric 1. VRb can be VR5 or VR7. \\
VT0 & The transition bit is appended to the end of the register. \\
VT1 & The transition bit is appended to the end of the register. \\
\hline
\end{tabular}

Opcode

Description
```

LSW: 1110 0110 1111 0110
MSW: 0000 0000 bbbb aaaa

```

This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16 -bits of the VRa and VRb registers. To instead load the state metrics into the low 16-bits use the VITHSEL instruction.
```

T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
{
VRbL = VR3L; // New state metric 0
TO[0:0] = 0; // Store the transition bit
}
else
{
VRbL = VR3H; // New state metric 0
TO[0:0] = 1; // Store the transition bit
}
T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
{
VRaL = VR4L; // New state metric 1
T1[0:0] = 0; // Store the transition bit
}
else
{
VRaL = VR4H; // New state metric 1
T1[0:0] = 1; // Store the transition bit
}

```

Flags This instruction does not modify any flags in the VSTATUS register.
Pipeline This is a single-cycle instruction.
Example
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.
See also VITHSEL VRa, VRb, VR4, VR3

InsTRUMENTS

\section*{VITLSEL VRa, VRb, VR4, VR3 || VMOV32 VR2, mem32 Viterbi Select Low with Parallel Load}

Operands
Before the operation, the path metrics are loaded into the registers as shown below. Typically this will have been done using a Viterbi AddSub or SubAdd instruction.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VR3L & 16-bit path metric 0 \\
VR3H & 16-bit path metric 1 \\
VR4L & 16-bit path metric 2 \\
VR4H & 16-bit path metric 3 \\
mem32 & Pointer to 32-bit memory location. \\
\hline
\end{tabular}

The result of the operation is the new state metrics stored in VRa and VRb as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VRaL & 16-bit state metric 0. VRa can be VR6 or VR8. \\
VRbL & 16-bit state metric 1. VRb can be VR5 or VR7. \\
VT0 & The transition bit is appended to the end of the register. \\
VT1 & The transition bit is appended to the end of the register. \\
VR2 & Contents of 32-bit memory pointed to by mem32. \\
\hline
\end{tabular}

\section*{Opcode}
```

LSW: 1110 0011 1111 1110
MSW: bbbb aaaa mem32

```
```

Description

```
```

T1 = T1 << 1 // Shift previous transition bits left
if (VR4L > VR4H)
\{
VRaL = VR4L; // New state metric 1
$\mathrm{T} 1[0: 0]=0 ; / /$ Store the transition bit
\}
else
\{
VRaL = VR4H; // New state metric 1
T1[0:0] = 1; // Store the transition bit
\}
VR2 = [mem32]
This instruction computes the new state metrics of a Viterbi butterfly operation and stores them in the higher 16 -bits of the VRa and VRb registers. To instead load the state metrics into the low 16 -bits use the VITHSEL instruction. In parallel the VR2 register is loaded with the contents of memory pointed to by [mem32].

```

Flags
Pipeline
Example
```

This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.
T0 = T0 << 1 // Shift previous transition bits left
T0 = T0 << 1 // Shift previous transition bits left
T0 = T0 << 1 // Shift previous transition bits left
if (VR3L > VR3H)
if (VR3L > VR3H)
if (VR3L > VR3H)
{
{
{
VRbL = VR3L; // New state metric 0
VRbL = VR3L; // New state metric 0
VRbL = VR3L; // New state metric 0
TO[0:0] = 0; // Store the transition bit
TO[0:0] = 0; // Store the transition bit
TO[0:0] = 0; // Store the transition bit
}
}
}
else
else
else
{
{
{
VRbL = VR3H; // New state metric 0
VRbL = VR3H; // New state metric 0
VRbL = VR3H; // New state metric 0
TO[0:0] = 1; // Store the transition bit
TO[0:0] = 1; // Store the transition bit
TO[0:0] = 1; // Store the transition bit
}
}
}
Refer to the example for VITDHADDSUB VR4, VR3, VR2, VRa.

```

\section*{VITSTAGE}

Operands

\section*{Opcode}

\section*{Description}

\section*{Parallel Butterfly Computation}

None
LSW: 1110010100100110
VITSTAGE instruction performs 32 viterbi butterflies in a single cycle. This instructions does the following:
- Depends on the Initial 64 State Metrics of the current stage stored in registers VSMO to VSM63
- Depends on the Branch Metrics Select configuration stored in registers VR2 to VR5
- Depends on the Computed Branch Metrics of the current stage stored in registers VR0 and VR1
- Computes the State Metrics for the next stage and updates registers VSMO to VSM63. The 16 -bit signed result of the computation is saturated if VSTATUS[SAT] == 1
- Computes transition bits for all 64 states and updates registers VT0 and VT1

Flags

\section*{Pipeline}

This instruction modifies the following bits in the VSTATUS register.
- OVFR is set if overflow is detected in the computation of a 16 -bit signed result

This is a single-cycle instruction.

\section*{Example}

See also
```

;
; Viterbi K=4 CR = 1/2
;
;etc ...
;
VSETK \#CONSTRAINT_LENGTH ; Set constraint length
MOV AR1, \#SMETRICINIT_OFFSET
VSMINIT *+XAR4[AR1] ; Initialize the state metrics
MOV AR1, \#NBITS_OFFSET
MOV AL, * + XAR4 [AR1]
LSR AL, 2
SUBB AL, \#2
MOV AR3, AL ; Initialize the BMSEL register
; for butterfly 0 to K-1
MOVL XAR6, *+XAR4 [BMSELINIT_OFFSET]
VMOV32 VR2, *XAR6 ; Initialize BMSEL for
VITBM2 VR0, *XAR0++ ; Calculate and store BMs in
; VROL and VROH
.align 2
RPTB _VITERBI_runK4CR12_stageAandB, AR3
_VITERBI_runK4CR12_stageA:
VITSTAGE ; Compute NSTATES/2 butterflies
VITBM2 VR0, *XAR0++ ; compute branch metrics for
; next butterfly
VMOV32 *XAR2++, VT1 ; Store VT1
VMOV32 *XAR2++, VT0 ; Store VT0
;
;etc ...
;
VITSTAGE || VITBM2 VR0, mem32
VITSTAGE || VMOV16 VROL, mem16

```
\begin{tabular}{|c|c|c|}
\hline \multicolumn{3}{|l|}{VITSTAGE || VITBM2 VR0, mem32 Parallel Butterfly Computation with Parallel Branch Metric Calculation CR=1/2} \\
\hline \multirow[t]{3}{*}{Operands} & Input & Output \\
\hline & VRo & Destination register \\
\hline & mem32 & Pointer to 32 -bit memory location \\
\hline
\end{tabular}

Opcode

\section*{Description}
\begin{tabular}{|c|c|}
\hline Flags & \begin{tabular}{l}
This instruction modifies the following bits in the VSTATUS register. \\
- OVFR is set if overflow is detected in the computation of a 16 -bit sign
\end{tabular} \\
\hline Pipeline & This is a single-cycle instruction. \\
\hline \multirow[t]{24}{*}{Example} & ; \\
\hline & ; Viterbi \(\mathrm{K}=4 \mathrm{CR}=1 / 2\) \\
\hline & ; \\
\hline & ; etc ... \\
\hline & ; \\
\hline & VSETK \#CONSTRAINT_LENGTH ; Set constraint length \\
\hline & MOV AR1, \#SMETRICINIT_OFFSET \\
\hline & VSMINIT *+XAR4[AR1] ; Initialize the state metrics \\
\hline & MOV AR1, \#NBITS_OFFSET \\
\hline & MOV AL, *+XAR4[AR1] \\
\hline & LSR AL, 2 \\
\hline & SUBB AL, \#2 \\
\hline & MOV AR3, AL \begin{tabular}{l}
; Initialize the BMSEL register \\
; for butterfly 0 to \(\mathrm{K}-1\)
\end{tabular} \\
\hline & MOVL XAR6, *+XAR4[BMSELINIT_OFFSET] \\
\hline & VMOV32 VR2, *XAR6 \(\quad\); Initialize BMSEL for \\
\hline & \begin{tabular}{ll} 
VITBM2 VRO, *XARO++ & ; Calculate and store BMs in \\
& ; VROL and VROH
\end{tabular} \\
\hline & .align 2 \\
\hline & RPTB _VITERBI_runK4CR12_stageAandB, AR3 \\
\hline & _VITERBI_runK4CR12_stageA: \\
\hline & VITSTAGE \(\quad\); Compute NSTATES/2 butterflies \\
\hline & ||VITBM2 VRO, *XAR0++ \(\quad\)\begin{tabular}{l}
; compute branch metrics for \\
; next butterfly
\end{tabular} \\
\hline & vMOV32 *XAR2++, VT1 ; Store VT1 \\
\hline & VMOV32 *XAR2++, VT0 ; Store VT0 \\
\hline & \\
\hline
\end{tabular}
www.ti.com \(\quad\) VITSTAGE || VITBM2 VR0, mem32 - Parallel Butterfly Computation with Parallel Branch Metric Calculation
\(C R=1 / 2\)

\section*{VITSTAGE || VMOV16 VR0L, mem16 Parallel Butterfly Computation with Parallel Load}

Operands
\begin{tabular}{ll}
\hline Input & Output \\
\hline VR0L & Low word of the destination register \\
mem16 & Pointer to 16-bit memory location \\
\hline
\end{tabular}

Opcode


\section*{VMOV32 VSM (k+1):VSM(k), mem32 Load Consecutive State Metrics}


\section*{VMOV32 mem32, VSM (k+1):VSM(k) Store Consecutive State Metrics}

Operands

Opcode

Description
\begin{tabular}{ll}
\hline Input & Output \\
\hline VSM( \(\mathrm{k}+1\) ):VSM(k) & Consecutive State Metric Registers (VSM1:VSM0 .... VSM63:VSM62) \\
mem32 & Pointer to 32-bit memory location \\
\hline
\end{tabular}

LSW: 1110001000001110 MSW: 000 n nnnn mem32

Store a pair of Consecutive State Metrics from memory:
```

[mem32] [31:16] = VSM(k+1);
[mem32] [15:0] = VSM(k);

```

\section*{NOTE:}
- \(n-k / 2\), used in opcode assignment
- \(k\) is always even

Flags

\section*{Pipeline}

Example

See also

This instruction does not affect any flags in the VSTATUS register.
This is a single-cycle instruction.
VMOV32 *XAR7++ VSM63: VSM62

VMOV32 VSM (k+1):VSM(k), mem32

VSETK \#3-bit
Operands
\begin{tabular}{ll}
\hline Input & Output \\
\hline \#3-bit & 3-bit immediate value \\
\hline
\end{tabular}

Opcode

> LSW: 1110011011110010
> MSW: 000010010000 0III

Description
Flags
Pipeline
VSTATUS[K] = \#3-bit Immediate
This instruction does not affect any flags in the VSTATUS register.
This is a single-cycle instruction.

\section*{Example}

See also

\section*{VSMINIT mem16 State Metrics Register initialization}

Operands
\begin{tabular}{ll}
\hline Input & Output \\
\hline mem16 & Pointer to 16-bit memory location \\
\hline
\end{tabular}

Opcode
```

LSW: 1111 0010 1100 0101
MSW: 0000 0001 mem16

```

Description
Initializes the state metric registers.
VSMO = 0
VSM1 to VSM63 = [mem16]
Flags
Pipeline
Example
This instruction does not affect any flags in the VSTATUS register.
This is a single-cycle instruction.

\section*{See also}

\section*{VTCLEAR}

Operands

Opcode

\section*{Description}

Flags
Pipeline

\section*{Example}

See also

\section*{Clear Transition Bit Registers}
none
LSW: 1110010100101001
Clear the VT0 and VT1 registers.
\(\mathrm{VTO}=0\);
VT1 \(=0\);
This instruction does not modify any flags in the VSTATUS register.
This is a single-cycle instruction.

VCLEARALL
VCLEAR VRa

\section*{VTRACE mem32, VR0, VT0, VT1 Viterbi Traceback, Store to Memory}

Operands
Before the operation, the path metrics are loaded into the registers as shown below using a Viterbi AddSub or SubAdd instruction.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VT0 & transition bit register 0 \\
VT1 & transition bit register 1 \\
VR0 & \begin{tabular}{l} 
Initial value is zero. After the first VTRACE, this contains information from the \\
previous trace-back.
\end{tabular} \\
\hline
\end{tabular}

The result of the operation is the new state metrics stored in VRa and VRb as shown below:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline [mem32] & Traceback result from the transition bits. \\
\hline
\end{tabular}

Opcode

\section*{Description}
```

LSW: 1110 00100000 1100
MSW: 0000 0000 mem32

```

Trace-back from the transition bits stored in VT0 and VT1 registers. Write the result to memory. The transition bits in the VT0 and VT1 registers are stored in the following format by the VITLSEL and VITHSEL instructions:
\begin{tabular}{ll} 
VT0[31] & Transition bit [State 0] \\
VT0[30] & Transition bit [State 1] \\
VT0[29] & Transition bit [State 2] \\
\(\ldots\) & ... \\
VT0[0] & Transition bit [State 31] \\
VT1[31] & Transition bit [State 32] \\
VT1[30] & Transition bit [State 33] \\
VT1[29] & Transition bit [State 34] \\
\(\ldots\) & ... \\
VT1[0] & Transition bit [State 63] \\
\hline
\end{tabular}
```

//
// Calculate the decoder output bit by performing a
// traceback from the transition bits stored in the VTO and VT1 registers
//
K = VSTATUS[K];
S = VR0[K-2:0];
VR0[31:K-1] = 0;
if (S < (1<< (K-2))){
temp[0] = VT0[(1 << (K-2))- 1 -S];
}else{
temp[0] = VT1[(1 << (K-1))- 1 -S];
}
[mem32][0] = temp;
[mem32][31:1] = 0;
VR0[K-2:0] = 2*VR0[K-2:0] + temp[0];

```

Flags This instruction does not modify any flags in the VSTATUS register.
Pipeline This is a single-cycle instruction.

\section*{Example}
```

//
// Example traceback code fragment

```

\section*{//}
// XAR5 points to the beginning of Decoder Output array
//
VCLEAR VR0
MOVL XAR5,*+XAR4[0]
//
// To retrieve each original message:
// Load VTO/VT1 with the stored transition values
// and use VTRACE instruction
//
VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++, VR0, VT0, VT1
VMOV32 VT0, *--XAR3
VMOV32 VT1, *--XAR3
VTRACE *XAR5++, VR0, VT0, VT1
...etc for each VT0/VT1 pair
See also
VTRACE VR1, VR0, VT0, VT1

\section*{VTRACE VR1, VR0, VT0, VT1 Viterbi Traceback, Store to Register}

Operands

\section*{Opcode}

Description

Flags This instruction does not modify any flags in the VSTATUS register.
Pipeline
Before the operation, the path metrics are loaded into the registers as shown below using a Viterbi AddSub or SubAdd instruction.
\begin{tabular}{ll}
\hline Input Register & Value \\
\hline VT0 & transition bit register 0 \\
VT1 & transiton bit register 1 \\
VR0 & Initial value is zero. After the first VTRACE, this contains information from the \\
& previous trace-back. \\
\hline
\end{tabular}

The result of the operation is the output of the decoder stored in VR1:
\begin{tabular}{ll}
\hline Output Register & Value \\
\hline VR1 & Traceback result from the transition bits. \\
\hline LSW: \(1110 \quad 010100101000\) \\
& \\
Trace-back from the transition bits stored in VT0 and VT1 registers. Write the result to \\
VR1. The transition bits in the VT0 and VT1 registers are stored in the following format \\
by the VITLSEL and VITHSEL instructions: \\
\hline VTO[31] & Transition bit [State 0] \\
VTO[30] & Transition bit [State 1] \\
VT0[29] & Transition bit [State 2] \\
\(\ldots\) & \(\ldots\) \\
VT0[0] & Transition bit [State 31] \\
VT1[31] & Transition bit [State 32] \\
VT1[30] & Transition bit [State 33] \\
VT1[29] & Transition bit [State 34] \\
\(\ldots\) & \(\ldots\) \\
VT1[0] & Transition bit [State 63]
\end{tabular}
```

//
// Calculate the decoder output bit by performing a
// traceback from the transition bits stored in the VTO and VT1 registers
//
K = VSTATUS[K];
S = VR0[K-2:0];
VR0[31:K-1] = 0;
if (S < (1<< (K-2))) {
temp[0] = VTO[(1<< (K-2))- 1 -S];
}else{
temp[0] = VT1[(1<< (K-1))- 1 -S];
}
if(VSTATUS [OPACK]==0) {
VR1 = VR1<<1;
VR1[0:0] = temp[0] ;
VR0[K-2:0] = 2*VR0[K-2:0] + temp[0];
}else{
VR1 = VR1>>1
VR1[31:31] = temp[0] ;
VR0[K-2:0] = 2*VR0[K-2:0] + temp[0];
}

```

This is a single-cycle instruction.

\section*{Example}

See also
VTRACE mem32, VR0, VT0, VT1

\section*{VTRACE VR1, VR0, VT0, VT1 || VMOV32 VT0, mem32 Trace-back with Parallel Load}
```

Operands

| Input Register | Value |
| :--- | :--- |
| VT0 | Traceback register |
| VT1 | Traceback register |
| VR0 | Decoded output bits register |
| VR1 | Decoded output bits register |
| mem32 | Pointer to 32-bit memory location |

```

\section*{Opcode}
```

LSW: 1110 0010 1011 0000

```
LSW: 1110 0010 1011 0000
MSW: 0000 0001 mem32
```

MSW: 0000 0001 mem32

```

\section*{Description}
```

Flags

```

\section*{Pipeline}

\section*{Example}
```

See also
VTRACE mem32, VR0, VT0, VT1

```

Texas
INSTRUMENTS

\subsection*{2.6 Rounding Mode}

This section details the rounding operation as applied to a right shift. When the rounding mode is enabled in the VSTATUS register, .5 will be added to the right shifted intermediate value before truncation. If rounding is disabled the right shifted value is only truncated. Table \(2-19\) shows the bit representation of two values, 11.0 and 13.0. The columns marked Bit-1, Bit-2 and Bit-3 hold temporary bits resulting from the right shift operation.

Table 2-19. Example: Values Before Shift Right
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline & Bit5 & Bit4 & Bit3 & Bit2 & Bit1 & Bit0 & Bit-1 & Bit-2 & Bit -3 & Value \\
\hline Val A & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 11.000 \\
\hline Val B & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 13.000 \\
\hline
\end{tabular}

Table 2-19 shows the intermediate values after the right shift has been applied to Val B. The columns marked Bit-1, Bit-2 and Bit-3 hold temporary bits resulting from the right shift operation.

Table 2-20. Example: Values after Shift Right
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline & Bit5 & Bit4 & Bit3 & Bit2 & Bit1 & Bit0 & Bit-1 & Bit-2 & Bit \(\mathbf{- 3}\) \\
Value \\
\hline Val A & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
11.000 \\
\hline Val B >>3 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
\hline
\end{tabular}

When the rounding mode is enabled, .5 will be added to the intermediate result before truncation. Table 2 21 shows the bit representation of \(\operatorname{Val} \mathrm{A}+\operatorname{Val}(\mathrm{B} \gg 3)\) operation with rounding. Notice .5 is added to the intermediate shifted right value. After the addition, the bits in Bit-1, Bit-2 and Bit-3 are removed. In this case the result of the operation will be 13 which is the truncated value after rounding.

Table 2-21. Example: Addition with Right Shift and Rounding
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline & Bit5 & Bit4 & Bit3 & Bit2 & Bit1 & Bit0 & Bit-1 & Bit-2 & Bit -3 \\
Value \\
\hline Val A & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline Val B >> 3 & 0 & 0 & 0 & 0 & 0 & 1 & 1.000 \\
\hline .5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline 1.625 \\
\hline Val A + Val B >>3+.5 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
\hline
\end{tabular}

When the rounding mode is disabled, the value is simply truncated. Table 2-22 shows the bit representation of the operation \(\mathrm{Val} \mathrm{A}+(\mathrm{Val} \mathrm{B} \mathrm{>>3)}\) without rounding. After the addition, the bits in Bit-1, Bit-2 and Bit-3 are removed. In this case the result of the operation will be 12 which is the truncated value without rounding.

Table 2-22. Example: Addition with Rounding After Shift Right
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|}
\hline & Bit5 & Bit4 & Bit3 & Bit2 & Bit1 & Bit0 & Bit-1 & Bit-2 & Bit -3 & Value \\
\hline Val A & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 11.000 \\
\hline Val B >>3 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1.625 \\
\hline Val A + Val B >>3 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 12.625 \\
\hline
\end{tabular}

Table 2-23 shows more examples of the intermediate shifted value along with the result if rounding is enabled or disabled. In each case, the truncated value is without .5 added and the rounded value is with .5 added.

Table 2-23. Shift Right Operation With and Without Rounding
\begin{tabular}{|c|c|c|cc|c|c|c|}
\hline Bit2 & Bit1 & Bit0 & Bit \(\mathbf{- 1}\) & Bit \(\mathbf{- 2}\) & Value & Result with RND = 0 & Result with RND = 1 \\
\hline 0 & 1 & 0 & 0 & 0 & 2.00 & 2 & 2 \\
\hline 0 & 0 & 1 & 1 & 1 & 1.75 & 1 & 2 \\
\hline 0 & 0 & 1 & 1 & 0 & 1.50 & 1 & 2 \\
\hline 0 & 0 & 1 & 0 & 1 & 1.25 & 1 & 1 \\
\hline 0 & 0 & 0 & 1 & 1 & 0.75 & 0 & 1 \\
\hline 0 & 0 & 0 & 1 & 0 & 0.50 & 0 & 1 \\
\hline 0 & 0 & 0 & 0 & 1 & 0.25 & 0 & 0 \\
\hline 0 & 0 & 0 & 0 & 0 & 0.00 & 0 & 0 \\
\hline 1 & 1 & 1 & 1 & 1 & -0.25 & 0 & 0 \\
\hline 1 & 1 & 1 & 1 & 0 & -0.50 & 0 & 0 \\
\hline 1 & 1 & 1 & 0 & 1 & -0.75 & 0 & -1 \\
\hline 1 & 1 & 1 & 0 & 0 & -1.00 & -1 & -1 \\
\hline 1 & 1 & 0 & 1 & 1 & -1.25 & -1 & -1 \\
\hline 1 & 1 & 0 & 1 & 0 & -1.50 & -1 & -1 \\
\hline 1 & 1 & 0 & 0 & 1 & -1.75 & -1 & -2 \\
\hline 1 & 1 & 0 & 0 & 0 & -2.00 & -2 & -2 \\
\hline
\end{tabular}

\section*{Trigonometric Math Unit (TMU)}

The Trigonometric Math Unit (TMU) is a fully programmable block that enhances the instruction set of the C28-FPU to more efficiently execute common trigonometric and arithmetic operations.
The TMU module described in this reference guide is a Type 0 TMU. For a list of all devices with a TMU module of the same type and to determine differences between the types, see the TMS320x28xx, 28xxx DSP Peripheral Reference Guide (SPRU566). This document describes the architecture and instruction set of the \(\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}+\mathrm{TMU}\).
Topic Page
3.1 Overview ..... 382
3.2 Components of the C28x+FPU Plus TMU ..... 382
3.3 Data Format ..... 382
3.4 Pipeline ..... 383
3.5 TMU Instruction Set ..... 388

\subsection*{3.1 Overview}

The TMU extends the capabilities of a C28x+FPU enabled processor by adding instructions to speed up the execution of common trigonometric and arithmetic operations listed in Table 3-1.

Table 3-1. TMU Supported Instructions
\begin{tabular}{|l|l|}
\hline Instructions & \(\mathbf{C}\) Equivalent Operation \\
\hline MPY2PIF32 RaH,RbH & \(\mathrm{a}=\mathrm{b}{ }^{*} 2 \mathrm{pi}\) \\
\hline DIV2PIF32 RaH,RbH & \(\mathrm{a}=\mathrm{b} / 2 \mathrm{pi}\) \\
\hline DIVF32 RaH,RbH,RcH & \(\mathrm{a}=\mathrm{b} / \mathrm{c}\) \\
\hline SQRTF32 RaH,RbH & \(\mathrm{a}=\operatorname{sqrt}(\mathrm{b})\) \\
\hline SINPUF32 RaH,RbH & \(\mathrm{a}=\sin \left(\mathrm{b}^{*} 2 \mathrm{pi}\right)\) \\
\hline COSPUF32 RaH,RbH & \(\mathrm{a}=\cos \left(\mathrm{b}^{*} 2 \mathrm{pi}\right)\) \\
\hline ATANPUF32 RaH,RbH & \(\mathrm{a}=\operatorname{atan}(\mathrm{b}) / 2 \mathrm{pi}\) \\
\hline QUADF32 RaH,RbH,RcH,RdH & Operation to assist in calculating ATANPU2 \\
\hline
\end{tabular}

\subsection*{3.2 Components of the C28x+FPU Plus TMU}

The TMU extends the capabilities of the C28x+FPU processors by adding new instructions and, in some cases, leveraging existing FPU instructions to carry out common arithmetic operations used in control applications.
No changes have been made to existing instructions, pipeline or memory bus architecture. All TMU instructions use the existing FPU register set (ROH to R7H) to carry out their operations. A detailed explanation of the workings of the FPU can be found in the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUEO2).

\subsection*{3.2.1 Interrupt Context Save and Restore}

Since the TMU uses the same register set and flags as the FPU, there are no special considerations with regards to interrupt context save and restore.
If a TMU operation is executing when an interrupt occurs, the C28 can initiate an interrupt context switch without affecting the TMU operation. The TMU will continue to process the operation to completion. Even though most TMU operations are multi-cycle, the TMU operation will have completed by the time register context save operations for the FPU are commenced. When restoring FPU registers, you must make sure that all TMU operations are completed before restoring any register used by another TMU operation.

\subsection*{3.3 Data Format}

The encoding of the floating-point formats is given in Table 3-2.
Table 3-2. IEEE 32-Bit Single Precision Floating-Point Format
\begin{tabular}{|c|c|c|c|}
\hline S32 & E32 (7:0) & M32 (22:0) & Value (V) \\
\hline 0 & 0 & 0 & Zero (V = 0) \\
\hline 1 & 0 & 0 & Negative Zero (V = -0) \\
\hline \[
\begin{aligned}
& 0+\mathrm{ve} \\
& 1 \text {-ve }
\end{aligned}
\] & 0 & non zero & De-normalized (V=(-1) \({ }^{\text {S** }} 2^{(-126) *}(0 . \mathrm{M})\) ) \\
\hline \[
\begin{aligned}
& 0+\mathrm{ve} \\
& 1 \text {-ve }
\end{aligned}
\] & 1 to 254 & 0 to 0x7FFFFF & Normal Range ( \(\mathrm{V}=(-1)^{\mathrm{S}}\) * \(2^{(\mathrm{E}-127)}\) * (1.M) ) \\
\hline 0 & 254 & 0x7FFFFF & Positive Max (V = +Max) \\
\hline 1 & 254 & 0x7FFFFF & Negative Max (V = - Max) \\
\hline 0 & max=255 & 0 & Positive Infinity ( \(\mathrm{V}=+\) Infinity \\
\hline 1 & max=255 & 0 & Negative Infinity ( \(\mathrm{V}=-\operatorname{lnfinity\text {)}}\) \\
\hline x & max \(=255\) & non zero & Not A Number ( \(\mathrm{V}=\mathrm{NaN}\) ) \\
\hline
\end{tabular}

INSTRUMENTS

The treatment of the various IEEE floating-point numerical formats for this TMU is the same as the FPU implementation given below:
Negative Zero: All TMU operations generate a positive ( \(\mathrm{S}==0\), \(\mathrm{E}==0, \mathrm{M}==0\) ) zero, never a negative zero if the result of the operation is zero. All TMU operations treat negative zero operations as zero.

De-Normalized Numbers: A de-normalized operand ( \(\mathrm{E}==0, \mathrm{M}!=0\) ) input is treated as zero \((\mathrm{E}==0, \mathrm{M}==0\) ) by all TMU operations. TMU operations never generate a de-normalized value.
Underflow: Underflow occurs when an operation generates a value that is too small to represent in the given floating-point format. Under such cases, a zero value is returned. If a TMU operation generates an underflow condition, then the latched underflow flag (LUF) is set to 1. The LUF flag will remain latched until cleared by the user executing an instruction that clears the flag.
Overflow: Overflow occurs when an operation generates a value that is too large to represent in the given floating-point format. Under such cases, a \(\pm\) Infinity value is returned. If a TMU operation generates an overflow condition, then the latched overflow flag (LVF) is set to 1 . The LVF flag will remain latched until cleared by the user executing an instruction that clears the flag.
Rounding: There are various rounding formats supported by the IEEE standard. Rounding has no meaning for TMU operations (rounding is inherent in the implementation). Hence rounding mode is ignored by TMU operations.

Infinity and Not a Number (NaN): An NaN operand ( \(\mathrm{E}==\mathrm{max}, \mathrm{M}!=0\) ) input is treated as Infinity ( \(\mathrm{E}==\mathrm{max}\), \(\mathrm{M}==0\) ) for all operations. TMU operations will never generate a NaN value but Infinity instead.

\subsection*{3.4 Pipeline}

The TMU enhances the instruction set of the C28-FPU and, therefore, operates the C28x pipeline in the same fashion as the FPU. For a detailed explanation on the working of the pipeline, see the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUEO2).

\subsection*{3.4.1 Pipeline and Register Conflicts}

In addition to the restrictions mentioned in the TMS320C28x Floating Point Unit and Instruction Set Reference Guide (SPRUEO2), the TMU places the following restrictions on its instructions:

\section*{Example 3-1. SINPUF32 Operation (4p cycles)}
```

SINPUF32 RaH,RbH ; Value in registers RbH read in this cycle.
Instruction1 ; Instructions 1-3 cannot operate on register RaH.
Instruction2 ; Instructions 1-3 can operate on register RbH.
Instruction3 ; Instructions 1-3 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result in RaH usable by Instruction 4.

```

\section*{Example 3-2. COSPUF32 Operation (4p cycles)}
```

COSPUF32 RaH,RbH ; Value in registers RbH read in this cycle.
Instruction1 ; Instructions 1-3 cannot operate on register RaH.
Instruction2 ; Instructions 1-3 can operate on register RbH.
Instruction3 ; Instructions 1-3 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result in RaH usable by Instruction4.

```

\section*{Example 3-3. ATANPUF32 Operation (4p cycles)}
```

ATANPUF32 RaH,RbH ; Value in registers RbH read in this cycle.
Instruction1 ; Instructions 1-3 cannot operate on register RaH.
Instruction2 ; Instructions 1-3 can operate on register RbH.
Instruction3 ; Instructions 1-3 can be any TMU/FPU/VCU/CPU operation.
; Result, LVF flag updated on Instruction3 (4th cycle).
Instruction4 ; Result in RaH usable by Instruction4.

```

\section*{Example 3-4. DIVF32 Operation (5p cycles)}
```

DIVF32 RaH,RbH,RcH ; Value in registers RbH \& RcH read in this cycle.
Instruction1 ; Instructions 1-4 cannot operate on register RaH.
Instruction2 ; Instructions 1-4 can operate on register RbH \& RcH.
Instruction3 ; Instructions 1-4 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result, LVF and LUF flags updated on Instruction4 (5th cycle).
Instruction5 ; Result in RaH usable by Instruction5.

```

Example 3-5. SQRTF32 Operation (5p cycles)
```

SQRTF32 RaH,RbH ; Value in register RbH read in this cycle.
Instruction1 ; Instructions 1-4 cannot operate on register RaH.
Instruction2 ; Instructions 1-4 can operate on register RbH.
Instruction3 ; Instructions 1-4 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result, LVF flag updated on Instruction4 (5th cycle).
Instruction5 ; Result in register RaH usable by Instruction5.

```

\section*{Example 3-6. QUADF32 Operations (5p cycles)}
```

QUADF32 RaH,RbH,RcH,RdH
; Value in registers RcH \& RdH read in this cycle.
Instruction1 ; Instructions 1-4 cannot operate on registers RaH \& RbH.
Instruction2 ; Instructions 1-4 can operate on register RbH.
Instruction3 ; Instructions 1-4 can be any TMU/FPU/VCU/CPU operation.
Instruction4 ; Result, LVF and LUF flags updated on Instruction4 (5th cycle).
Instruction5 ; Result in registers RaH \& RbH usable by Instruction5.

```

Texas
InsTruments

\subsection*{3.4.2 Delay Slot Requirements}

The Delay slot requirements for the TMU instructions are presented in Table 3-3.
Table 3-3. Delay Slot Requirements for TMU Instructions
\begin{tabular}{|c|c|}
\hline Case & Description \\
\hline 1 & Any Single Cycle FPU operation (including any memory load/store operation) SINPUF32/COSPUF32/ATANPUF32/QUADF32/MPY2PIF32/DIV2PIF32/DIVF32/SQRTF32 \\
\hline 2 & \begin{tabular}{l}
All FPU 2p-cycle operations MPY/ADD/SUB/.... \\
NOP \\
NOP \\
SINPUF32/COSPUF32/ATANPUF32/QUADF32/MPY2PIF32/DIV2PIF32/DIVF32/SQRTF32
\end{tabular} \\
\hline 3 & \begin{tabular}{l}
SINPUF32/COSPUF32/ATANPUF32 \\
NOP \\
NOP \\
NOP \\
All TMU or FPU operations
\end{tabular} \\
\hline 4 & \begin{tabular}{l}
QUADF32/DIVF32/SQRTF32 \\
NOP \\
NOP \\
NOP \\
NOP \\
All TMU or FPU operations
\end{tabular} \\
\hline \multicolumn{2}{|l|}{Special Cases Involving MPY2PIF32/DIV2PIF32} \\
\hline 5 & \[
\begin{aligned}
& \text { MPY2PIF32/DIV2PIF32 } \\
& \text { NOP } \\
& \text { SINPUF32/COSPUF32 }
\end{aligned}
\] \\
\hline 6 & \begin{tabular}{l}
MPY2PIF32/DIV2PIF32 \\
NOP \\
NOP \\
ATANPUF32/QUADF32/DIVF32/SQRTF32
\end{tabular} \\
\hline 7 & \begin{tabular}{l}
MPY2PIF32/DIV2PIF32 \\
NOP \\
NOP \\
All FPU operations
\end{tabular} \\
\hline 8 & \begin{tabular}{l}
MPY2PIF32/DIV2PIF32 \\
NOP \\
MOV32 mem,RxH; Special case: Store result of MPY2PIF32/DIV2PIF32 to memory (but does not include MOV32 operation between CPU and FPU registers).
\end{tabular} \\
\hline
\end{tabular}

\section*{Notes:}

The "NOPs" can be any other FPU, TMU, VCU or CPU operation that does not conflict with the current active TMU operation (does not use same destination register). For example,

\section*{Example 3-7. Use of Non-Conflicting Instructions in Delay Slots}
```

SINPUF32 R0H,R1H
COSPUF32 R2H,R1H
MOV32 R4H,@VarA
MOV32 R5H,@VarB
ADDF32 R3H,R4H,R0H ; SINPUF32 value (R0H) used here
ADDF32 R7H,R5H,R2H ; COSPUF32 value (R2H) used here

```

The delay FPU slot requirements apply to the operation whereby the destination register value is subsequently used by the TMU operation. For example, in the following case, a parallel MPY and MOV operation precedes the TMU operation and the result from MPY operation is used, then two delay slots are required (Case 2 of Table 3-3):

\section*{Example 3-8. Delay Slot Requirement for TMU Instructions That Use Results of Prior FPU Instructions}
```

MPYF32 R3H,R2H,R1H
||MOV32 R4H,@varA
NOP
NOP
SINPUF32 R6H,R3H

```

If however the result of the parallel MOV operation is used, then no delay slots are required since the MOV will complete in a single cycle. (Case 1 of Table 3-3):

\section*{Example 3-9. FPU Instruction Followed by a Non-Dependent TMU Instruction}
```

MPYF32 R3H,R2H,R1H
||MOV32 R4H,@varA
SINPUF32 R6H,R4H

```

\subsection*{3.4.3 Effect of Delay Slot Operations on the Flags}

The LVF and LUF flags can only be set. If multiple operations (from FPU or TMU) try to set the flags, the operations on the flags are ORed together. Operations that set the LVF or LUF flags (either FPU or TMU) are allowed in delay slots. For example, the following sequence of operations is valid:

\section*{Example 3-10. Valid Back-to-Back Instructions That may Set the LVF, LUF Flag}
```

MPY2PIF32 ROH,R0H
MPY2PIF32 R1H,R1H

```

If the SETFLG, SAVE, RESTORE, MOVSTO, or loading and storing of the STF register, operations try to modify the state of the LVF, LUF flags while a TMU or any other FPU operation is trying to set the flags, the LUV, LVF flags are undefined. This can only occur if the SAVE, SETFLG, RESTORE, MOVSTO, or loading and storing of the STF register, operations are placed in the delay slots of the pipeline operations; this should be avoided. This also applies to ZF and NF flags, which are not affected by TMU operations.

\subsection*{3.4.4 Multi-Cycle Operations in Delay Slots}

A multi-cycle operation like RET, BRANCH, CALL is equivalent to a minimum four NOPs. For example, the code shown in Example 3-11 returns the correct value because LRETR takes a minimum of four cycles to execute (equivalent to four NOPs):

\section*{Example 3-11. Multi-Cycle Operation in the Delay Slot of a TMU Instruction}
```

DIVF32 R0H,R2H,R1H
LRETR

```

\subsection*{3.4.5 Moves From FPU Registers to C28x Registers}

When transferring from floating-point unit registers (result of an FPU or TMU operation) to the C28x CPU register, additional pipeline alignment is required as shown in Example 3-12.

\section*{Example 3-12. Floating-Point to C28x Register Software Pipeline Alignment}
```

; SINPUF32: Per unit sine: 4 pipeline cycle operation
; An alignment cycle is required before copying ROH to ACC
SINPUF32 R0H,R1H
NOP ; Delay Slot 1
NOP ; Delay Slot 2
NOP ; Delay Slot 3
NOP ; Alignment cycle
MOV32 @ACC,ROH

```

\subsection*{3.5 TMU Instruction Set}

This section describes the assembly language instructions of the TMU.

\subsection*{3.5.1 Instruction Descriptions}

This section provides 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. TMU instructions follow the same format as the C28x and the \(\mathrm{C} 28 \mathrm{x}+\mathrm{FPU}\); 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 C28x TMU are given in Table 3-4. For information on the operands of standard C28x instructions, see the TMS320C28x CPU and Instruction Set Reference Guide (SPRU430).

Table 3-4. Operand Nomenclature
\begin{tabular}{|l|l|}
\hline Symbol & Description \\
\hline \#16FHi & \begin{tabular}{l} 
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.
\end{tabular} \\
\hline \#16FHiHex & \begin{tabular}{l}
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.
\end{tabular} \\
\hline \#16FLoHex & A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value \\
\hline \#32Fhex & 32-bit immediate value that represents an IEEE 32-bit floating-point value \\
\hline \#32F & Immediate float value represented in floating-point representation \\
\hline \#0.0 & Immediate zero \\
\hline \#RC & 16-bit immediate value for the repeat count \\
\hline *(0:16bitAddr) & Condition to test the flags in the STF register \\
\hline CNDF & Selected flags from STF register (OR) 11 bit mask indicating which floating-point status flags to change \\
\hline FLAG & Label representing the end of the repeat block \\
\hline label & Pointer (using any of the direct or indirect addressing modes) to a 16-bit memory location \\
\hline mem16 & Pointer (using any of the direct or indirect addressing modes) to a 32-bit memory location \\
\hline mem32 & R0H to R7H registers \\
\hline RaH & R0H to R7H registers \\
\hline RbH & R0H to R7H registers \\
\hline RcH & R0H to R7H registers \\
\hline RdH & R0H to R7H registers \\
\hline ReH & R0H to R7H registers \\
\hline RfH & Repeat Block Register \\
\hline RB & FPU Status Register \\
\hline STF & Flag value of 0 or 1 for selected flag (OR) 11 bit mask indicating the flag value; 0 or 1 \\
\hline VALUE & \\
\hline
\end{tabular}

\section*{INSTRUCTION dest1, source1, source2 Short Description}

\section*{Operands}
\begin{tabular}{ll} 
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
\end{tabular}

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

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

This section describes the instruction in terms of pipeline cycles.
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.

\subsection*{3.5.2 Common Restrictions}

For all the TMU instructions, the inputs are conditioned as follows (LVF, LUF are not affected):
- Negative zero is treated as positive zero
- Positive or negative denormalized numbers are treated as positive zero
- Positive and negative NaN are treated as positve and negative infinity respectively

\subsection*{3.5.3 Instructions}

The instructions are listed alphabetically.

\section*{Table 3-5. Summary of Instructions}
Title ..... Page
MPY2PIF32 RaH, RbH - 32-Bit Floating-Point Multiply by Two Pi ..... 391

\section*{Table 3-5. Summary of Instructions (continued)}

DIV2PIF32 RaH, RbH - 32-Bit Floating-Point Divide by Two Pi...................................................................... 392
DIVF32 RaH, RbH, RcH — 32-Bit Floating-Point Division............................................................................ 393
SQRTF32 RaH, RbH — 32-Bit Floating-Point Square Root............................................................................ 395
SINPUF32 RaH, RbH — 32-Bit Floating-Point Sine (per unit) ...................................................................... 396
COSPUF32 RaH, RbH — 32-Bit Floating-Point Cosine (per unit) ...................................................................... 398
ATANPUF32 RaH, RbH — 32-Bit Floating-Point ArcTangent (per unit)........................................................... 400
QUADF32 RaH, RbH, RcH —Quadrant Determination Used in Conjunction With ATANPUF32()............................ 401

\section*{MPY2PIF32 RaH, RbH 32-Bit Floating-Point Multiply by Two Pi}

\section*{Operands}
\(\mathrm{RaH} \quad\) Floating-point destination register (ROH to R7H)
\(\mathrm{RbH} \quad\) Floating-point source register (ROH to R7H)

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001110000 \\
MSW & 00000000 00bb baaa
\end{tabular}

\section*{Description \\ This operation is similar to the MPYF32 operation except that the second operand is the} constant value 2 pi:
\[
\mathrm{RaH}=\mathrm{RbH} * 2 \mathrm{pi}
\]

This operation is used in converting Per Unit values to Radians. Per Unit values are used in control applications to represent normalized radians:
\begin{tabular}{cc} 
Per Unit & Radians \\
1.0 & 2 pi \\
0.0 & 0 \\
1.0 & 2 pi
\end{tabular}
\(2 \mathrm{pi}=6.28318530718=1.570796326795{ }^{*} 2^{\wedge} 2\)
In IEEE 32-bit Floating point format:
\[
\begin{aligned}
& S=0 \ll 31=0 \times 00000000 \\
& E=(2+127) \ll 23=129 \ll 23=0 \times 40800000 \\
& M=\left(1.570796326795^{*} 2^{\wedge} 23\right) \& 0 \times 007 F F F F F=0 \times 00490 F D B \\
& 2 p i=S+E+M=0 \times 40 C 90 F D B
\end{aligned}
\]

Flags
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & Yes \\
\hline
\end{tabular}
\(\left.\left.\begin{array}{ll}\text { Restrictions } & \text { If }(\text { RaH result is too big for floating-point number, Ea }>255) \text { \{ } \\ \text { RaH }= \pm \text { Infinity }\end{array}\right\} \begin{array}{l}\text { LVF }=1 ;\end{array}\right\}\)

Pipeline Instruction takes 2 pipeline cycles to execute if followed by either SINPUF32, COSPUF32 or MOV32 mem, Rx operations and 3 pipeline cycles for all other operations (FPU or TMU).

\section*{Example}
\[
\begin{array}{lll}
\text {; } ; \text { Convert Per Unit value to Radians: } \\
\text { MOV32 } & \text { ROH, @PerUnit } & ; \text { R0H }=\text { Per Unit value } \\
\text { MPY2PIF32 } & \text { R0H, R0H } & ; \text { ROH }=\text { R0H * 2pi } \\
\text { NOP } & & ; \text { pipeline delay } \\
\text { MOV32 } & \text { @Radians,R0H } & ; \text { store Radian result } \\
& & ; 4 \text { cycles }
\end{array}
\]

\section*{DIV2PIF32 RaH, RbH 32-Bit Floating-Point Divide by Two Pi}

\section*{Operands}
\(\mathrm{RaH} \quad\) Floating-point destination register (ROH to R7H)
\(\mathrm{RbH} \quad\) Floating-point source register (ROH to R7H)

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001110001 \\
MSW & 00000000 00bb baaa
\end{tabular}

\section*{Description \\ This operation is similar to the MPYF32 operation except that the second operand is the} constant value \(1 / 2\) pi:
\[
\mathrm{RaH}=\mathrm{RbH}^{*} 1 / 2 \mathrm{pi}
\]

This operation is used in converting Radians to Per unit values. Per unit values are used in control representing normalized Radians:
\begin{tabular}{cc} 
Per Unit & Radians \\
1.0 & 2 pi \\
0.0 & 0 \\
-1.0 & -2 pi
\end{tabular}

In IEEE 32-bit Floating point format:
\[
\begin{aligned}
& 1 / 2 \mathrm{pi}=0.1591549430919=1.273239544735^{*} 2^{\wedge}-3 \\
& S=0 \ll 31=0 \times 00000000 \\
& E=(-3+127) \ll 23=124 \ll 23=0 \times 3 E 000000 \\
& M=\left(1.273239544735^{*} 2^{\wedge} 23\right) \& 0 \times 007 F F F F F=0 \times 0022 F 983 \\
& 1 / 2 \mathrm{pi}=\text { S+E+M = 0x3E22F983 }
\end{aligned}
\]

\section*{Flags}
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & No \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Restrictions & If( RaH result is too small for floating-point number, Ea < 0) \{ \\
\hline & \(\mathrm{RaH}=0.0\) \\
\hline & \(L U F=1 ;\) \\
\hline & \} \\
\hline
\end{tabular}

Pipeline Instruction takes 2 pipeline cycles to execute if followed by either SINPUF32, COSPUF32 or MOV32 mem, Rx operations and 3 pipeline cycles for all other operations (FPU or TMU).

Example
```

;; Convert Per Unit value to Radians:
MOV32 ROH,@Radians ; ROH = Radian value
DIV2PIF32 ROH,ROH ; ROH = ROH * 1/2pi
NOP ; pipeline delay
MOV32 @Per Unit ; store Per Unit result
; 4 cycles

```

InSTRUMENTS

\section*{DIVF32 RaH, RbH, RcH 32-Bit Floating-Point Division}

\section*{Operands}
\begin{tabular}{ll}
RaH & Floating-point destination register (ROH to R7H) \\
RbH & Floating-point source register (ROH to R7H) \\
RcH & Floating-point source register (ROH to R7H)
\end{tabular}

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001110100 \\
MSW & 0000 000c ccbb baaa
\end{tabular}

\section*{Description}
\[
\mathrm{RaH}=\mathrm{RbH} / \mathrm{RcH}
\]

The sequence of operations are as follows:
```

Sa = Sb ^ Sc;
// Set sign of result
Ea = (Eb - Ec) + 127; // Calculate Exponent
Ma = Mb / Mc; // 0.5 < Ma < 2.0
if(Ma < 1.0){){ // Re-normalize mantissa range
Ea = Ea - 1;
Ma = Ma * 2.0;
}
if(Ea >= 255){ // Chek if result too big:
Ea = 255; // Return Inf
Ma = 0;
LVF = 1; // Set overflow flag
}
if((Ea == 0) \& (Ma != 0)){ // Check if result Denorm value:
Sa = 0;
Ea = 0; // Return zero
Ma = 0;
LUF = 1; // Set underflow flag
}
if(Ea < 0){ // Check if result too small:
Sa = 0;
Ea = 0; // Return zero
Ma = 0
LUF = 1; // Set underflow flag
}

```

Flags
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

\section*{Restrictions}

The following boundary conditions apply:
\begin{tabular}{|c|c|c|c|}
\hline Division & Result & LVF & LUF \\
\hline \(0 / 0\) & 0 & 1 & - \\
\hline \(0 / \operatorname{lnf}\) & 0 & - & 1 \\
\hline \(\operatorname{lnf} /\) Normal & \(\operatorname{lnf}\) & 1 & - \\
\hline \(\operatorname{lnf} / 0\) & \(\operatorname{lnf}\) & 1 & - \\
\hline \(\operatorname{lnf} / \operatorname{lnf}\) & \(\operatorname{lnf}\) & - & 1 \\
\hline Normal/0 & \(\operatorname{lnf}\) & 1 & - \\
\hline Normal/lnf & 0 & - & 1 \\
\hline
\end{tabular}

Pipeline Instruction takes 5 pipeline cycles to execute.

Example
```

    ;; Calculate Z = Y/X
        MOV32 R0H,@X ; ROH = X
    MOV32 R1H,@Y ; R1H = Y
    DIVF32 R2H,R1H,R0H ; R2H = R1H/R0H = Y/X = Z
    NOP ; pipeline delay
    NOP ; pipeline delay
    NOP ; pipeline delay
    NOP ; pipeline delay
    MOV32 @Z,R2H ; Z = Y/X
        ; 8 cycles
    ```

\section*{SQRTF32 RaH, RbH 32-Bit Floating-Point Square Root}

\section*{Operands}
RaH Floating-point destination register (ROH to R7H)
\(\mathrm{RbH} \quad\) Floating-point source register ( ROH to R 7 H )

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001110111 \\
MSW & 00000000 00bb baaa
\end{tabular}

\section*{Description \(\quad\) Rah \(=\sqrt{R b H}\)}

\section*{Flags}
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & Yes \\
\hline
\end{tabular}


Pipeline Instruction takes 5 pipeline cycles to execute.

\section*{Example}
```

;; Calculate Y = sqrt(X)
MOV32 R0H,@X ; ROH = X
SQRTF32 R1H,R0H ; R1H = sqrt(X)
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MOV32 @Y,R1H ; Y = sqrt(X)
; 7 cycles

```

\section*{SINPUF32 RaH, RbH 32-Bit Floating-Point Sine (per unit)}

\section*{Operands}
\(\mathrm{RaH} \quad\) Floating-point destination register (ROH to R7H)
\(\mathrm{RbH} \quad\) Floating-point source register (ROH to R7H)

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001111000 \\
MSW & 00000000 00bb baaa
\end{tabular}

\section*{Description \\ This instruction performs the following equivalent operation:}

PerUnit \(=\) fraction \((\mathrm{RbH})\)
RaH \(=\sin\) (PerUnit*2pi)
In control applications radians are usually normalized to the range of -1.0 to 1.0 .

Per Unit
1.0
0.0
-1.0

Radians
2pi
0
-2pi

The operation takes the fraction of the input value RbH . This equates to the cosine waveform repeating itself every 2 pi radians
\begin{tabular}{|c|c|c|c|}
\hline RbH & Per Unit & Radians & Sine Value \\
\hline 2.0 & 0.0 & 0 & 0.0 \\
\hline 1.75 & 0.75 & \(3 \mathrm{pi} / 2\) & -1.0 \\
\hline 1.5 & 0.5 & pi & 0.0 \\
\hline 1.25 & 0.25 & \(\mathrm{pi} / 2\) & 1.0 \\
\hline 1.0 & 0.0 & 0 & 0.0 \\
\hline 0.75 & 0.75 & \(3 \mathrm{pi} / 2\) & -1.0 \\
\hline 0.5 & 0.5 & pi & 0.0 \\
\hline 0.25 & 0.25 & \(\mathrm{pi} / 2\) & 1.0 \\
\hline 0.0 & 0.0 & 0 & 0.0 \\
\hline-0.25 & -0.25 & \(-\mathrm{pi} / 2\) & -1.0 \\
\hline-0.5 & -0.5 & -pi & 0.0 \\
\hline-0.75 & -0.75 & \(-3 \mathrm{pi} / 2\) & 1.0 \\
\hline-1.0 & 0.0 & 0 & 0.0 \\
\hline-1.25 & -0.25 & \(-\mathrm{pi} / 2\) & -1.0 \\
\hline-1.5 & -0.5 & -pi & 0.0 \\
\hline-1.75 & -0.75 & \(-3 \mathrm{pi} / 2\) & 1.0 \\
\hline-2.0 & 0.0 & 0 & 0.0 \\
\hline
\end{tabular}

\section*{Flags}
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Restrictions & If the input value is too small \(\left(<=2^{\wedge}-33\right)\) or too big ( \(>=2^{\wedge} 22\) ), then returned as 0.0 (no flags affected). \\
\hline Pipeline & Instruction takes 4 pipeline cycles to execute. \\
\hline \multirow[t]{8}{*}{Example} & ; \(;\) Convert Radian value to PerUnit value and ; ; calculate Sin value: \\
\hline & \(\begin{array}{ll}\text { MOV32 } & \text { R0H, @RadianValue } \\ \text { DIV2PIF32 } & \text { R1H,R0H }\end{array}\) \\
\hline & \begin{tabular}{ll} 
& \(; R 1 H=R 0 H / 2 p i=\) \\
NOP & \(;\) per Unit Value \\
& pipeline delay
\end{tabular} \\
\hline & SINPUF32 R2H,R1H ; R2H = SINPU(fraction(R1H)) \\
\hline & NOP ; pipeline delay \\
\hline & NOP ; pipeline delay \\
\hline & NOP ; pipeline delay \\
\hline & \begin{tabular}{ll} 
MOV32 @SinValue, R2H & \(;\) Sin Value=sin(Radian Value) \\
& \(; 8\) cycles
\end{tabular} \\
\hline
\end{tabular}

\section*{COSPUF32 RaH, RbH 32-Bit Floating-Point Cosine (per unit)}

\section*{Operands}
\(\mathrm{RaH} \quad\) Floating-point destination register (ROH to R7H)
\(\mathrm{RbH} \quad\) Floating-point source register (ROH to R7H)

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001111001 \\
MSW & 00000000 00bb baaa
\end{tabular}

\section*{Description \\ This instruction performs the following equivalent operation:}

PerUnit \(=\) fraction \((\mathrm{RbH})\)
RaH \(=\cos (\) PerUnit*2pi)
In control applications radians are usually normalized to the range of -1.0 to 1.0 .

Per Unit
1.0
0.0
-1.0

Radians
2pi
0
\(-2 p i\)

The operation takes the fraction of the input value RbH . This equates to the cosine waveform repeating itself every 2pi radians
\begin{tabular}{|c|c|c|c|}
\hline RbH & Per Unit & Radians & Cosine Value \\
\hline 2.0 & 0.0 & 0 & 1.0 \\
\hline 1.75 & 0.75 & \(3 \mathrm{pi} / 2\) & 0.0 \\
\hline 1.5 & 0.5 & pi & -1.0 \\
\hline 1.25 & 0.25 & \(\mathrm{pi} / 2\) & 0.0 \\
\hline 1.0 & 0.0 & 0 & 1.0 \\
\hline 0.75 & 0.75 & \(3 \mathrm{pi} / 2\) & 0.0 \\
\hline 0.5 & 0.5 & pi & -1.0 \\
\hline 0.25 & 0.25 & \(\mathrm{pi} / 2\) & 0.0 \\
\hline 0.0 & 0.0 & 0 & 1.0 \\
\hline-0.25 & -0.25 & \(-\mathrm{pi} / 2\) & 0.0 \\
\hline-0.5 & -0.5 & -pi & -1.0 \\
\hline-0.75 & -0.75 & \(-3 \mathrm{pi} / 2\) & 0.0 \\
\hline-1.0 & 0.0 & 0 & 1.0 \\
\hline-1.25 & -0.25 & \(-\mathrm{pi} / 2\) & 0.0 \\
\hline-1.5 & -0.5 & -pi & -1.0 \\
\hline-1.75 & -0.75 & \(-3 \mathrm{pi} / 2\) & 0.0 \\
\hline-2.0 & 0.0 & 0 & 1.0 \\
\hline
\end{tabular}

\section*{Flags}
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & No \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline Restrictions & If the input value is too small \(\left(<=2^{\wedge}-33\right)\) or too big ( \(>=2^{\wedge} 22\) ), then returned as 1.0 (no flags affected). \\
\hline Pipeline & Instruction takes 4 pipeline cycles to execute. \\
\hline \multirow[t]{8}{*}{Example} & \begin{tabular}{l}
; Convert Radian value to PerUnit value and \\
; ; calculate Sin value:
\end{tabular} \\
\hline & \begin{tabular}{ll} 
MOV32 & R0H, @RadianValue ; R0H = Radian value \\
DIV2PIF32 & R1H,R0H
\end{tabular} \\
\hline & NOP \(\quad ; \quad\) R1H=R0H/2pi= Per Unit Value \\
\hline & COSPUF32 R2H,R1H ; R2H = COSPU(fraction(R1H)) \\
\hline & NOP ; pipeline delay \\
\hline & NOP ; pipeline delay \\
\hline & NOP ; pipeline delay \\
\hline & \(\begin{array}{ll}\text { MOV32 @CosValue, R2H } & ; \operatorname{Cos} \text { Value=cos(Radian Value) } \\ & ; 8 \text { cycles }\end{array}\) \\
\hline
\end{tabular}

\section*{ATANPUF32 RaH, RbH 32-Bit Floating-Point ArcTangent (per unit)}

\section*{Operands}
\(\mathrm{RaH} \quad\) Floating-point destination register (ROH to R7H)
\(\mathrm{RbH} \quad\) Floating-point source register (ROH to R7H)

\section*{Opcode}
\begin{tabular}{ll} 
LSW & 1110001001111010 \\
MSW & 00000000 00bb baaa
\end{tabular}

\section*{Description \\ This instruction computes the arc tangent of a given value and returns the result as a} per-unit value:
\[
\text { PerUnit }=\operatorname{atan}(\mathrm{RbH}) / 2 \mathrm{pi}
\]

The operation limits the input ranget of the input value RbH to:
\[
-1.0<=\mathrm{RbH}<=1.0
\]

Values outside this range return 0.125 as follows:
\begin{tabular}{|ccccc|}
\hline RbH & Per Unit & Radians & ATANPU Value & LVF Flag \\
\hline\(>1.0\) & 0.125 & \(\mathrm{pi} / 4\) & 0.125 & 1 \\
\hline 1.0 & 0.125 & \(\mathrm{pi} / 4\) & 0.125 & \\
\hline 0.0 & 0.0 & 0 & 0.0 & \\
\hline-1.0 & -0.125 & \(-\mathrm{pi} / 4\) & -0.125 & \\
\hline\(<-1.0\) & -0.125 & \(-\mathrm{pi} / 4\) & -0.125 & 1 \\
\hline
\end{tabular}

\section*{Flags}
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & No & Yes \\
\hline
\end{tabular}

Pipeline Instruction takes 4 pipeline cycles to execute.

\section*{Example}
```

;; Calculate ATAN and generate Per Unit value and
;; convert to Radians:
MOV32 ROH,@AtanValue ; ROH = Atan Value
ATANPUF32 R1H,R0H ; R1H = ATANPU(R0H)
NOP ; pipeline delay
NOP ; pipeline delay
NOP ; pipeline delay
MPY2PIF32 R2H,R1H ; R2H = R1H * 2pi
; ; = Radian value
NOP ; pipeline delay
MOV @RadianValue,R2H ; Store result
; 8 cycles

```

\section*{QUADF32 RaH, RbH, RcH Quadrant Determination Used in Conjunction With ATANPUF32()}

\section*{Operands}
\begin{tabular}{ll} 
RaH & Floating-point destination register (ROH to R7H) \\
RbH & Floating-point destination register (ROH to R7H) \\
RcH & Floating-point source register (ROH to R7H) \\
RdH & Floating-point source register (ROH to R7H)
\end{tabular}

Opcode

LSW
MSW

\section*{Description}

0000 dddc ccbb baaa
This operation, in conjunction with atanpu(), is used in calculating atanpu2() for a full circle:
\[
\begin{aligned}
& \mathrm{RdH}=\mathrm{X} \text { value } \\
& \mathrm{RcH}=\mathrm{Y} \text { value } \\
& \mathrm{RbH}=\text { Ratio of } X \mathrm{Y} \\
& \mathrm{RaH}=\text { Quadrant value }(0.0, \pm 0.25, \pm 0.5)
\end{aligned}
\]

Calculation of RaH (Quadrant) and RbH (Ratio) Based on RcH (Y) and RdH (X) Values shows how the values RaH and RbH are generated based on the contents of RbH and RcH.

The algorithm for this instruction is as follows:
```

if( (fabs(RcH(Y)) == 0.0) \& (fabs(RdH(X)) == 0.0) ) {
RaH(Quadrant) = 0.0;
RbH(Ratio) = 0.0;
}else if( fabs(RcH(Y)) < = fabs(RdH(X)) ) {
RbH(Ratio) = RcH(Y) / RdH(X);
if( RdH(X) >= 0.0 )
RaH(Quadrant) = 0.0;
else {
if( RcH(Y) >= 0.0 )
RaH(Quadrant) = 0.5;
else
RaH (Quadrant) = -0.5;
}
}else {
if( RcH(Y) >= 0.0 )
RaH(Quadrant) = 0.25;
else
RaH(Quadrant) = -0.25;
RbH(Ratio) = - RdH(X) / RcH(Y);
}

```

\section*{Flags}
\begin{tabular}{|cccccccc|}
\hline Flag & TF & ZI & NI & ZF & NF & LUF & LVF \\
\hline Modified & No & No & No & No & No & Yes & Yes \\
\hline
\end{tabular}

\section*{Restrictions}
\begin{tabular}{|c|c|c|c|}
\hline Division & Result & LVF & LUF \\
\hline \(0 / 0\) & 0 & 1 & - \\
\hline \(0 / \operatorname{lnf}\) & 0 & - & 1 \\
\hline \(\operatorname{lnf} /\) Normal & \(\operatorname{lnf}\) & 1 & - \\
\hline \(\operatorname{lnf} / 0\) & \(\operatorname{lnf}\) & 1 & - \\
\hline \(\operatorname{lnf} / \operatorname{lnf}\) & \(\operatorname{lnf}\) & - & 1 \\
\hline Normal/0 & \(\operatorname{lnf}\) & 1 & - \\
\hline Normal/lnf & 0 & - & 1 \\
\hline
\end{tabular}

Pipeline Instruction takes 5 pipeline cycles to execute.
Example


Calculation of RaH (Quadrant) and RbH (Ratio) Based on RcH (Y) and RdH (X) Values


If( ( \(\mathrm{Y}|>|\mathrm{X}|) \&(\mathrm{Y}<0))\)
\(\{\) Quadrant \(=-0.25 ;\) Ratio \(=-X / Y\) \}

\section*{Revision History}

\section*{Changes from March 15, 2014 to November 7, 2015 \\ Page}

- Section 1.4.4: Revised this paragraph. .................................................................................................. 21
- Chapter 2: C28 Viterbi, Complex Math and CRC Unit-II (VCU-II) : Made changes to the majority of the LSW and MSW
\(\qquad\)

\section*{IMPORTANT NOTICE}

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in Tl's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.
TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.
TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.
In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, Tl's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.
No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.
Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have not been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio

Amplifiers
Data Converters
DLP® Products
DSP
Clocks and Timers
Interface
Logic
Power Mgmt
Microcontrollers
RFID
OMAP Applications Processors
Wireless Connectivity

Applications
Automotive and
Transportation
Communications and Telecom

Computers and Peripherals
Consumer Electronics
Energy and Lighting
Industrial
Medical
Security
Space, Avionics and Defense
Video and Imaging

TI E2E Community
www.ti.com/omap
www.ti.com/automotive
www.ti.com/communications
www.ti.com/computers
www.ti.com/consumer-apps
www.ti.com/energy
www.ti.com/industrial
www.ti.com/medical
www.ti.com/security
www.ti.com/space-avionics-defense
www.ti.com/video
e2e.ti.com


Компания «ЭлектроПласт» предлагает заключение долгосрочных отношений при поставках импортных электронных компонентов на взаимовыгодных условиях!

Наши преимущества:
- Оперативные поставки широкого спектра электронных компонентов отечественного и импортного производства напрямую от производителей и с крупнейших мировых складов;
- Поставка более 17-ти миллионов наименований электронных компонентов;
- Поставка сложных, дефицитных, либо снятых с производства позиций;
- Оперативные сроки поставки под заказ (от 5 рабочих дней);
- Экспресс доставка в любую точку России;
- Техническая поддержка проекта, помощь в подборе аналогов, поставка прототипов;
- Система менеджмента качества сертифицирована по Международному стандарту ISO 9001;
- Лицензия ФСБ на осуществление работ с использованием сведений, составляющих государственную тайну;
- Поставка специализированных компонентов (Xilinx, Altera, Analog Devices, Intersil, Interpoint, Microsemi, Aeroflex, Peregrine, Syfer, Eurofarad, Texas Instrument, Miteq, Cobham, E2V, MA-COM, Hittite, Mini-Circuits,General Dynamics и др.);

Помимо этого, одним из направлений компании «ЭлектроПласт» является направление «Источники питания». Мы предлагаем Вам помощь Конструкторского отдела:
- Подбор оптимального решения, техническое обоснование при выборе компонента;
- Подбор аналогов;
- Консультации по применению компонента;
- Поставка образцов и прототипов;
- Техническая поддержка проекта;
- Защита от снятия компонента с производства.


Как с нами связаться
Телефон: 8 (812) 3095832 (многоканальный) Факс: 8 (812) 320-02-42
Электронная почта: org@eplast1.ru
Адрес: 198099, г. Санкт-Петербург, ул. Калинина, дом 2 , корпус 4 , литера A.```

