

TMS320x2803x Piccolo Control Law Accelerator (CLA)

Reference Guide



Literature Number: SPRUGE6B
May 2009–Revised May 2010

| | |
|---|------------|
| Preface | 6 |
| 1 Control Law Accelerator (CLA) Overview | 8 |
| 2 CLA Interface | 10 |
| 2.1 CLA Memory | 10 |
| 2.2 CLA Memory Bus | 10 |
| 2.3 Shared Peripherals and EALLOW Protection | 11 |
| 2.4 CLA Tasks and Interrupt Vectors | 12 |
| 3 CLA Configuration and Debug | 13 |
| 3.1 Building a CLA Application | 13 |
| 3.2 Typical CLA Initialization Sequence | 13 |
| 3.3 Debugging CLA Code | 15 |
| 3.4 CLA Illegal Opcode Behavior | 16 |
| 3.5 Resetting the CLA | 16 |
| 4 Register Set | 17 |
| 4.1 Register Memory Mapping | 17 |
| 4.2 Task Interrupt Vector Registers | 18 |
| 4.3 Configuration Registers | 18 |
| 4.4 Execution Registers | 29 |
| 5 Pipeline | 32 |
| 5.1 Pipeline Overview | 32 |
| 5.2 CLA Pipeline Alignment | 32 |
| 5.3 Parallel Instructions | 35 |
| 6 Instruction Set | 37 |
| 6.1 Instruction Descriptions | 37 |
| 6.2 Addressing Modes and Encoding | 39 |
| 6.3 Instructions | 41 |
| Appendix A CLA and CPU Arbitration | 151 |
| A.1 CLA and CPU Arbitration | 151 |
| Appendix B Revision History | 154 |

List of Figures

| | | |
|----|--|----|
| 1 | CLA Block Diagram | 9 |
| 2 | Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8) Register | 18 |
| 3 | Control Register (MCTL) | 18 |
| 4 | Memory Configuration Register (MMEMCFG)..... | 20 |
| 5 | CLA Peripheral Interrupt Source Select 1 Register (MPISRCSEL1) | 21 |
| 6 | Interrupt Enable Register (MIER) | 22 |
| 7 | Interrupt Flag Register (MIFR)..... | 23 |
| 8 | Interrupt Overflow Flag Register (MIOVF) | 24 |
| 9 | Interrupt Run Status Register (MIRUN) | 25 |
| 10 | Interrupt Force Register (MIFRC) | 26 |
| 11 | Interrupt Flag Clear Register (MICLR)..... | 27 |
| 12 | Interrupt Overflow Flag Clear Register (MICLROVF)..... | 28 |
| 13 | Program Counter (MPC)..... | 29 |
| 14 | CLA Status Register (MSTF) | 29 |

List of Tables

| | | |
|----|---|-----|
| 1 | CLA Module Control and Status Register Set | 17 |
| 2 | Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8) Field Descriptions | 18 |
| 3 | Control Register (MCTL) Field Descriptions | 19 |
| 4 | Memory Configuration Register (MMEMCFG) Field Descriptions | 20 |
| 5 | Peripheral Interrupt Source Select 1 (MPISRCSEL1) Register Field Descriptions | 21 |
| 6 | Interrupt Enable Register (MIER) Field Descriptions | 22 |
| 7 | Interrupt Flag Register (MIFR) Field Descriptions | 23 |
| 8 | Interrupt Overflow Flag Register (MIOVF) Field Descriptions | 24 |
| 9 | Interrupt Run Status Register (MIRUN) Field Descriptions | 25 |
| 10 | Interrupt Force Register (MIFRC) Field Descriptions | 26 |
| 11 | Interrupt Flag Clear Register (MICLR) Field Descriptions | 27 |
| 12 | Interrupt Overflow Flag Clear Register (MICLROVF) Field Descriptions | 28 |
| 13 | Program Counter (MPC) Field Descriptions | 29 |
| 14 | CLA Status (MSTF) Register Field Descriptions | 30 |
| 15 | Write Followed by Read - Read Occurs First | 33 |
| 16 | Write Followed by Read - Write Occurs First | 33 |
| 17 | ADC to CLA Early Interrupt Response | 35 |
| 18 | Operand Nomenclature | 37 |
| 19 | INSTRUCTION dest, source1, source2 Short Description..... | 38 |
| 20 | Addressing Modes..... | 39 |
| 21 | Shift Field Encoding..... | 39 |
| 22 | Condition Field Encoding | 40 |
| 23 | Instructions | 41 |
| 24 | Pipeline Activity For MBCNDD, Branch Not Taken | 56 |
| 25 | Pipeline Activity For MBCNDD, Branch Taken | 56 |
| 26 | Pipeline Activity For MCCNDD, Call Not Taken | 62 |
| 27 | Pipeline Activity For MCCNDD, Call Taken | 62 |
| 28 | Pipeline Activity For MMOV16 MARx, MRa , #16I | 94 |
| 29 | Pipeline Activity For MMOV16 MAR0/MAR1, mem16..... | 97 |
| 30 | Pipeline Activity For MMOVI16 MAR0/MAR1, #16I | 110 |
| 31 | Pipeline Activity For MRCNDD, Return Not Taken | 132 |

| | | |
|----|--|-----|
| 32 | Pipeline Activity For MRCNDD, Return Taken | 132 |
| 33 | Pipeline Activity For MSTOP | 135 |
| 34 | Revisions to this Document..... | 154 |

Read This First

The C28x Control Law Accelerator (CLA) is an independent, fully-programmable, 32-bit floating-point math processor that brings concurrent control-loop execution to the C28x family. The low interrupt-latency of the CLA allows it to read ADC samples "just-in-time." This significantly reduces the ADC sample to output delay to enable faster system response and higher MHz control loops. By using the CLA to service time-critical control loops, the main CPU is free to perform other system tasks such as communications and diagnostics. This document provides an overview of the architectural structure and instruction set of the C28x Control Law Accelerator.

The Control Law Accelerator module described in this reference guide is a Type 0 CLA. See the *TMS320x28xx, 28xxx DSP Peripheral Reference Guide* ([SPRU566](#)) for a list of all devices with a CLA module of the same type, to determine the differences between the types, and for a list of device-specific differences within a type. This document describes the architecture, pipeline, instruction set, and interrupts of the C28x Control Law Accelerator.

About This Manual

The TMS320C2000™ is part of the TMS320™ family.

Notational Conventions

This document uses the following conventions.

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

Related Documentation

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

[SPRS584](#) — **TMS320F28032, TMS320F28033, TMS320F28034, TMS320F28035 Piccolo Microcontrollers Data Manual** contains the pinout, signal descriptions, as well as electrical and timing specifications for the 2803x devices.

[SPRZ295](#) — **TMS320F28032, TMS320F28033, TMS320F28034, TMS320F28035 Piccolo MCU Silicon Errata** describes known advisories on silicon and provides workarounds.

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.

Peripheral Guides—

[SPRUGL8](#) — **TMS320x2803x Piccolo System Control and Interrupts Reference Guide** describes the various interrupts and system control features of the 2803x microcontrollers (MCUs).

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

[SPRUG00](#) — TMS320x2803x Piccolo Boot ROM Reference Guide describes the purpose and features of the boot loader (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.

[SPRUGE6](#) — TMS320x2803x Piccolo Control Law Accelerator (CLA) Reference Guide describes the operation of the Control Law Accelerator (CLA).

[SPRUGE2](#) — TMS320x2803x Piccolo Local Interconnect Network (LIN) Module Reference Guide describes the operation of the Local Interconnect Network (LIN) Module.

[SPRUFK8](#) — TMS320x2803x Piccolo Enhanced Quadrature Encoder Pulse (eQEP) Reference Guide describes the operation of the Enhanced Quadrature Encoder Pulse (eQEP) .

[SPRUGL7](#) — TMS320x2803x Piccolo Enhanced Controller Area Network (eCAN) Reference Guide describes the operation of the Enhanced Controller Area Network (eCAN).

[SPRUGE5](#) — TMS320x2802x, 2803x Piccolo Analog-to-Digital Converter (ADC) and Comparator Reference Guide describes how to configure and use the on-chip ADC module, which is a 12-bit pipelined ADC.

[SPRUGE9](#) — TMS320x2802x, 2803x Piccolo 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.

[SPRUGE8](#) — TMS320x2802x, 2803x Piccolo High-Resolution Pulse Width Modulator (HRPWM) describes the operation of the high-resolution extension to the pulse width modulator (HRPWM).

[SPRUGH1](#) — TMS320x2802x, 2803x Piccolo Serial Communications Interface (SCI) Reference Guide describes how to use the SCI.

[SPRUFZ8](#) — TMS320x2802x, 2803x Piccolo Enhanced Capture (eCAP) Module Reference Guide describes the enhanced capture module. It includes the module description and registers.

[SPRUG71](#) — TMS320x2802x, 2803x Piccolo 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.

[SPRUFZ9](#) — TMS320x2802x, 2803x Piccolo Inter-Integrated Circuit (I2C) Reference Guide describes the features and operation of the inter-integrated circuit (I2C) module.

Tools Guides—

[SPRU513](#) — TMS320C28x Assembly Language Tools 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™ C/C++ compiler. This compiler accepts ANSI standard C/C++ source code and produces TMS320 DSP assembly language source code for the TMS320C28x device.

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

TMS320x2803x Piccolo Control Law Accelerator (CLA)

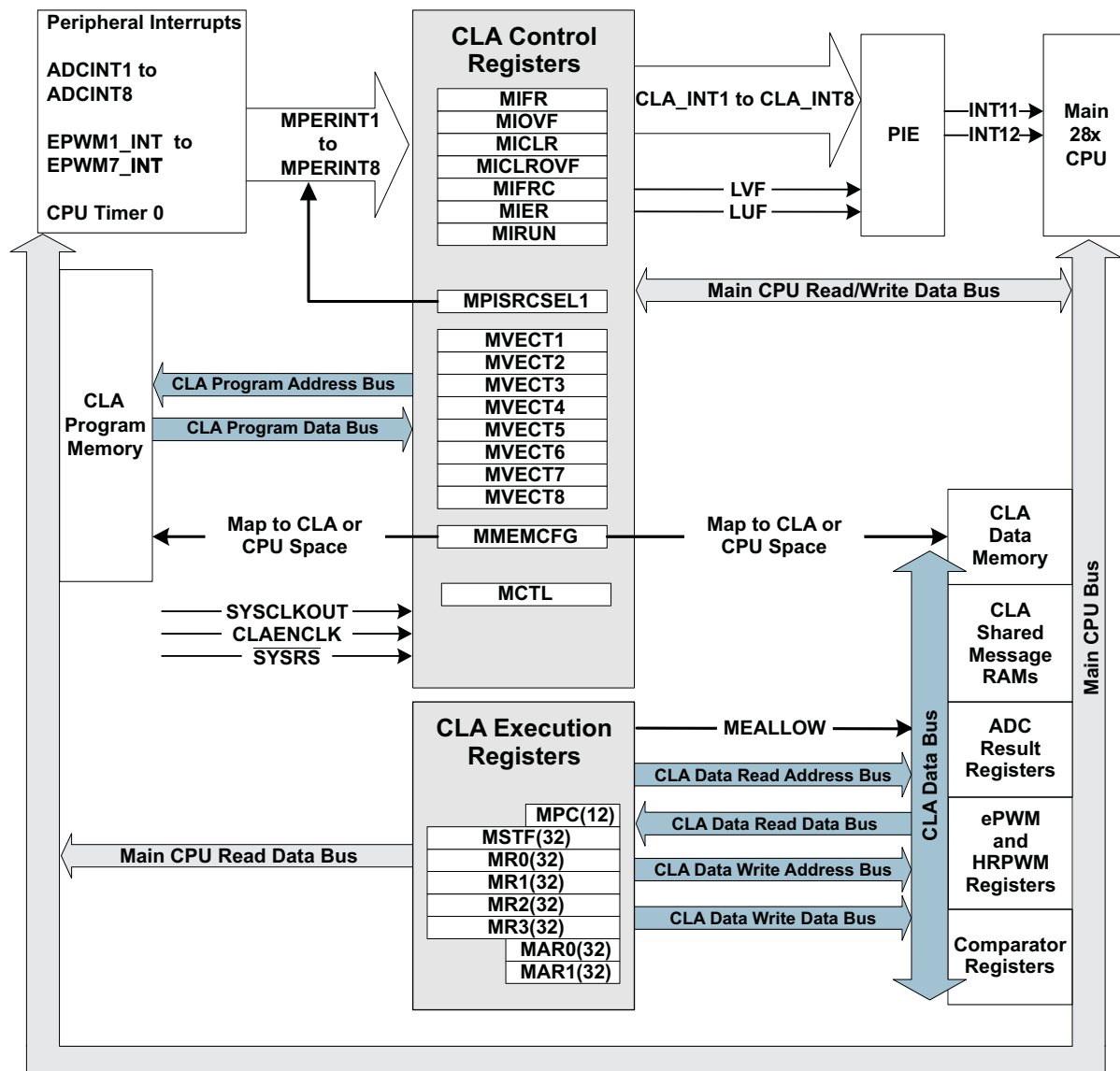
The C28x Control Law Accelerator (CLA) is an independent, fully-programmable, 32-bit floating-point math processor that brings concurrent control-loop execution to the C28x family. The low interrupt-latency of the CLA allows it to read ADC samples "just-in-time". This significantly reduces the ADC sample to output delay to enable faster system response and higher MHz control loops. By using the CLA to service time-critical control loops, the main CPU is free to perform other system tasks such as communications and diagnostics. This chapter provides an overview of the architectural structure and components of the C28x Control Law Accelerator.

1 Control Law Accelerator (CLA) Overview

The control law accelerator extends the capabilities of the C28x CPU by adding parallel processing. Time-critical control loops serviced by the CLA can achieve low ADC sample to output delay. Thus, the CLA enables faster system response and higher frequency control loops. Utilizing the CLA for time-critical tasks frees up the main CPU to perform other system and communication functions concurrently. The following is a list of major features of the CLA.

- Clocked at the same rate as the main CPU (SYSCLKOUT).
- An independent architecture allowing CLA algorithm execution independent of the main C28x CPU.
 - Complete bus architecture:
 - Program address bus and program data bus
 - Data address bus, data read bus and data write bus
 - Independent eight stage pipeline.
 - 12-bit program counter (MPC)
 - Four 32-bit result registers (MR0-MR3)
 - Two 16-bit auxiliary registers (MAR0, MAR1)
 - Status register (MSTF)
- Instruction set includes:
 - IEEE single-precision (32-bit) floating point math operations
 - Floating-point math with parallel load or store
 - Floating-point multiply with parallel add or subtract
 - 1/X and 1/sqrt(X) estimations
 - Data type conversions.
 - Conditional branch and call
 - Data load/store operations
- The CLA program code can consist of up to eight tasks or interrupt service routines.
 - The start address of each task is specified by the MVECT registers.
 - No limit on task size as long as the tasks fit within the CLA program memory space.
 - One task is serviced at a time through to completion. There is no nesting of tasks.
 - Upon task completion a task-specific interrupt is flagged within the PIE.
 - When a task finishes the next highest-priority pending task is automatically started.
- Task trigger mechanisms:
 - C28x CPU via the IACK instruction
 - Task1 to Task7: the corresponding ADC or ePWM module interrupt. For example:
 - Task1: ADCINT1 or EPWM1_INT

- Task2: ADCINT2 or EPWM2_INT
- Task7: ADCINT7 or EPWM7_INT
- Task8: ADCINT8 or by CPU Timer 0.
- Memory and Shared Peripherals:
 - Two dedicated message RAMs for communication between the CLA and the main CPU.
 - The C28x CPU can map CLA program and data memory to the main CPU space or CLA space.
 - The CLA has direct access to the ePWM+HRPWM, Comparator and ADC Result registers.

Figure 1. CLA Block Diagram


2 CLA Interface

This chapter describes how the C28x main CPU can interface to the CLA and vice versa.

2.1 CLA Memory

The CLA can access three types of memory: program, data and message RAMs. The behavior and arbitration for each type of memory is described in detail in [Appendix A](#).

- **CLA Program Memory**

At reset memory designated for CLA program is mapped to the main CPU memory and is treated like any other memory block. While mapped to CPU space, the main CPU can copy the CLA program code into the memory block. During debug the block can also be loaded directly by Code Composer Studio. Once the memory is initialized with CLA code, the main CPU maps it to the CLA program space by writing a 1 to the MMEMCFG[PROGE] bit. When mapped to the CLA program space, the block can only be accessed by the CLA for fetching opcodes. The main CPU can only perform debugger accesses when the CLA is either halted or idle. If the CLA is executing code, then all debugger accesses are blocked and the memory reads back all 0x0000.

CLA program memory is protected by the code security module. All CLA program fetches are performed as 32-bit read operations and all opcodes must be aligned to an even address. Since all CLA opcodes are 32-bits, this alignment naturally occurs.

- **CLA Data Memory**

There are two CLA data memory blocks on the device. At reset, both blocks are mapped to the main CPU memory space and treated by the CPU like any other memory block. While mapped to CPU space, the main CPU can initialize the memory with data tables and coefficients for the CLA to use.

Once the memory is initialized with CLA data the main CPU maps it to the CLA space. Each block can be individually mapped via the MMEMCFG[RAM0E] and MMEMCFG[RAM1E] bits. When mapped to the CLA data space, the memory can be accessed only by the CLA for data operations. The main CPU can only perform debugger accesses in this mode.

Both CLA data RAMs are protected by the code security module and emulation code security logic.

- **CLA Shared Message RAMs**

There are two small memory blocks for data sharing and communication between the CLA and the main CPU. The message RAMs are always mapped to both CPU and CLA memory spaces and are protected by the code security module. The message RAMs allow data accesses only; no program fetches can be performed.

- **CLA to CPU Message RAM**

The CLA can use this block to pass data to the main CPU. This block is both readable and writable by the CLA. This block is also readable by the main CPU but writes by the main CPU are ignored.

- **CPU to CLA Message RAM**

The main CPU can use this block to pass data and messages to the CLA. This message RAM is both readable and writable by the main CPU. The CLA can perform reads but writes by the CLA are ignored.

2.2 CLA Memory Bus

The CLA has dedicated bus architecture similar to that of the C28x CPU where there is a program read, data read and data write bus. Thus there can be simultaneous instruction fetch, data read and data write in a single cycle. Like the C28x CPU, the CLA expects memory logic to align any 32-bit read or write to an even address. If the address-generation logic generates an odd address, the CLA will begin reading or writing at the previous even address. This alignment does not affect the address values generated by the address-generation logic.

- **CLA Program Bus**

The CLA program bus has a access range of 2048 32-bit instructions. Since all CLA instructions are 32-bits, this bus always fetches 32-bits at a time and the opcodes must be even word aligned. The amount of program space available for the CLA is device dependent as described in the device-specific data manual.

- **CLA Data Read Bus**

The CLA data read bus has a 64K x 16 address range. The bus can perform 16 or 32-bit reads and

will automatically stall if there are memory access conflicts. The data read bus has access to both the message RAMs, CLA data memory and the ePWM, HRPWM, Comparator and ADC result registers.

- **CLA Data Write Bus**

The CLA data write bus has a 64K x 16 address range. This bus can perform 16 or 32-bit writes. The bus will automatically stall if there are memory access conflicts. The data write bus has access to the CLA to CPU message RAM, CLA data memory and the ePWM, HRPWM, and Comparator registers.

2.3 Shared Peripherals and EALLOW Protection

The ePWM, HRPWM, Comparator, and ADC result registers can be accessed by both the CLA and the main CPU. [Appendix A](#) describes in detail the CLA and CPU arbitration when both access these registers.

Several peripheral control registers are protected from spurious 28x CPU writes by the EALLOW protection mechanism. These same registers are also protected from spurious CLA writes. The EALLOW bit in the main CPU status register 1 (ST1) indicates the state of protection for the main CPU. Likewise the MEALLOW bit in the CLA status register (MSTF) indicates the state of write protection for the CLA. The MEALLOW CLA instruction enables write access by the CLA to EALLOW protected registers. Likewise the MEDIS CLA instruction will disable write access. This way the CLA can enable/disable write access independent of the main CPU.

The 2803x ADC offers the option to generate an early interrupt pulse when the ADC begins conversion. If this option is used to start a ADC triggered CLA task then the 8th instruction can read the result as soon as the conversion completes. The CLA pipeline activity for this scenario is shown in [Section 5](#).

2.4 CLA Tasks and Interrupt Vectors

The CLA program code is divided up into tasks or interrupt service routines. Tasks do not have a fixed starting location or length. The CLA program memory can be divided up as desired. The CLA knows where a task begins by the content of the associated interrupt vector (MVECT1 to MVECT8) and the end is indicated by the MSTOP instruction.

The CLA supports 8 tasks. Task 1 has the highest priority and task 8 has the lowest priority. A task can be requested by a peripheral interrupt or by software:

- **Peripheral interrupt trigger**

Each task has specific interrupt sources that can trigger it. Configure the MPISRCSEL1 register to select from the potential sources. For example, task 1 (MVECT1) can be triggered by ADCINT1 or EPWM1_INT as specified in MPISRCSEL1[PERINT1SEL]. You can not, however, trigger task 1 directly using EPWM2_INT. If you need to trigger a task using EPWM2_INT then the best solution is to use task 2 (MVECT2). Another possible solution is to take EPWM2_INT with the main CPU and trigger a task with software.

To disable the peripheral from sending an interrupt request to the CLA set the PERINT1SEL option to no interrupt.

- **Software trigger**

Tasks can also be started by the main CPU software writing to the MIFRC register or by the IACK instruction. Using the IACK instruction is more efficient because it does not require you to issue an EALLOW to set MIFR bits. Set the MCTL[IACKE] bit to enable the IACK feature. Each bit in the operand of the IACK instruction corresponds to a task. For example IACK #0x0001 will set bit 0 in the MIFR register to start task 1. Likewise IACK #0x0003 will set bits 0 and 1 in the MIFR register to start task 1 and task 2.

The CLA has its own fetch mechanism and can run and execute a task independent of the main CPU. Only one task is serviced at a time; there is no nesting of tasks. The task currently running is indicated in the MIRUN register. Interrupts that have been received but not yet serviced are indicated in the flag register (MIFR). If an interrupt request from a peripheral is received and that same task is already flagged, then the overflow flag bit is set. Overflow flags will remain set until they are cleared by the main CPU.

If the CLA is idle (no task is currently running) then the highest priority interrupt request that is both flagged (MIFR) and enabled (MIER) will start. The flow is as follows

1. The associated RUN register bit is set (MIRUN) and the flag bit (MIFR) is cleared.
2. The CLA begins execution at the location indicated by the associated interrupt vector (MVECTx). MVECT is an offset from the first program memory location.
3. The CLA executes instructions until the MSTOP instruction is found. This indicates the end of the task.
4. The MIRUN bit is cleared.
5. The task-specific interrupt to the PIE is issued. This informs the main CPU that the task has completed.
6. The CLA returns to idle.

Once a task completes the next highest-priority pending task is automatically serviced and this sequence repeats.

3 CLA Configuration and Debug

This section discusses the steps necessary to configure and debug the CLA.

3.1 Building a CLA Application

The Control Law Accelerator is programmed in CLA assembly code using the instructions described in [Section 6](#). CLA assembly code can, and should, reside in the same project with C28x code. **The only restriction is the CLA code must be in its own assembly section. This can be easily done using the .sect assembly directive.** This does not prevent CLA and C28x code from being linked into the same memory region in the linker command file.

System and CLA initialization are performed by the main CPU. This would typically be done in C or C++ but can also include C28x assembly code. The main CPU will also copy the CLA code to the program memory and, if needed, initialize the CLA data RAM(s). Once system initialization is complete and the application begins, the CLA will service its interrupts using the CLA assembly code (or tasks). Concurrently the main CPU can perform other tasks.

The C2000 codegen tools V5.2.x and higher support CLA instructions when the following switch is set: --cla_support = cla0.

3.2 Typical CLA Initialization Sequence

A typical CLA initialization sequence is performed by the main CPU as described in this section.

1. Copy CLA code into the CLA program RAM

The source for the CLA code can initially reside in the flash or a data stream from a communications peripheral or anywhere the main CPU can access it. The debugger can also be used to load code directly to the CLA program RAM during development.

2. Initialize CLA data RAM if necessary

Populate the CLA data RAM with any required data coefficients or constants.

3. Configure the CLA registers

Configure the CLA registers, but keep interrupts disabled until later (leave MIER == 0):

- **Enable the CLA clock in the PCLKCR3 register.**
PCLKCR3 register is defined in the device-specific system control and interrupts reference guide.
- **Populate the CLA task interrupt vectors: MVECT1 to MVECT8.**
Each vector needs to be initialized with the start address of the task to be executed when the CLA receives the associated interrupt. This address is an offset from the first address in CLA program memory. For example, 0x0000 corresponds to the first CLA program memory address.
- **Select the task interrupt sources**
For each task select the interrupt source in the PERINT1SEL register. **If a task is going to be generated by software, select no interrupt.**
- **Enable IACK to start a task from software if desired**
To enable the IACK instruction to start a task set the MCTL[IACKE] bit. Using the IACK instruction avoids having to set and clear the EALLOW bit.
- **Map CLA data RAM(s) to CLA space if necessary**
Map either or both of the data RAMs to the CLA space by writing a 1 to the MMEMCFG[RAM0E] and MMEMCFG[RAM1E] bits. After the memory is mapped to CLA space the main CPU cannot access it. **Allow two SYSCLKOUT cycles between changing the map configuration of this memory and accessing it.**
- **Map CLA program RAM to CLA space**
Map the CLA program RAM to CLA space by setting the MMEMCFG[PROGE] bit. After the memory is remapped to CLA space the main CPU will only be able to make debug accesses to the memory block. Allow two SYSCLKOUT cycles between changing the map configuration of these memories and accessing them.

4. Initialize the PIE vector table and registers

When a CLA task completes the associated interrupt in the PIE will be flagged. The CLA overflow and underflow flags also have associated interrupts within the PIE.

5. Enable CLA tasks/interrupts

Set appropriate bits in the interrupt enable register (MIER) to allow the CLA to service interrupts.

6. Initialize other peripherals

Initialize any peripherals (ePWM, ADC etc.) that will generate an interrupt to the CLA and be serviced by a CLA task.

The CLA is now ready to service interrupts and the message RAMs can be used to pass data between the CPU and the CLA. Typically mapping of the CLA program and data RAMs occurs only during the initialization process. If after some time you want to re-map these memories back to CPU space then disable interrupts and make sure all tasks have completed by checking the MIRUN register. Always allow two SYSCLKOUT cycles when changing the map configuration of these memories and accessing them.

3.3 Debugging CLA Code

Debugging the CLA code is a simple process that occurs independently of the main CPU.

1. Insert a breakpoint in CLA code

Insert a CLA breakpoint (MDEBUGSTOP instruction) into the code where you want the CLA to halt, then rebuild and reload the code. Because the CLA does not flush its pipeline when you single-step, the MDEBUGSTOP instruction must be inserted as part of the code. The debugger cannot insert it as needed.

If CLA breakpoints are not enabled, then the MDEBUGSTOP will be ignored and is treated as a MNOP. The MDEBUGSTOP instruction can be placed anywhere in the CLA code as long as it is not within three instructions of a MBCNDD, MCCNDD, or MRCNDD instruction.

2. Enable CLA breakpoints

First, enable the CLA breakpoints in the debugger. In Code Composer Studio V3.3, this is done by connecting the CLA debug window (debug->connect). Breakpoints are disabled when this window is disconnected.

3. Start the task

There are three ways to start the task:

- The peripheral can assert an interrupt
- The main CPU can execute an IACK instruction, or
- You can manually write to the MIFRC register in the debugger window

When the task starts, the CLA will execute instructions until the MDEBUGSTOP is in the D2 phase of the pipeline. At this point, the CLA will halt and the pipeline will be frozen. The MPC register will reflect the address of the MDEBUGSTOP instruction.

4. Single-step the CLA code

Once halted, you can single-step the CLA code one cycle at a time. The behavior of a CLA single-step is different than the main C28x. When issuing a CLA single-step, the pipeline is clocked only one cycle and then again frozen. On the 28x CPU, the pipeline is flushed for each single-step.

You can also run to the next MDEBUGSTOP or to the end of the task. If another task is pending, it will automatically start when you run to the end of the task.

NOTE: When CLA program memory is mapped to the CLA memory space, a CLA fetch has higher priority than CPU debug reads. For this reason, it is possible for the CLA to permanently block CPU debug accesses if the CLA is executing in a loop. This might occur when initially developing CLA code due to a bug that causes an infinite loop. To avoid locking up the main CPU, the program memory will return all 0x0000 for CPU debug reads when the CLA is running. When the CLA is halted or idle then normal CPU debug read and write access to CLA program memory can be performed.

If the CLA gets caught in a infinite loop, you can use a soft or hard reset to exit the condition. A debugger reset will also exit the condition.

There are special cases that can occur when single-stepping a task such that the program counter, MPC, reaches the MSTOP instruction at the end of the task.

• MPC halts at or after the MSTOP with a task already pending

If you are single-stepping or halted in "task A" and "task B" comes in before the MPC reaches the MSTOP, then "task B" will start if you continue to step through the MSTOP instruction. Basically if "task B" is pending before the MPC reaches MSTOP in "task A" then there is no issue in "task B" starting and no special action is required.

• MPC halts at or after the MSTOP with no task pending

In this case you have single-stepped or halted in "task A" and the MPC has reached the MSTOP with no tasks pending. If "task B" comes in at this point, it will be flagged in the MIFR register but it may or may not start if you continue to single-step through the MSTOP instruction of "task A."

It depends on exactly when the new task comes in. To reliably start "task B" perform a soft reset and reconfigure the MIER bits. Once this is done, you can start single-stepping "task B."

This case can be handled slightly differently if there is control over when "task B" comes in (for example using the IACK instruction to start the task). In this case you have single-stepped or halted

in "task A" and the MPC has reached the MSTOP with no tasks pending. Before forcing "task B," run free to force the CLA out of the debug state. Once this is done you can force "task B" and continue debugging.

5. If desired, disable CLA breakpoints

In CCS V3.3 you can disable the CLA breakpoints by disconnecting the CLA debug window. Make sure to first issue a run or reset; otherwise, the CLA will be halted and no other tasks will start.

3.4 CLA Illegal Opcode Behavior

If the CLA fetches an opcode that does not correspond to a legal instruction, it will behave as follows:

- The CLA will halt with the illegal opcode in the D2 phase of the pipeline as if it were a breakpoint. This will occur whether CLA breakpoints are enabled or not.
- The CLA will issue the task-specific interrupt to the PIE.
- The MIRUN bit for the task will remain set.

Further single-stepping ignored once execution halts due to an illegal op-code. To exit this situation, issue either a soft or hard reset of the CLA as described in [Section 3.5](#).

3.5 Resetting the CLA

There may be times when you need to reset the CLA. For example, during code debug the CLA may enter an infinite loop due to a code bug. The CLA has two types of resets: hard and soft. Both of these resets can be performed by the debugger or by the main CPU.

- **Hard Reset**

Writing a 1 to the MCTL[HARDRESET] bit will perform a hard reset of the CLA. The behavior of a hard reset is the same as a system reset (via \overline{XRS} or the debugger). In this case all CLA configuration and execution registers will be set to their default state and CLA execution will halt.

- **Soft Reset**

Writing a 1 to the MCTL[SOFTRESET] bit performs a soft reset of the CLA. If a task is executing it will halt and the associated MIRUN bit will be cleared. All bits within the interrupt enable (MIER) register will also be cleared so that no new tasks start.

4 Register Set

The CLA register set is independent from that of the main CPU. This chapter describes the CLA register set.

4.1 Register Memory Mapping

The table below describes the CLA module control and status register set.

Table 1. CLA Module Control and Status Register Set

| Name | Offset | Size (x16) | EALLOW | CSM Protected | Description |
|---|--------|---------------|--------|------------------|---|
| Task Interrupt Vectors | | | | | |
| MVECT1 | 0x0000 | 1 | Yes | Yes | Task 1 Interrupt Vector |
| MVECT2 | 0x0001 | 1 | Yes | Yes | Task 2 Interrupt Vector |
| MVECT3 | 0x0002 | 1 | Yes | Yes | Task 3 Interrupt Vector |
| MVECT4 | 0x0003 | 1 | Yes | Yes | Task 4 Interrupt Vector |
| MVECT5 | 0x0004 | 1 | Yes | Yes | Task 5 Interrupt Vector |
| MVECT6 | 0x0005 | 1 | Yes | Yes | Task 6 Interrupt Vector |
| MVECT7 | 0x0006 | 1 | Yes | Yes | Task 7 Interrupt Vector |
| MVECT8 | 0x0007 | 1 | Yes | Yes | Task 8 Interrupt Vector |
| Configuration Registers | | | | | |
| MCTL | 0x0010 | 1 | Yes | Yes | Control Register |
| MMEMCFG | 0x0011 | 1 | Yes | Yes | Memory Configuration Register |
| MPISRCSEL1 | 0x0014 | 2 | Yes | Yes | Peripheral Interrupt Source Select 1 Register |
| MIFR | 0x0020 | 1 | Yes | Yes | Interrupt Flag Register |
| MIOVF | 0x0021 | 1 | Yes | Yes | Interrupt Overflow Flag Register |
| MIFRC | 0x0022 | 1 | Yes | Yes | Interrupt Force Register |
| MICLR | 0x0023 | 1 | Yes | Yes | Interrupt Flag Clear Register |
| MICLROVF | 0x0024 | 1 | Yes | Yes | Interrupt Overflow Flag Clear Register |
| MIER | 0x0025 | 1 | Yes | Yes | Interrupt Enable Register |
| MIRUN | 0x0026 | 1 | Yes | Yes | Interrupt Run Status Register |
| Execution Registers ⁽¹⁾ | | | | | |
| MPC | 0x0028 | 1 | - | Yes | CLA Program Counter |
| MAR0 | 0x0029 | 1 | - | Yes | CLA Auxiliary Register 0 |
| MAR1 | 0x002A | 1 | - | Yes | CLA Auxiliary Register 1 |
| MSTF | 0x002E | 2 | - | Yes | CLA Floating-Point Status Register |
| MR0 | 0x0030 | 2 | - | Yes | CLA Floating-Point Result Register 0 |
| MR1 | 0x0034 | 2 | - | Yes | CLA Floating-Point Result Register 1 |
| MR2 | 0x0038 | 2 | - | Yes | CLA Floating-Point Result Register 2 |
| MR3 | 0x003C | 2 | - | Yes | CLA Floating-Point Result Register 3 |

⁽¹⁾ The main C28x CPU only has read access to the CLA execution registers for debug purposes. The main CPU cannot perform CPU or debugger writes to these registers.

4.2 Task Interrupt Vector Registers

Each CLA interrupt has its own interrupt vector (MVECT1 to MVECT8). This interrupt vector points to the first instruction of the associated task. When a task begins, the CLA will start fetching instructions at the location indicated by the appropriate MVECT register .

4.2.1 Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8) Register

The task interrupt vector registers (MVECT1/2/3/4/5/6/7/8) are shown in [Section 4.2.1](#) and described in [Figure 2](#).

Figure 2. Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8) Register

| | | | |
|----------|----|-------|---|
| 15 | 12 | 11 | 0 |
| Reserved | | MVECT | |
| R-0 | | R-0 | |

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

Table 2. Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|-------|----------|-------------|--|
| 15-12 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 11-0 | MVECT | 0000 - 0FFF | Offset of the first instruction in the associated task from the start of CLA program space. The CLA will begin instruction fetches from this location when the specific task begins. For example: If CLA program memory begins at CPU address 0x009000 and the code for task 5 begins at CPU address 0x009120, then MVECT5 should be initialized with 0x0120. There is one MVECT register per task. Interrupt 1 uses MVECT1, interrupt 2 uses MVECT2 and so forth. |

⁽¹⁾ These registers are protected by EALLOW and the code security module.

4.3 Configuration Registers

The configuration registers are described here.

4.3.1 Control Register (MCTL)

The configuration control register (MCTL) is shown in [Figure 3](#) and described in [Table 3](#).

Figure 3. Control Register (MCTL)

| | | | | | |
|----------|---|--|-------|-----------|-----------|
| 15 | | | | 8 | |
| Reserved | | | | | |
| R -0 | | | | | |
| 7 | 3 | | 2 | 1 | 0 |
| Reserved | | | IACKE | SOFTRESET | HARDRESET |
| R/W-0 | | | R/W-0 | R/W-0 | R/W-0 |

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

Table 3. Control Register (MCTL) Field Descriptions

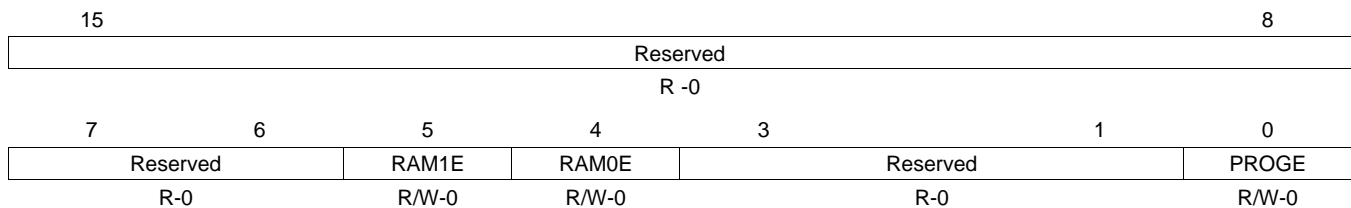
| Bits | Name | Value | Description ⁽¹⁾ |
|------|-----------|--------|--|
| 15-3 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 2 | IACKE | 0 1 | <p>IACK enable</p> <p>0 The CLA ignores the IACK instruction. (default)</p> <p>1 Enable the main CPU to use the IACK #16bit instruction to set MIFR bits in the same manner as writing to the MIFRC register. Each bit in the operand, #16bit, corresponds to a bit in the MIFRC register. Using IACK has the advantage of not having to first set the EALLOW bit. This allows the main CPU to efficiently trigger a CLA task through software.</p> <p>Examples IACK #0x0001 Write a 1 to MIFRC bit 0 to force task 1 IACK #0x0003 Write a 1 to MIFRC bit 0 and 1 to force task 1 and task 2</p> |
| 1 | SOFTRESET | 0 1 | <p>Soft Reset</p> <p>0 This bit always reads back 0 and writes of 0 are ignored.</p> <p>1 Writing a 1 will cause a soft reset of the CLA. This will stop the current task, clear the MIRUN flag and clear all bits in the MIER register. After a soft reset you must wait at least 1 SYSCLKOUT cycle before reconfiguring the MIER bits. If these two operations are done back-to-back then the MIER bits will not get set.</p> |
| 0 | HARDRESET | 0 1 | <p>Hard Reset</p> <p>0 This bit always reads back 0 and writes of 0 are ignored.</p> <p>1 Writing a 1 will cause a hard reset of the CLA. This will set all CLA registers to their default state.</p> |

⁽¹⁾ This register is protected by EALLOW and the code security module.

4.3.2 Memory Configuration Register (MMEMCFG)

The MMEMCFG register is used to map the CLA program and data RAMs to either the CPU or the CLA memory space. Typically mapping of the CLA program and data RAMs occurs only during the initialization process. If after some time the you want to re-map these memories back to CPU space then disable interrupts (MIER) and make sure all tasks have completed by checking the MIRUN register. Allow two SYSCLKOUT cycles between changing the map configuration of these memories and accessing them. Refer to [Section A.1.3](#) for CLA and CPU access arbitration details.

Figure 4. Memory Configuration Register (MMEMCFG)



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

Table 4. Memory Configuration Register (MMEMCFG) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|-------|----------|---------------------------|--|
| 15-6 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 5 | RAM1E | <div>0</div> <div>1</div> | <div>CLA Data RAM 1 Enable</div> <div>Allow two SYSCLKOUT cycles between changing this bit and accessing the memory.</div> <div>The CLA data SARAM block 1 is mapped to the main CPU program and data space. CLA reads will return zero. (default)</div> <div>The CLA data SARAM block 1 is mapped to CLA data space. The main CPU can only make debug accesses to this block.</div> |
| 4 | RAM0E | <div>0</div> <div>1</div> | <div>CLA Data RAM 0 Enable</div> <div>Allow two SYSCLKOUT cycles between changing this bit and accessing the memory.</div> <div>The CLA data SARAM block 0 is mapped to the main CPU program and data space. CLA reads will return zero. (default)</div> <div>The CLA data SARAM block 0 is mapped to CLA data space. The main CPU can only make debug accesses to this block.</div> |
| 3 - 1 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 0 | PROGE | <div>0</div> <div>1</div> | <div>CLA Program Space Enable</div> <div>Allow two SYSCLKOUT cycles between changing this bit and accessing the memory.</div> <div>CLA program SARAM is mapped to the main CPU program and data space. If the CLA attempts a program fetch the result will be the same as an illegal opcode fetch as described in Section 3.4. (default)</div> <div>CLA program SARAM is mapped to the CLA program space. The main CPU can only make debug accesses to this block.</div> <div>In this state a CLA fetch has higher priority than CPU debug reads. It is, therefore, possible for the CLA to permanently block debug accesses if the CLA is executing in a loop. This might occur when initially developing CLA code due to a bug. To avoid this issue, the program memory will return all 0x0000 for CPU debug reads (ignore writes) when the CLA is running. When the CLA is halted or idle then normal CPU debug read and write access can be performed.</div> |

⁽¹⁾ This register is protected by EALLOW and the code security module.

4.3.3 CLA Peripheral Interrupt Source Select 1 Register (MPISRCSEL1)

Each task has specific peripherals that can start it. For example, Task2 can be started by ADCINT2 or EPWM2_INT. To configure which of the possible peripherals will start a task configure the MPISRCSEL1 register shown in [Figure 5](#). Choosing the option "no interrupt source" means that only the main CPU software will be able to start the given task.

Figure 5. CLA Peripheral Interrupt Source Select 1 Register (MPISRCSEL1)

| | | | | | | | |
|------------|----|----|----|------------|----|----|----|
| 31 | 28 | 27 | 24 | 23 | 20 | 19 | 16 |
| PERINT8SEL | | | | PERINT7SEL | | | |
| R/W-0 | | | | R/W-0 | | | |
| 15 | 12 | 11 | 8 | 7 | 4 | 3 | 0 |
| PERINT4SEL | | | | PERINT3SEL | | | |
| R/W-0 | | | | R/W-0 | | | |
| PERINT2SEL | | | | PERINT1SEL | | | |
| R/W-0 | | | | R/W-0 | | | |

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

Table 5. Peripheral Interrupt Source Select 1 (MPISRCSEL1) Register Field Descriptions

| Bits | Field | Value ⁽¹⁾ | Description ⁽²⁾ |
|---------|------------|----------------------|--|
| 31 - 28 | PERINT8SEL | 0000 | Task 8 Peripheral Interrupt Input Select |
| | | 0010 | ADCINT8 is the input for interrupt task 8. (default) |
| | | 0010 | CPU Timer 0 is the input for interrupt task 8. |
| | | xxx1 | No interrupt source for task 8. |
| 27 - 24 | PERINT7SEL | 0000 | Task 7 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT7 is the input for interrupt task 7. (default) |
| | | 0010 | ePWM7 is the input for interrupt task 7. (EPWM7_INT) |
| | | xxx1 | No interrupt source for task 7. |
| 23 - 20 | PERINT6SEL | 0000 | Task 6 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT6 is the input for interrupt task 6. (default) |
| | | 0010 | ePWM6 is the input for interrupt task 6. (EPWM6_INT) |
| | | xxx1 | No interrupt source for task 6. |
| 19 - 16 | PERINT5SEL | 0000 | Task 5 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT5 is the input for interrupt task 5. (default) |
| | | 0010 | ePWM5 is the input for interrupt task 5. (EPWM5_INT) |
| | | xxx1 | No interrupt source for task 5. |
| 15 - 12 | PERINT4SEL | 0000 | Task 4 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT4 is the input for interrupt task 4. (default) |
| | | 0010 | ePWM4 is the input for interrupt task 4. (EPWM4_INT) |
| | | xxx1 | No interrupt source for task 4. |
| 11 - 8 | PERINT3SEL | 0000 | Task 3 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT3 is the input for interrupt task 3. (default) |
| | | 0010 | ePWM3 is the input for interrupt task 3. (EPWM3_INT) |
| | | xxx1 | No interrupt source for task 3. |
| 7 - 4 | PERINT2SEL | 0000 | Task 2 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT2 is the input for interrupt task 2. (default) |
| | | 0010 | ePWM2 is the input for interrupt task 2. (EPWM2_INT) |
| | | xxx1 | No interrupt source for task 2. |
| 3 - 0 | PERINT1SEL | 0000 | Task 1 Peripheral Interrupt Input Select |
| | | 0000 | ADCINT1 is the input for interrupt task 1. (default) |
| | | 0010 | ePWM1 is the input for interrupt task 1. (EPWM1_INT) |
| | | xxx1 | No interrupt source |

⁽¹⁾ All values not shown are reserved.

⁽²⁾ This register is protected by EALLOW and the code security module.

4.3.4 Interrupt Enable Register (MIER)

Setting the bits in the interrupt enable register (MIER) allow an incoming interrupt or main CPU software to start the corresponding CLA task. Writing a 0 will block the task, but the interrupt request will still be latched in the flag register (MIFLG). Setting the MIER register bit to 0 while the corresponding task is executing will have no effect on the task. The task will continue to run until it hits the MSTOP instruction.

When a soft reset is issued, the MIER bits are cleared. There should always be at least a 1 SYSCLKOUT delay between issuing the soft reset and reconfiguring the MIER bits.

Figure 6. Interrupt Enable Register (MIER)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | 6 | | | | | 5 | | | | | 4 | | | | | 3 | | | | | 2 | | | | | 1 | | | | | 0 | | | | |
| INT8 | | | | | INT7 | | | | | INT6 | | | | | INT5 | | | | | INT4 | | | | | INT3 | | | | | INT2 | | | | | INT1 | | | | |
| R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | |

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

Table 6. Interrupt Enable Register (MIER) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|--------|--|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 1 | Task 8 Interrupt Enable Task 8 interrupt is disabled. (default) Task 8 interrupt is enabled. |
| 6 | INT7 | 0 1 | Task 7 Interrupt Enable Task 7 interrupt is disabled. (default) Task 7 interrupt is enabled. |
| 5 | INT6 | 0 1 | Task 6 Interrupt Enable Task 6 interrupt is disabled. (default) Task 6 interrupt is enabled. |
| 4 | INT5 | 0 1 | Task 5 Interrupt Enable Task 5 interrupt is disabled. (default) Task 5 interrupt is enabled. |
| 3 | INT4 | 0 1 | Task 4 Interrupt Enable Task 4 interrupt is disabled. (default) Task 4 interrupt is enabled. |
| 2 | INT3 | 0 1 | Task 3 Interrupt Enable Task 3 interrupt is disabled. (default) Task 3 interrupt is enabled. |
| 1 | INT2 | 0 1 | Task 2 Interrupt Enable Task 2 interrupt is disabled. (default) Task 2 interrupt is enabled. |
| 0 | INT1 | 0 1 | Task 1 Interrupt Enable Task 1 interrupt is disabled. (default) Task 1 interrupt is enabled. |

⁽¹⁾ This register is protected by EALLOW and the code security module.

4.3.5 Interrupt Flag Register (MIFR)

Each bit in the interrupt flag register corresponds to a CLA task. The corresponding bit is automatically set when the task request is received from the peripheral interrupt. The bit can also be set by the main CPU writing to the MIFRC register or using the IACK instruction to start the task. To use the IACK instruction to begin a task first enable this feature in the MCTL register. If the bit is already set when a new peripheral interrupt is received, then the corresponding overflow bit will be set in the MIOVF register.

The corresponding MIFR bit is automatically cleared when the task begins execution. This will occur if the interrupt is enabled in the MIER register and no other higher priority task is pending. The bits can also be cleared manually by writing to the MICLR register. Writes to the MIFR register are ignored.

Figure 7. Interrupt Flag Register (MIFR)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | 6 | | | | | 5 | | | | | 4 | | | | | 3 | | | | | 2 | | | | | 1 | | | | | 0 | | | | |
| INT8 | | | | | INT7 | | | | | INT6 | | | | | INT5 | | | | | INT4 | | | | | INT3 | | | | | INT2 | | | | | INT1 | | | | |
| R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | | | | | |

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

Table 7. Interrupt Flag Register (MIFR) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|-------|---|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 | Task 8 Interrupt Flag A task 8 interrupt is currently not flagged. (default) |
| | | 1 | A task 8 interrupt has been received and is pending execution. |
| 6 | INT7 | 0 | Task 7 Interrupt Flag A task 7 interrupt is currently not flagged. (default) |
| | | 1 | A task 7 interrupt has been received and is pending execution. |
| 5 | INT6 | 0 | Task 6 Interrupt Flag A task 6 interrupt is currently not flagged. (default) |
| | | 1 | A task 6 interrupt has been received and is pending execution. |
| 4 | INT5 | 0 | Task 5 Interrupt Flag A task 5 interrupt is currently not flagged. (default) |
| | | 1 | A task 5 interrupt has been received and is pending execution. |
| 3 | INT4 | 0 | Task 4 Interrupt Flag A task 4 interrupt is currently not flagged. (default) |
| | | 1 | A task 4 interrupt has been received and is pending execution. |
| 2 | INT3 | 0 | Task 3 Interrupt Flag A task 3 interrupt is currently not flagged. (default) |
| | | 1 | A task 3 interrupt has been received and is pending execution. |
| 1 | INT2 | 0 | Task 2 Interrupt Flag A task 2 interrupt is currently not flagged. (default) |
| | | 1 | A task 2 interrupt has been received and is pending execution. |
| 0 | INT1 | 0 | Task 1 Interrupt Flag A task 1 interrupt is currently not flagged. (default) |
| | | 1 | A task 1 interrupt has been received and is pending execution. |

⁽¹⁾ This register is protected by the code security module.

4.3.6 Interrupt Overflow Flag Register (MIOVF)

Each bit in the overflow flag register corresponds to a CLA task. The bit is set when an interrupt overflow event has occurred for the specific task. An overflow event occurs when the MIFR register bit is already set when a new interrupt is received from a peripheral source. The MIOVF bits are only affected by peripheral interrupt events. They do not respond to a task request by the main CPU IACK instruction or by directly setting MIFR bits. The overflow flag will remain latched and can only be cleared by writing to the overflow flag clear (MICLROVF) register. Writes to the MIOVF register are ignored.

Figure 8. Interrupt Overflow Flag Register (MIOVF)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | 6 | | | | | 5 | | | | | 4 | | | | | 3 | | | | | 2 | | | | | 1 | | | | | 0 | | | | |
| INT8 | | | | | INT7 | | | | | INT6 | | | | | INT5 | | | | | INT4 | | | | | INT3 | | | | | INT2 | | | | | INT1 | | | | |
| R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | | | | | |

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

Table 8. Interrupt Overflow Flag Register (MIOVF) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|--------|--|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 1 | Task 8 Interrupt Overflow Flag A task 8 interrupt overflow has not occurred. (default) A task 8 interrupt overflow has occurred. |
| 6 | INT7 | 0 1 | Task 7 Interrupt Overflow Flag A task 7 interrupt overflow has not occurred. (default) A task 7 interrupt overflow has occurred. |
| 5 | INT6 | 0 1 | Task 6 Interrupt Overflow Flag A task 6 interrupt overflow has not occurred. (default) A task 6 interrupt overflow has occurred. |
| 4 | INT5 | 0 1 | Task 5 Interrupt Overflow Flag A task 5 interrupt overflow has not occurred. (default) A task 5 interrupt overflow has occurred. |
| 3 | INT4 | 0 1 | Task 4 Interrupt Overflow Flag A task 4 interrupt overflow has not occurred. (default) A task 4 interrupt overflow has occurred. |
| 2 | INT3 | 0 1 | Task 3 Interrupt Overflow Flag A task 3 interrupt overflow has not occurred. (default) A task 3 interrupt overflow has occurred. |
| 1 | INT2 | 0 1 | Task 2 Interrupt Overflow Flag A task 2 interrupt overflow has not occurred. (default) A task 2 interrupt overflow has occurred. |
| 0 | INT1 | 0 1 | Task 1 Interrupt Overflow Flag A task 1 interrupt overflow has not occurred. (default) A task 1 interrupt overflow has occurred. |

⁽¹⁾ This register is protected by the code security module.

4.3.7 Interrupt Run Status Register (MIRUN)

The interrupt run status register (MIRUN) indicates which task is currently executing. Only one MIRUN bit will ever be set to a 1 at any given time. The bit is automatically cleared when the task completes and the respective interrupt is fed to the peripheral interrupt expansion (PIE) block of the device. This lets the main CPU know when a task has completed. The main CPU can stop a currently running task by writing to the MCTL[SOFTRESET] bit. This will clear the MIRUN flag and stop the task. In this case no interrupt will be generated to the PIE.

Figure 9. Interrupt Run Status Register (MIRUN)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|------|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | 6 | | | | | 5 | | | | | 4 | | | | | 3 | | | | | 2 | | | | | 1 | | | | | 0 | | | | |
| INT8 | | | | | INT7 | | | | | INT6 | | | | | INT5 | | | | | INT4 | | | | | INT3 | | | | | INT2 | | | | | INT1 | | | | |
| R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | R-0 | | | | | | | | | |

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

Table 9. Interrupt Run Status Register (MIRUN) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|--------|---|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 1 | Task 8 Run Status Task 8 is not executing. (default) Task 8 is executing. |
| 6 | INT7 | 0 1 | Task 7 Run Status Task 7 is not executing. (default) Task 7 is executing. |
| 5 | INT6 | 0 1 | Task 6 Run Status Task 6 is not executing. (default) Task 6 is executing. |
| 4 | INT5 | 0 1 | Task 5 Run Status Task 5 is not executing. (default) Task 5 is executing. |
| 3 | INT4 | 0 1 | Task 4 Run Status Task 4 is not executing. (default) Task 4 is executing. |
| 2 | INT3 | 0 1 | Task 3 Run Status Task 3 is not executing. (default) Task 3 is executing. |
| 1 | INT2 | 0 1 | Task 2 Run Status Task 2 is not executing. (default) Task 2 is executing. |
| 0 | INT1 | 0 1 | Task 1 Run Status Task 1 is not executing. (default) Task 1 is executing. |

⁽¹⁾ This register is protected by the code security module.

4.3.8 Interrupt Force Register (MIFRC)

The interrupt force register can be used by the main CPU to start tasks through software. Writing a 1 to a MIFRC bit will set the corresponding bit in the MIFR register. Writes of 0 are ignored and reads always return 0. The IACK #16bit operation can also be used to start tasks and has the same effect as the MIFRC register. To enable IACK to set MIFR bits you must first set the MCTL[IACKE] bit. Using IACK has the advantage of not having to first set the EALLOW bit. This allows the main CPU to efficiently trigger CLA tasks through software.

Figure 10. Interrupt Force Register (MIFRC)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|--|--|-------|--|--|--|--|--|--|-------|---|--|--|--|--|--|-------|--|--|--|--|--|--|-------|--|--|--|--|--|--|-------|--|--|--|--|--|--|-------|--|--|--|--|--|--|------|--|--|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | | | 6 | | | | | | | 5 | | | | | | | 4 | | | | | | | 3 | | | | | | | 2 | | | | | | | 1 | | | | | | | 0 | | | | | | |
| INT8 | | | | | | | INT7 | | | | | | | INT6 | | | | | | | INT5 | | | | | | | INT4 | | | | | | | INT3 | | | | | | | INT2 | | | | | | | INT1 | | | | | | |
| R/W-0 | | | | | | | R/W-0 | | | | | | | R/W-0 | | | | | | | R/W-0 | | | | | | | R/W-0 | | | | | | | R/W-0 | | | | | | | R/W-0 | | | | | | | | | | | | | |

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

Table 10. Interrupt Force Register (MIFRC) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|--------|--|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 1 | Task 8 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 8 interrupt. |
| 6 | INT7 | 0 1 | Task 7 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 7 interrupt. |
| 5 | INT6 | 0 1 | Task 6 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 6 interrupt. |
| 4 | INT5 | 0 1 | Task 5 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 5 interrupt. |
| 3 | INT4 | 0 1 | Task 4 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 4 interrupt. |
| 2 | INT3 | 0 1 | Task 3 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 3 interrupt. |
| 1 | INT2 | 0 1 | Task 2 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 2 interrupt. |
| 0 | INT1 | 0 1 | Task 1 Interrupt Force This bit always reads back 0 and writes of 0 have no effect. Write a 1 to force the task 1 interrupt. |

⁽¹⁾ This register is protected by EALLOW and the code security module.

4.3.9 Interrupt Flag Clear Register (MICLR)

Normally bits in the MIFR register are automatically cleared when a task begins. The interrupt flag clear register can be used to instead manually clear bits in the interrupt flag (MIFR) register. Writing a 1 to a MICLR bit will clear the corresponding bit in the MIFR register. Writes of 0 are ignored and reads always return 0.

Figure 11. Interrupt Flag Clear Register (MICLR)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | 6 | | | | | 5 | | | | | 4 | | | | | 3 | | | | | 2 | | | | | 1 | | | | | 0 | | | | |
| INT8 | | | | | INT7 | | | | | INT6 | | | | | INT5 | | | | | INT4 | | | | | INT3 | | | | | INT2 | | | | | INT1 | | | | |
| R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | |

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

Table 11. Interrupt Flag Clear Register (MICLR) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|--------|--|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 1 | Task 8 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 8 interrupt flag. |
| 6 | INT7 | 0 1 | Task 7 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 7 interrupt flag. |
| 5 | INT6 | 0 1 | Task 6 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 6 interrupt flag. |
| 4 | INT5 | 0 1 | Task 5 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 5 interrupt flag. |
| 3 | INT4 | 0 1 | Task 4 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 4 interrupt flag. |
| 2 | INT3 | 0 1 | Task 3 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 3 interrupt flag. |
| 1 | INT2 | 0 1 | Task 2 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 2 interrupt flag. |
| 0 | INT1 | 0 1 | Task 1 Interrupt Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 1 interrupt flag. |

⁽¹⁾ This register is protected by EALLOW and the code security module.

4.3.10 Interrupt Overflow Flag Clear Register (MICLROVF)

Overflow flag bits in the MIOVF register are latched until manually cleared using the MICLROVF register. Writing a 1 to a MICLROVF bit will clear the corresponding bit in the MIOVF register. Writes of 0 are ignored and reads always return 0.

Figure 12. Interrupt Overflow Flag Clear Register (MICLROVF)

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|-------|--|--|--|--|------|--|--|--|--|
| 15 | | | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | | | | | | | | | | | |
| Reserved | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| R -0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | | | | | 6 | | | | | 5 | | | | | 4 | | | | | 3 | | | | | 2 | | | | | 1 | | | | | 0 | | | | |
| INT8 | | | | | INT7 | | | | | INT6 | | | | | INT5 | | | | | INT4 | | | | | INT3 | | | | | INT2 | | | | | INT1 | | | | |
| R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | R/W-0 | | | | | | | | | |

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

Table 12. Interrupt Overflow Flag Clear Register (MICLROVF) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|------|----------|--------|--|
| 15-8 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 7 | INT8 | 0 1 | Task 8 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 8 interrupt overflow flag. |
| 6 | INT7 | 0 1 | Task 7 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 7 interrupt overflow flag. |
| 5 | INT6 | 0 1 | Task 6 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 6 interrupt overflow flag. |
| 4 | INT5 | 0 1 | Task 5 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 5 interrupt overflow flag. |
| 3 | INT4 | 0 1 | Task 4 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 4 interrupt overflow flag. |
| 2 | INT3 | 0 1 | Task 3 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 3 interrupt overflow flag. |
| 1 | INT2 | 0 1 | Task 2 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 2 interrupt overflow flag. |
| 0 | INT1 | 0 1 | Task 1 Interrupt Overflow Flag Clear This bit always reads back 0 and writes of 0 have no effect. Write a 1 to clear the task 1 interrupt overflow flag. |

⁽¹⁾ This register is protected by EALLOW and the code security module.

4.4 Execution Registers

The CLA program counter is initialized by the appropriate MVECTx register when an interrupt is received and a task begins execution. The MPC points to the instruction in the decode 2 (D2) stage of the CLA pipeline. After a MSTOP operation, if no other tasks are pending, the MPC will remain pointing to the MSTOP instruction. The MPC register can be read by the main C28x CPU for debug purposes. The main CPU cannot write to MPC.

4.4.1 MPC Register

The MPC register is described in [Figure 13](#) and described in [Table 13](#).

Figure 13. Program Counter (MPC)

| | | | |
|----------|----|-----|---|
| 15 | 12 | 11 | 0 |
| Reserved | | MPC | |
| R-0 | | R-0 | |

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

Table 13. Program Counter (MPC) Field Descriptions

| Bits | Name | Value | Description ⁽¹⁾ |
|-------|----------|-------------|---|
| 15-12 | Reserved | | Any writes to these bit(s) must always have a value of 0. |
| 11-0 | MPC | 0000 - 0FFF | Points to the instruction currently in the decode 2 phase of the CLA pipeline. The value is the offset from the first address in the CLA program space. |

⁽¹⁾ This register is protected by the code security module. The main CPU can read this register for debug purposes but it can not write to it.

4.4.2 MSTF Register

The CLA status register (MSTF) reflects the results of different operations. These are the basic rules for the flags:

- Zero and negative flags are cleared or set based on:
 - floating-point moves to registers
 - the result of compare, minimum, maximum, negative and absolute value operations
 - the integer result of operations such as MMOV16, MAND32, MOR32, MXOR32, MCMP32, MASR32, MLRS32
- Overflow and underflow flags are set by floating-point 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.

The MSTF register is shown in [Figure 14](#) and described in [Table 14](#).

Figure 14. CLA Status Register (MSTF)

| | | | |
|----------|----------|----------|----------|
| 31 | 24 | 23 | 16 |
| Reserved | | | |
| RPC | | | |
| R/W-0 | | | |
| 15 | 12 | 11 | 10 |
| 9 | 8 | 7 | 6 |
| 5 | 4 | 3 | 2 |
| 1 | 0 | | |
| RPC | MEALLOW | Reserved | RND32 |
| Reserved | Reserved | TF | Reserved |
| ZF | NF | LUF | LVF |
| R/W-0 | R/W-0 | R-0 | R/W-0 |
| R-0 | R/W-0 | R/W-0 | R/W-0 |

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

Table 14. CLA Status (MSTF) Register Field Descriptions

| Bits | Field | Value | Description ⁽¹⁾ |
|---------|----------|--------|--|
| 31 - 24 | Reserved | 0 | Reserved for future use |
| 23 - 12 | RPC | | Return program counter. The RPC is used to save and restore the MPC address by the MCCNDD and MRCNDD operations. |
| 11 | MEALLOW | 0 1 | This bit enables and disables CLA write access to EALLOW protected registers. This is independent of the state of the EALLOW bit in the main CPU status register. This status bit can be saved and restored by the MMOV32 STF instruction. The CLA cannot write to EALLOW protected registers. This bit is cleared by the MEDIS CLA instruction. The CLA is allowed to write to EALLOW protected registers. This bit is set by the MEALLOW CLA instruction. |
| 10 | Reserved | 0 | Any writes to these bit(s) must always have a value of 0. |
| 9 | RND32 | 0 1 | Round 32-bit Floating-Point Mode Use the MSETFLG and MMOV32 MSTF instructions to change the rounding mode. If this bit is zero, the MMPYF32, MADDF32 and MSUBF32 instructions will round to zero (truncate). If this bit is one, the MMPYF32, MADDF32 and MSUBF32 instructions will round to the nearest even value. |
| 8 - 7 | Reserved | 0 | Reserved for future use |
| 6 | TF | 0 1 | Test Flag The TESTTF instruction can modify this flag based on the condition tested. The MSETFLG and MMOV32 MSTF, mem32 instructions can also be used to modify this flag. The condition tested with the TESTTF instruction is false. The condition tested with the TESTTF instruction is true. |
| 5 - 4 | Reserved | | These two bits may change based on integer results. These flags are not, however, used by the CLA and therefore marked as reserved. |
| 3 | ZF | 0 1 | Zero Flag ^{(2) (3)} <ul style="list-style-type: none"> Instructions that modify this flag based on the floating-point value stored in the destination register: MMOV32, MMOVD32, MOVDD32, ABSF32, MNEGF32 Instructions that modify this flag based on the floating-point result of the operation: MCMPF32, MMAXF32, and MMINF32 Instructions that modify this flag based on the integer result of the operation: MMOV16, MAND32, MOR32, MXOR32, MCMP32, MASR32, MLSR32 and MLSL32 The MSETFLG and MMOV32 MSTF, mem32 instructions can also be used to modify this flag The value is not zero. The value is zero. |
| 2 | NF | 0 1 | Negative Flag ^{(2) (3)} <ul style="list-style-type: none"> Instructions that modify this flag based on the floating-point value stored in the destination register: MMOV32, MMOVD32, MOVDD32, ABSF32, MNEGF32 Instructions that modify this flag based on the floating-point result of the operation: MCMPF32, MMAXF32, and MMINF32 Instructions that modify this flag based on the integer result of the operation: MMOV16, MAND32, MOR32, MXOR32, MCMP32, MASR32, MLSR32 and MLSL32 The MSETFLG and MMOV32 MSTF, mem32 instructions can also be used to modify this flag. The value is not negative. The value is negative. |

⁽¹⁾ This register is protected by the code security module. The main CPU can read this register for debug purposes but it can not write to it.

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

⁽³⁾ A DeNorm floating-point value is treated as a positive zero value when configuring the ZF and NF flags.

Table 14. CLA Status (MSTF) Register Field Descriptions (continued)

| Bits | Field | Value | Description ⁽¹⁾ |
|------|-------|-------|--|
| 1 | LUF | | Latched Underflow Flag The following instructions will set this flag to 1 if an underflow occurs: MMPYF32, MADDF32, MSUBF32, MMACF32, MEINVF32, MEISQRTF32 The MSETFLG and MMOV32 MSTF, mem32 instructions can also be used to modify this flag. |
| | | 0 | An underflow condition has not been latched. |
| | | 1 | An underflow condition has been latched. |
| 0 | LVF | | Latched Overflow Flag The following instructions will set this flag to 1 if an overflow occurs: MMPYF32, MADDF32, MSUBF32, MMACF32, MEINVF32, MEISQRTF32 The MSETFLG and MMOV32 MSTF, mem32 instructions can also be used to modify this flag. |
| | | 0 | An overflow condition has not been latched. |
| | | 1 | An overflow condition has been latched. |

5 Pipeline

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

5.1 Pipeline Overview

The CLA pipeline is very similar to the C28x pipeline. The pipeline has eight stages:

- **Fetch 1 (F1)**
During the F1 stage the program read address is placed on the CLA program address bus.
- **Fetch 2 (F2)**
During the F2 stage the instruction is read using the CLA program data bus.
- **Decode 1 (D1)**
During D1 the instruction is decoded.
- **Decode 2 (D2)**
Generate the data read address. Changes to MAR0 and MAR1 due to post-increment using indirect addressing takes place in the D2 phase. Conditional branch decisions are also made at this stage based on the MSTF register flags.
- **Read 1 (R1)**
Place the data read address on the CLA data-read address bus. If a memory conflict exists, the R1 stage will be stalled.
- **Read 2 (R2)**
Read the data value using the CLA data read data bus.
- **Execute (EXE)**
Execute the operation. Changes to MAR0 and MAR1 due to loading an immediate value or value from memory take place in this stage.
- **Write (W)**
Place the write address and write data on the CLA write data bus. If a memory conflict exists, the W stage will be stalled.

5.2 CLA Pipeline Alignment

The majority of the CLA instructions do not require any special pipeline considerations. This section lists the few operations that do require special consideration.

- **Write Followed by Read**
In both the CLA pipeline the read operation occurs before the write. This means that if a read operation immediately follows a write, then the read will complete first as shown in [Table 15](#). In most cases this does not cause a problem since the contents of one memory location does not depend on the state of another. For accesses to peripherals where a write to one location can affect the value in another location the code must wait for the write to complete before issuing the read as shown in [Table 16](#).
This behavior is different for the 28x CPU. For the 28x CPU any write followed by read to the same location is protected by what is called write-followed-by-read protection. This protection automatically stalls the pipeline so that the write will complete before the read. In addition some peripheral frames are protected such that a 28x CPU write to one location within the frame will always complete before a read to the frame. The CLA does not have this protection mechanism. Instead the code must wait to perform the read.

Table 15. Write Followed by Read - Read Occurs First

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|----------------------|----|----|----|----|----|----|----|----|
| I1 MMOV16 @Reg1, MR3 | I1 | | | | | | | |
| I2 MMOV16 MR2, @Reg2 | I2 | I1 | | | | | | |
| | | I2 | I1 | | | | | |
| | | | I2 | I1 | | | | |
| | | | | I2 | I1 | | | |
| | | | | | I2 | I1 | | |
| | | | | | | I2 | I1 | |
| | | | | | | | I2 | I1 |

Table 16. Write Followed by Read - Write Occurs First

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|----------------------|----|----|----|----|----|----|----|----|
| I1 MMOV16 @Reg1, MR3 | I1 | | | | | | | |
| I2 | I2 | I1 | | | | | | |
| I3 | I3 | I2 | I1 | | | | | |
| I4 | I4 | I3 | I2 | I1 | | | | |
| I5 MMOV16 MR2, @Reg2 | I5 | I4 | I3 | I2 | I1 | | | |
| | | I5 | I4 | I3 | I2 | I1 | | |
| | | | I5 | I4 | I3 | I2 | I1 | |
| | | | | I5 | I4 | I3 | I2 | I1 |
| | | | | | I5 | I4 | I3 | |
| | | | | | | I5 | I4 | |
| | | | | | | | I5 | |

- Delayed Conditional instructions: MBCNDD, MCCNDD and MRCNDD**

Referring to [Example 1](#), the following applies to delayed conditional instructions:

- **I1**

I1 is the last instruction that can effect the CNDF flags for the branch, call or return instruction. The CNDF flags are tested in the D2 phase of the pipeline. That is, a decision is made whether to branch or not when [MBCNDD](#), [MCCNDD](#) or [MRCNDD](#) is in the D2 phase.

- **I2, I3 and I4**

The three instructions proceeding MBCNDD can change MSTF flags but will have no effect on whether the MBCNDD instruction branches or not. This is because the flag modification will occur after the D2 phase of the branch, call or return instruction. These three instructions must not be a [MSTOP](#), [MDEBUGSTOP](#), [MBCNDD](#), [MCCNDD](#) or [MRCNDD](#).

- **I5, I6 and I7**

The three instructions following a branch, call or return are always executed irrespective of whether the condition is true or not. These instructions must not be MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.

For a more detailed description refer to the functional description for [MBCNDD](#), [MCCNDD](#) and [MRCNDD](#).

Example 1. Code Fragment For MBCNDD, MCCNDD or MRCNDD

```

<Instruction 1>    ; I1 Last instruction that can affect flags for
                  ; the branch, call or return operation

<Instruction 2>    ; I2 Cannot be stop, branch, call or return
<Instruction 3>    ; I3 Cannot be stop, branch, call or return
<Instruction 4>    ; I4 Cannot be stop, branch, call or return

<branch/call/ret> ; MBCNDD, MCCNDD or MRCNDD

                  ; I5-I7: Three instructions after are always
                  ; executed whether the branch/call or return is
                  ; taken or not

<Instruction 5>    ; I5 Cannot be stop, branch, call or return
<Instruction 6>    ; I6 Cannot be stop, branch, call or return
<Instruction 7>    ; I7 Cannot be stop, branch, call or return

<Instruction 8>    ; I8
<Instruction 9>    ; I9
....

```

- **Stop or Halting a Task: MSTOP and MDEBUGSTOP**

The MSTOP and MDEBUGSTOP instructions cannot be placed three instructions before or after a conditional branch, call or return instruction (MBCNDD, MCCNDD or MRCNDD). Refer to Example 1. To single-step through a branch/call or return, insert the MDEBUGSTOP at least four instructions back and step from there.

- **Loading MAR0 or MAR1**

A load of auxiliary register MAR0 or MAR1 will occur in the EXE phase of the pipeline. Any post increment of MAR0 or MAR1 using indirect addressing will occur in the D2 phase of the pipeline. Referring to Example 2, the following applies when loading the auxiliary registers:

- **I1 and I2**
The two instructions following the load instruction will use the value in MAR0 or MAR1 before the update occurs.
- **I3**
Loading of an auxiliary register occurs in the EXE phase while updates due to post-increment addressing occur in the D2 phase. Thus I3 cannot use the auxiliary register or there will be a conflict. In the case of a conflict, the update due to address-mode post increment will win and the auxiliary register will not be updated with #_X.
- **I4**
Starting with the 4th instruction MAR0 or MAR1 will have the new value.

Example 2. Code Fragment for Loading MAR0 or MAR1

```

; Assume MAR0 is 50 and #_X is 20

MMOVI16 MAR0, #_X ; Load MAR0 with address of X (20)
<Instruction 1>    ; I1 Will use the old value of MAR0 (50)
<Instruction 2>    ; I2 Will use the old value of MAR0 (50)
<Instruction 3>    ; I3 Cannot use MAR0
<Instruction 4>    ; I4 Will use the new value of MAR0 (20)
<Instruction 5>    ; I5 Will use the new value of MAR0 (20)
....

```

5.2.1 ADC Early Interrupt to CLA Response

The 2803x ADC offers the option to generate an early interrupt pulse when the ADC begins conversion.

This option is selected by setting the ADCCTL1[INTPULSEPOS] bit as documented in the TMS320x2802x, x2803x Piccolo Analog-to-Digital Converter and Comparator Reference Guide ([SPRUGE5](#)). If this option is used to start a CLA task then the CLA will be able to read the result as soon as the conversion completes and the ADC result register updates. This just-in-time sampling along with the low interrupt response of the CLA enable faster system response and higher frequency control loops.

The timing for the ADC conversion is shown in the ADC Reference Guide timing diagrams. From a CLA perspective, the pipeline activity is shown in [Table 17](#). The 8th instruction is in the R2 phase just in time to read the result register. While the first 7 instructions in the task (I1 to I7) will enter the R2 phase of the pipeline too soon to read the conversion, they can be efficiently used for pre-processing calculations needed by the task.

Table 17. ADC to CLA Early Interrupt Response

| ADC Activity | CLA Activity | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|---------------------|--------------------|----|----|----|----|----|----|---|---|
| Sample | | | | | | | | | |
| Sample | | | | | | | | | |
| ... | | | | | | | | | |
| Sample | | | | | | | | | |
| Conversion (1) | Interrupt Received | | | | | | | | |
| Conversion (2) | Task Startup | | | | | | | | |
| Conversion (3) | Task Startup | | | | | | | | |
| Conversion (4) | I1 | I1 | | | | | | | |
| Conversion (5) | I2 | I2 | I1 | | | | | | |
| Conversion (6) | I3 | I3 | I2 | I1 | | | | | |
| Conversion (7) | I4 | I4 | I3 | I2 | I1 | | | | |
| Conversion (8) | I5 | I5 | I4 | I3 | I2 | I1 | | | |
| Conversion (9) | I6 | I6 | I5 | I4 | I3 | I2 | I1 | | |
| Conversion (10) | I7 | I7 | I6 | I5 | I4 | I3 | I2 | | |
| Conversion (11) | I8 Read ADC RESULT | I8 | I7 | I6 | I5 | I4 | I3 | | |
| Conversion (12) | | | I8 | I7 | I6 | I5 | I4 | | |
| Conversion (13) | | | | I8 | I7 | I6 | I5 | | |
| Conversion Complete | | | | | I8 | I7 | I6 | | |
| RESULT Latched | | | | | | I8 | I7 | | |
| RESULT Available | | | | | | | I8 | | |

5.3 Parallel Instructions

Parallel instructions are single opcodes that perform two operations in parallel. The following types of parallel instructions are available: math operation in parallel with a move operation, or two math operations in parallel. **Both operations complete in a single cycle and there are no special pipeline alignment requirements.**

Example 3. Math Operation with Parallel Load

```

; MADDF32 || MMOV32 instruction: 32-bit floating-point add with parallel move
; MADDF32 is a 1 cycle operation
; MMOV32 is a 1 cycle operation
; MADDF32 MR0, MR1, #2 ; MR0 = MR1 + 2,
; || MMOV32 MR1, @Val ; MR1 gets the contents of Val
; <-- MMOV32 completes here (MR1 is valid)
; <-- DDF32 completes here (MR0 is valid)
; Any instruction, can use MR1 and/or MR0

MMPYF32 MR0, MR0, MR1

```

Example 4. Multiply with Parallel Add

Example 4. Multiply with Parallel Add (continued)

```

; MMPYF32 || MADDF32 instruction: 32-bit floating-point multiply with parallel add
; MMPYF32 is a 1 cycle operation
; MADDF32 is a 1 cycle operation
    MMPYF32 MR0, MR1, MR3      ; MR0 = MR1 * MR3
    || MADDF32 MR1, MR2, MR0    ; MR1 = MR2 + MR0 (Uses value of MR0 before MMPYF32)
                                ; <-- MMPYF32 and MADDF32 complete here (MR0 and MR1 are valid)
    MMPYF32 MR1, MR1, MR0      ; Any instruction, can use MR1 and/or MR0

```

6 Instruction Set

This section describes the assembly language instructions of the control law accelerator. Also described are parallel operations, conditional operations, resource constraints, and addressing modes. The instructions listed here are independent from C28x and C28x+FPU instruction sets.

6.1 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. CLA 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 CLA are given in [Table 18](#).

Table 18. 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 |
| #SHIFT | Immediate value of 1 to 32 used for arithmetic and logical shifts. |
| addr | Opcode field indicating the addressing mode |
| CNDF | Condition to test the flags in the MSTF register |
| FLAG | Selected flags from MSTF register (OR) 8 bit mask indicating which floating-point status flags to change |
| MAR0 | auxiliary register 0 |
| MAR1 | auxiliary register 1 |
| MARx | Either MAR0 or MAR1 |
| mem16 | 16-bit memory location accessed using direct or indirect addressing modes |
| mem32 | 32-bit memory location accessed using direct or indirect addressing modes |
| MRa | MR0 to MR3 registers |
| MRb | MR0 to MR3 registers |
| MRc | MR0 to MR3 registers |
| MRd | MR0 to MR3 registers |
| MRe | MR0 to MR3 registers |
| MRf | MR0 to MR3 registers |
| MSTF | CLA Floating-point Status Register |
| shift | Opcode field indicating the number of bits to shift. |
| VALUE | Flag value of 0 or 1 for selected flag (OR) 8 bit mask indicating the flag value; 0 or 1 |

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 19. INSTRUCTION dest, source1, source2 Short Description

| | Description |
|--------------|--|
| 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 |
| Opcode | This section shows the opcode for the instruction |
| Description | Detailed description of the instruction execution is described. Any constraints on the operands imposed by the processor or the assembler are discussed. |
| Restrictions | Any constraints on the operands or use of the instruction imposed by the processor are discussed. |
| Pipeline | This section describes the instruction in terms of pipeline cycles as described in Section 5 |
| Example | 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 CLA is correctly configured and the main CPU has passed it data. |
| Operands | 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). |

6.2 Addressing Modes and Encoding

The CLA uses the same address to access data and registers as the main CPU. For example if the main CPU accesses an ePWM register at address 0x00 6800, then the CLA will access it using address 0x6800. Since all CLA accessible memory and registers are within the low 64k x 16 of memory, only the low 16-bits of the address are used by the CLA.

To address the CLA data memory, message RAMs and shared peripherals, the CLA supports two addressing modes:

- Direct addressing mode: Uses the address of the variable or register directly.
- Indirect addressing with 16-bit post increment. This mode uses either XAR0 or XAR1.

The CLA does not use a data page pointer or a stack pointer. The two addressing modes are encoded as shown in [Table 20](#).

Table 20. Addressing Modes

| Addressing Mode | 'addr' Opcode Field Encode ⁽¹⁾ | Description |
|------------------------|---|---|
| @dir | 0000 | Direct Addressing Mode Example 1: MMOV32 MR1, @_VarA Example 2: MMOV32 MR1, @_EPwm1Regs.CMPA.all In this case the 'mmmm mmmm mmmm mmmm' opcode field will be populated with the 16-bit address of the variable. This is the low 16-bits of the address that you would use to access the variable using the main CPU. For example @_VarA will populate the address of the variable VarA, and @_EPwm1Regs.CMPA.all will populate the address of the CMPA register. |
| *MAR0[#imm16]++ | 0001 | MAR0 Indirect Addressing with 16-bit Immediate Post Increment *MAR1[#imm16]++ 0010 MAR1 Indirect Addressing with 16-bit Immediate Post Increment addr = MAR0 (or MAR1) Access memory using the address stored in MAR0 (or MAR1). MAR0 (or MAR1) += Then post increment MAR0 (or MAR1) by #imm16. #imm16 Example 1: MMOV32 MR0, *MAR0[2]++ Example 2: MMOV32 MR1, *MAR1[-2]++ For a post increment of 0 the assembler will accept both *MAR0 and *MAR0[0]++. The 'mmmm mmmm mmmm mmmm' opcode field will be populated with the signed 16-bit pointer offset. For example if #imm16 is 2, then the opcode field will be 0x0002. Likewise if #imm16 is -2, then the opcode field will be 0xFFFF. If addition of the 16-bit immediate causes overflow, then the value will wrap around on a 16-bit boundary. |

⁽¹⁾ Values not shown are reserved.

Encoding for the shift fields in the MASR32, MLSR32 and ML32 instructions is shown in [Table 21](#)

Table 21. Shift Field Encoding

| Shift Value | 'shift' Opcode Field Encode |
|-------------|-----------------------------|
| 1 | 0000 |
| 2 | 0001 |
| 3 | 0010 |
| | |
| 32 | 1111 |

[Table 22](#) shows the condition field encoding for conditional instructions such as MNEGF, MSWAPF, MBCNDD, MCCNDD and MRCNDD

Table 22. Condition Field Encoding

| Encode ⁽¹⁾ | CNDF | Description | MSTF 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 OR 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 and NF flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

6.3 Instructions

The instructions are listed alphabetically, preceded by a summary.

Table 23. Instructions

| Title | Page |
|---|------|
| MABSF32 MRa, MRb — 32-bit Floating-Point Absolute Value | 43 |
| MADD32 MRa, MRb, MRc — 32-bit Integer Add | 44 |
| MADDF32 MRa, #16FHi, MRb — 32-bit Floating-Point Addition | 45 |
| MADDF32 MRa, MRb, MRc — 32-bit Floating-Point Addition | 48 |
| MADDF32 MRd, MRe, MRf MMOV32 mem32, MRa — 32-bit Floating-Point Addition with Parallel Move | 49 |
| MADDF32 MRd, MRe, MRf MMOV32 MRa, mem32 — 32-bit Floating-Point Addition with Parallel Move | 50 |
| MAND32 MRa, MRb, MRc — Bitwise AND | 52 |
| MASR32 MRa, #SHIFT — Arithmetic Shift Right | 53 |
| MBCNDD 16BitDest {, CNDF} — Branch Conditional Delayed | 54 |
| MCCNDD 16BitDest {, CNDF} — Call Conditional Delayed | 59 |
| MCMP32 MRa, MRb — 32-bit Integer Compare for Equal, Less Than or Greater Than | 63 |
| MCMPF32 MRa, MRb — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than | 64 |
| MCMPF32 MRa, #16FHi — 32-bit Floating-Point Compare for Equal, Less Than or Greater Than | 65 |
| MDEBUGSTOP — Debug Stop Task | 67 |
| MEALLOW — Enable CLA Write Access to EALLOW Protected Registers | 68 |
| MEDIS — Disable CLA Write Access to EALLOW Protected Registers | 69 |
| MEINVF32 MRa, MRb — 32-bit Floating-Point Reciprocal Approximation | 70 |
| MEISQRTF32 MRa, MRb — 32-bit Floating-Point Square-Root Reciprocal Approximation | 72 |
| MF32TOI16 MRa, MRb — Convert 32-bit Floating-Point Value to 16-bit Integer | 74 |
| MF32TOI16R MRa, MRb — Convert 32-bit Floating-Point Value to 16-bit Integer and Round | 75 |
| MF32TOI32 MRa, MRb — Convert 32-bit Floating-Point Value to 32-bit Integer | 76 |
| MF32TOUI16 MRa, MRb — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer | 77 |
| MF32TOUI16R MRa, MRb — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round | 78 |
| MF32TOUI32 MRa, MRb — Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer | 79 |
| MFRACF32 MRa, MRb — Fractional Portion of a 32-bit Floating-Point Value | 80 |
| MI16TOF32 MRa, MRb — Convert 16-bit Integer to 32-bit Floating-Point Value | 81 |
| MI16TOF32 MRa, mem16 — Convert 16-bit Integer to 32-bit Floating-Point Value | 82 |
| MI32TOF32 MRa, mem32 — Convert 32-bit Integer to 32-bit Floating-Point Value | 83 |
| MI32TOF32 MRa, MRb — Convert 32-bit Integer to 32-bit Floating-Point Value | 84 |
| MLSL32 MRa, #SHIFT — Logical Shift Left | 85 |
| MLSR32 MRa, #SHIFT — Logical Shift Right | 86 |
| MMACF32 MR3, MR2, MRd, MRe, MRf MMOV32 MRa, mem32 — 32-bit Floating-Point Multiply and Accumulate with Parallel Move | 87 |
| MMAXF32 MRa, MRb — 32-bit Floating-Point Maximum | 90 |
| MMAXF32 MRa, #16FHi — 32-bit Floating-Point Maximum | 91 |
| MMINF32 MRa, MRb — 32-bit Floating-Point Minimum | 92 |
| MMINF32 MRa, #16FHi — 32-bit Floating-Point Minimum | 93 |
| MMOV16 MARx, mem16 — Load MAR1 with 16-bit Value | 97 |
| MMOV16 mem16, MARx — Move 16-bit Auxiliary Register Contents to Memory | 99 |
| MMOV16 mem16, MRa — Move 16-bit Floating-Point Register Contents to Memory | 100 |
| MMOV32 mem32, MRa — Move 32-bit Floating-Point Register Contents to Memory | 101 |
| MMOV32 mem32, MSTF — Move 32-bit MSTF Register to Memory | 102 |
| MMOV32 MRa, mem32 {, CNDF} — Conditional 32-bit Move | 103 |
| MMOV32 MRa, MRb {, CNDF} — Conditional 32-bit Move | 105 |
| MMOV32 MSTF, mem32 — Move 32-bit Value from Memory to the MSTF Register | 107 |
| MMOVD32 MRa, mem32 — Move 32-bit Value from Memory with Data Copy | 108 |

Table 23. Instructions (continued)

| | |
|---|-----|
| MMOVIZ MRa, #16FHi —Load the Upper 16-bits of a 32-bit Floating-Point Register | 112 |
| MMOVZ16 MRa, mem16 —Load MRx with 16-bit Value | 113 |
| MMPYF32 MRa, MRb, MRc —32-bit Floating-Point Multiply | 115 |
| MMPYF32 MRa, #16FHi, MRb —32-bit Floating-Point Multiply | 116 |
| MMPYF32 MRa, MRb, #16FHi —32-bit Floating-Point Multiply | 118 |
| MMPYF32 MRa, MRb, MRc MADDF32 MRd, MRe, MRf —32-bit Floating-Point Multiply with Parallel Add | 120 |
| MMPYF32 MRd, MRe, MRf MMOV32 MRa, mem32 —32-bit Floating-Point Multiply with Parallel Move..... | 122 |
| MMPYF32 MRd, MRe, MRf MMOV32 mem32, MRa —32-bit Floating-Point Multiply with Parallel Move..... | 124 |
| MMPYF32 MRa, MRb, MRc MSUBF32 MRd, MRe, MRf —32-bit Floating-Point Multiply with Parallel Subtract | 125 |
| MNEGF32 MRa, MRb{, CNDF} —Conditional Negation | 126 |
| MNOP —No Operation..... | 128 |
| MOR32 MRa, MRb, MRc —Bitwise OR | 129 |
| MRCNDD {CNDF} —Return Conditional Delayed..... | 130 |
| MSETFLG FLAG, VALUE —Set or clear selected floating-point status flags | 133 |
| MSTOP —Stop Task | 134 |
| MSUB32 MRa, MRb, MRc —32-bit Integer Subtraction..... | 136 |
| MSUBF32 MRa, MRb, MRc —32-bit Floating-Point Subtraction | 137 |
| MSUBF32 MRa, #16FHi, MRb —32-bit Floating Point Subtraction..... | 138 |
| MSUBF32 MRd, MRe, MRf MMOV32 MRa, mem32 —32-bit Floating-Point Subtraction with Parallel Move..... | 139 |
| MSUBF32 MRd, MRe, MRf MMOV32 mem32, MRa —32-bit Floating-Point Subtraction with Parallel Move..... | 141 |
| MSWAPF MRa, MRb {, CNDF} —Conditional Swap | 142 |
| MTESTTF CNDF —Test MSTF Register Flag Condition..... | 144 |
| MUI16TOF32 MRa, mem16 —Convert unsigned 16-bit integer to 32-bit floating-point value | 146 |
| MUI16TOF32 MRa, MRb —Convert unsigned 16-bit integer to 32-bit floating-point value | 147 |
| MUI32TOF32 MRa, mem32 —Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value | 148 |
| MUI32TOF32 MRa, MRb —Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value..... | 149 |
| MXOR32 MRa, MRb, MRc —Bitwise Exclusive Or | 150 |

MABSF32 MRa, MRb 32-bit Floating-Point Absolute Value

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 0010 0000
```

Description

The absolute value of MRb is loaded into MRa. Only the sign bit of the operand is modified by the MABSF32 instruction.

```
if (MRb < 0) {MRa = -MRb};
else {MRa = MRb};
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified as follows:

```
NF = 0;
ZF = 0;
if ( MRa(30:23) == 0) ZF = 1;
```

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ MR0, #-2.0 ; MR0 = -2.0 (0xC0000000)
MABSF32 MR0, MR0 ; MR0 = 2.0 (0x40000000), ZF = NF = 0

MMOVIZ MR0, #5.0 ; MR0 = 5.0 (0x40A00000)
MABSF32 MR0, MR0 ; MR0 = 5.0 (0x40A00000), ZF = NF = 0

MMOVIZ MR0, #0.0 ; MR0 = 0.0
MABSF32 MR0, MR0 ; MR0 = 0.0 ZF = 1, NF = 0
```

See also

[MNEGF32 MRa, MRb {, CNDF}](#)

MADD32 MRa, MRb, MRc 32-bit Integer Add

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point destination register (MR0 to MR3) |
| MRc | CLA floating-point destination register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 000cc bbaa
MSW: 0111 1110 1100 0000
```

Description

32-bit integer addition of MRb and MRc.

```
MARa(31:0) = MARb(31:0) + MRc(31:0);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; };
```

Pipeline

This is a single-cycle instruction.

Example

```
; Given A = (int32)1
; B = (int32)2
; C = (int32)-7
;
; Calculate Y2 = A + B + C
;
_ClalTask1:
    MMOV32 MR0, @_A      ; MR0 = 1 (0x00000001)
    MMOV32 MR1, @_B      ; MR1 = 2 (0x00000002)
    MMOV32 MR2, @_C      ; MR2 = -7 (0xFFFFFFFF9)
    MADD32 MR3, MR0, MR1 ; A + B
    MADD32 MR3, MR2, MR3 ; A + B + C = -4 (0xFFFFFFF4)
    MMOV32 @_y2, MR3      ; Store y2
    MSTOP                 ; end of task
```

See also

[MAND32 MRa, MRb, MRc](#)
[MASR32 MRa, #SHIFT](#)
[MLSL32 MRa, #SHIFT](#)
[MLSR32 MRa, #SHIFT](#)
[MOR32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)
[MSUB32 MRa, MRb, MRc](#)

MADDF32 MRa, #16FHi, MRb 32-bit Floating-Point Addition

Operands

| | |
|--------|---|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| #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. |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: I I I I I I I I I I I I I I
MSW: 0 1 1 1 0 1 1 1 1 1 0 0 b b a a
```

Description

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

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

```
MRa = MRb + #16FHi:0;
```

This instruction can also be written as MADDF32 MRa, MRb, #16FHi.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MADDF32 generates an underflow condition.
- LVF = 1 if MADDF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```
; Add to MR1 the value 2.0 in 32-bit floating-point format
; Store the result in MR0
MADDF32 MR0, #2.0, MR1    ; MR0 = 2.0 + MR1

; Add to MR3 the value -2.5 in 32-bit floating-point format
; Store the result in MR2
MADDF32 MR2, #-2.5, MR3   ; MR2 = -2.5 + MR3

; Add to MR3 the value 0x3FC00000 (1.5)
; Store the result in MR3
MADDF32 MR3, #0x3FC0, MR3 ; MR3 = 1.5 + MR3
```

See also

[MADDF32 MRa, MRb, #16FHi](#)
[MADDF32 MRa, MRb, MRc](#)
[MADDF32 MRd, MRc, MRf || MMOV32 MRa, mem32](#)
[MADDF32 MRd, MRc, MRf || MMOV32 mem32, MRa](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRc, MRf](#)

MADDF32 MRa, MRb, #16FHi

Operands

| | |
|--------|---|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

```
LSW: I I I I I I I I I I I I I I
MSW: 0 1 1 1 0 1 1 1 1 1 0 0 b b a a
```

Description

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

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

```
MRa = MRb + #16FHi:0;
```

This instruction can also be written as MADDF32 MRa, #16FHi, MRb.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MADDF32 generates an underflow condition.
- LVF = 1 if MADDF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example 1

```
; X is an array of 32-bit floating-point values
; Find the maximum value in an array X
; and store it in Result
;
_ClalTask1:
    MMOVI16    MAR1, #_X          ; Start address
    MUI16TOF32 MR0, @_len         ; Length of the array
    MNOP                          ; delay for MAR1 load
    MNOP                          ; delay for MAR1 load
    MMOV32     MR1, *MAR1[2]++    ; MR1 = X0
LOOP
    MMOV32     MR2, *MAR1[2]++    ; MR2 = next element
    MMAXF32    MR1, MR2           ; MR1 = MAX(MR1, MR2)
    MADDF32    MR0, MR0, #-1.0    ; Decrement the counter
    MCMPF32    MR0 #0.0           ; Set/clear flags for MBCNDD
    MNOP
    MNOP
    MNOP
    MBCNDD    LOOP, NEQ           ; Branch if not equal to zero
    MMOV32    @_Result, MR1       ; Always executed
    MNOP      ; Always executed
    MNOP      ; Always executed
    MSTOP     ; End of task
```

Example 2

```

; Show the basic operation of MADDF32
;
; Add to MR1 the value 2.0 in 32-bit floating-point format
; Store the result in MR0
    MADDF32 MR0, MR1, #2.0      ; MR0 = MR1 + 2.0

; Add to MR3 the value -2.5 in 32-bit floating-point format
; Store the result in MR2
    MADDF32 MR2, MR3, #-2.5    ; MR2 = MR3 + (-2.5)

; Add to MR0 the value 0x3FC00000 (1.5)
; Store the result in MR0
    MADDF32 MR0, MR0, #0x3FC0  ; MR0 = MR0 + 1.5

```

See also

[MADDF32 MRa, #16FHi, MRb](#)
[MADDF32 MRa, MRb, MRc](#)
[MADDF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)
[MADDF32 MRd, MRe, MRf || MMOV32 mem32, MRa](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRe, MRf](#)

MADDF32 MRa, MRb, MRc 32-bit Floating-Point Addition

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| MRc | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 000 0000 00cc bbaa
MSW: 0111 1100 0010 0000
```

Description

Add the contents of MRc to the contents of MRb and load the result into MRa.

$MRa = MRb + MRc;$

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MADDF32 generates an underflow condition.
- LVF = 1 if MADDF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```
; Given M1, X1 and B1 are 32-bit floating point numbers
; Calculate Y1 = M1*X1+B1
;
_ClalTask1:
    MMOV32 MR0,@M1      ; Load MR0 with M1
    MMOV32 MR1,@X1      ; Load MR1 with X1
    MMPYF32 MR1,MR1,MR0 ; Multiply M1*X1
| | MMOV32 MR0,@B1      ; and in parallel load MR0 with B1
    MADDF32 MR1,MR1,MR0 ; Add M*X1 to B1 and store in MR1
    MMOV32 @Y1,MR1      ; Store the result
    MSTOP               ; end of task
```

See also

[MADDF32 MRa, #16FHi, MRb](#)
[MADDF32 MRa, MRb, #16FHi](#)
[MADDF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)
[MADDF32 MRd, MRe, MRf || MMOV32 mem32, MRa](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRe, MRf](#)

MADDF32 MRd, MRe, MRf||MMOV32 mem32, MRa 32-bit Floating-Point Addition with Parallel Move

Operands

| | |
|-------|--|
| MRd | CLA floating-point destination register for the MADDF32 (MR0 to MR3) |
| MRe | CLA floating-point source register for the MADDF32 (MR0 to MR3) |
| MRf | CLA floating-point source register for the MADDF32 (MR0 to MR3) |
| mem32 | 32-bit memory location accessed using direct or indirect addressing. This will be the destination of the MMOV32. |
| MRa | CLA floating-point source register for the MMOV32 (MR0 to MR3) |

Opcode

```
LSW: mmmm mmmm mmmm mmmm
MSW: 0101 ffee ddaa addr
```

Description

Perform an MADDF32 and a MMOV32 in parallel. Add MRf to the contents of MRe and store the result in MRd. In parallel move the contents of MRa to the 32-bit location mem32.

```
MRd = MRe + MRf;
[mem32] = MRa;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MADDF32 generates an underflow condition.
- LVF = 1 if MADDF32 generates an overflow condition.

Pipeline

Both MADDF32 and MMOV32 complete in a single cycle.

Example

```
; Given A, B and C are 32-bit floating-point numbers
; Calculate Y2 = (A * B)
;           Y3 = (A * B) + C
;
_ClalTask2:
    MMOV32    MR0, @_A        ; Load MR0 with A
    MMOV32    MR1, @_B        ; Load MR1 with B
    MMPYF32   MR1, MR1, MR0    ; Multiply A*B
    | MMOV32   MR0, @_C        ; and in parallel load MR0 with C
    MADDF32   MR1, MR1, MR0    ; Add (A*B) to C
    | MMOV32   @_Y2, MR1       ; and in parallel store A*B
    MMOV32    @_Y3, MR1       ; Store the A*B + C
    MSTOP                                           ; end of task
```

See also

[MADDF32 MRa, #16FHi, MRb](#)
[MADDF32 MRa, MRb, #16FHi](#)
[MADDF32 MRa, MRb, MRc](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRe, MRf](#)
[MADDF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)

MADDF32 MRd, MRe, MRf ||MMOV32 MRa, mem32 32-bit Floating-Point Addition with Parallel Move

Operands

| | |
|-------|---|
| MRd | CLA floating-point destination register for the MADDF32 (MR0 to MR3). MRd cannot be the same register as MRa. |
| MRe | CLA floating-point source register for the MADDF32 (MR0 to MR3) |
| MRf | CLA floating-point source register for the MADDF32 (MR0 to MR3) |
| MRa | CLA floating-point destination register for the MMOV32 (MR0 to MR3). MRa cannot be the same register as MRd. |
| mem32 | 32-bit memory location accessed using direct or indirect addressing. This is the source for the MMOV32. |

Opcode

```
LSW: mmmm mmmm mmmm mmmm
MSW: 0001 ffee ddaa addr
```

Description

Perform an MADDF32 and a MMOV32 operation in parallel. Add MRf to the contents of MRe and store the result in MRd. In parallel move the contents of the 32-bit location mem32 to MRa.

```
MRd = MRe + MRf;
MRa = [mem32];
```

Restrictions

The destination register for the MADDF32 and the MMOV32 must be unique. That is, MRa and MRd cannot be the same register.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MADDF32 generates an underflow condition.
- LVF = 1 if MADDF32 generates an overflow condition.

The MMOV32 Instruction will set the NF and ZF flags as follows:

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

Pipeline

The MADDF32 and the MMOV32 both complete in a single cycle.

Example 1

```
; Given A, B and C are 32-bit floating-point numbers
; Calculate Y1 = A + 4B
;           Y2 = A + C
;
_ClalTask1:
    MMOV32 MR0, @A          ; Load MR0 with A
    MMOV32 MR1, @B          ; Load MR1 with B
    MMPYF32 MR1, MR1, #4.0  ; Multiply 4 * B
    || MMOV32 MR2, @C        ; and in parallel load C
    MADDF32 MR3, MR0, MR1   ; Add A + 4B
    MADDF32 MR3, MR0, MR2   ; Add A + C
    || MMOV32 @Y1, MR3      ; and in parallel store A+4B
    MMOV32 @Y2, MR3         ; store A + C
                                ; end of task
```

Example 2

```

; Given A, B and C are 32-bit floating-point numbers
; Calculate Y3 = (A + B)
;           Y4 = (A + B) * C
;
_ClalTask2:
    MOV32 MR0, @A      ; Load MR0 with A
    MOV32 MR1, @B      ; Load MR1 with B
    MADDF32 MR1, MR1, MR0 ; Add A+B
    || MOV32 MR0, @C    ; and in parallel load MR0 with C
    MPYF32 MR1, MR1, MR0 ; Multiply (A+B) by C
    || MOV32 @Y3, MR1   ; and in parallel store A+B
    MOV32 @Y4, MR1      ; Store the (A+B) * C
    MSTOP              ; end of task

```

See also

[MADDF32 MRa, #16FHi, MRb](#)
[MADDF32 MRa, MRb, #16FHi](#)
[MADDF32 MRa, MRb, MRc](#)
[MADDF32 MRd, MRc, MRf || MOV32 mem32, MRa](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRc, MRf](#)

MAND32 MRa, MRb, MRc *Bitwise AND*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| MRc | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 00cc bbaa
MSW: 0111 1100 0110 0000
```

Description

Bitwise AND of MRb with MRc.

```
MRa(31:0) = MRb(31:0) AND MRc(31:0);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ MR0, #0x5555 ; MR0 = 0x5555AAAA
MMOVXI MR0, #0xAAAA

MMOVIZ MR1, #0x5432 ; MR1 = 0x5432FEDC
MMOVXI MR1, #0xFEDC

; 0101 AND 0101 = 0101 (5)
; 0101 AND 0100 = 0100 (4)
; 0101 AND 0011 = 0001 (1)
; 0101 AND 0010 = 0000 (0)
; 1010 AND 1111 = 1010 (A)
; 1010 AND 1110 = 1010 (A)
; 1010 AND 1101 = 1000 (8)
; 1010 AND 1100 = 1000 (8)

MAND32 MR2, MR1, MR0 ; MR3 = 0x5410AA88
```

See also

[MADD32 MRa, MRb, MRc](#)
[MASR32 MRa, #SHIFT](#)
[MLSL32 MRa, #SHIFT](#)
[MLSR32 MRa, #SHIFT](#)
[MOR32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)
[MSUB32 MRa, MRb, MRc](#)

MASR32 MRa, #SHIFT *Arithmetic Shift Right*

Operands

| | |
|--------|---|
| MRa | CLA floating-point source/destination register (MR0 to MR3) |
| #SHIFT | Number of bits to shift (1 to 32) |

Opcode

```
LSW: 0000 0000 0shi ftaa
MSW: 0111 1011 0100 0000
```

Description

Arithmetic shift right of MRa by the number of bits indicated. The number of bits can be 1 to 32.

```
MARa(31:0) = Arithmetic Shift(MARa(31:0) by #SHIFT bits);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
; Given m2 = (int32)32
;         x2 = (int32)64
;         b2 = (int32)-128
;
; Calculate
;         m2 = m2/2
;         x2 = x2/4
;         b2 = b2/8
;
_ClalTask2:
    MMOV32 MR0, @_m2 ; MR0 = 32 (0x00000020)
    MMOV32 MR1, @_x2 ; MR1 = 64 (0x00000040)
    MMOV32 MR2, @_b2 ; MR2 = -128 (0xFFFFFFFF80)
    MASR32 MR0, #1 ; MR0 = 16 (0x00000010)
    MASR32 MR1, #2 ; MR1 = 16 (0x00000010)
    MASR32 MR2, #3 ; MR2 = -16 (0xFFFFFFFFF0)
    MMOV32 @_m2, MR0 ; store results
    MMOV32 @_x2, MR1
    MMOV32 @_b2, MR2
    MSTOP ; end of task
```

See also

[MADD32 MRa, MRb, MRc](#)
[MAND32 MRa, MRb, MRc](#)
[MLSL32 MRa, #SHIFT](#)
[MLSR32 MRa, #SHIFT](#)
[MOR32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)
[MSUB32 MRa, MRb, MRc](#)

MBCNDD 16BitDest {, CNDF} *Branch Conditional Delayed*

Operands

| | |
|-----------|---|
| 16BitDest | 16-bit destination if condition is true |
| CNDF | Optional condition tested |

Opcode

```
LSW: dest dest dest dest
MSW: 0111 1001 1000 cndf
```

Description

If the specified condition is true, then branch by adding the signed 16BitDest value to the MPC value. Otherwise, continue without branching. If the address overflows, it wraps around. Therefore a value of "0xFFFFE" will put the MPC back to the MBCNDD instruction. Since the MPC is only 12-bits, unused bits the upper 4 bits of the destination address are ignored.

Please refer to the pipeline section for important information regarding this instruction.

```
if (CNDF == TRUE) MPC += 16BitDest;
```

CNDF is one of the following conditions:

| Encode ⁽¹⁾ | CNDF | Description | MSTF 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 OR 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 and NF flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Restrictions

The MBCNDD instruction is not allowed three instructions before or after a MBCNDD, MCCNDD or MRCNDD instruction. Refer to the pipeline section for more information.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

The MBCNDD instruction by itself is a single-cycle instruction. As shown in [Table 24](#) for each branch 6 instruction slots are executed; three before the branch instruction (I2-I4) and three after the branch instruction (I5-I7). The total number of cycles for a branch taken or not taken depends on the usage of these slots. That is, the number of cycles depends on how many slots are filled with a MNOP as well as which slots are filled. The effective number of cycles for a branch can, therefore, range from 1 to 7 cycles. The number of cycles for a branch taken may not be the same as for a branch not taken.

Referring to [Table 24](#) and [Table 25](#), the instructions before and after MBCNDD have the following properties:

- **I1**
 - I1 is the last instruction that can effect the CNDF flags for the MBCNDD instruction. The CNDF flags are tested in the D2 phase of the pipeline. That is, a decision is made whether to branch or not when MBCNDD is in the D2 phase.
 - There are no restrictions on the type of instruction for I1.
- **I2, I3 and I4**
 - The three instructions proceeding MBCNDD can change MSTF flags but will have no effect on whether the MBCNDD instruction branches or not. This is because the flag modification will occur after the D2 phase of the MBCNDD instruction.
 - These instructions must not be the following: MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.
- **I5, I6 and I7**
 - The three instructions following MBCNDD are always executed irrespective of whether the branch is taken or not.
 - These instructions must not be the following: MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.

```

<Instruction 1> ; I1 Last instruction that can affect flags for
                ; the MBCNDD operation
<Instruction 2> ; I2 Cannot be stop, branch, call or return
<Instruction 3> ; I3 Cannot be stop, branch, call or return
<Instruction 4> ; I4 Cannot be stop, branch, call or return
MBCNDD _Skip, NEQ ; Branch to Skip if not equal to zero
                ; Three instructions after MBCNDD are always
                ; executed whether the branch is taken or not
<Instruction 5> ; I5 Cannot be stop, branch, call or return
<Instruction 6> ; I6 Cannot be stop, branch, call or return
<Instruction 7> ; I7 Cannot be stop, branch, call or return
<Instruction 8> ; I8
<Instruction 9> ; I9
....
_Skip:
<Destination 1> ; d1 Can be any instruction
<Destination 2> ; d2
<Destination 3> ; d3
....
....
MSTOP
....

```

Table 24. Pipeline Activity For MBCNDD, Branch Not Taken

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------|--------|--------|--------|--------|-----|-----|-----|---|
| I1 | I1 | | | | | | | |
| I2 | I2 | I1 | | | | | | |
| I3 | I3 | I2 | I1 | | | | | |
| I4 | I4 | I3 | I2 | I1 | | | | |
| MBCNDD | MBCNDD | I4 | I3 | I2 | I1 | | | |
| I5 | I5 | MBCNDD | I4 | I3 | I2 | I1 | | |
| I6 | I6 | I5 | MBCNDD | I4 | I3 | I2 | I1 | |
| I7 | I7 | I6 | I5 | MBCNDD | I4 | I3 | I2 | |
| I8 | I8 | I7 | I6 | I5 | - | I4 | I3 | |
| I9 | I9 | I8 | I7 | I6 | I5 | - | I4 | |
| I10 | I10 | I9 | I8 | I7 | I6 | I5 | - | |
| | | I10 | I9 | I8 | I7 | I6 | I5 | |
| | | | I10 | I9 | I8 | I7 | I6 | |
| | | | | I10 | I9 | I8 | I7 | |
| | | | | | I10 | I9 | I8 | |
| | | | | | | I10 | I9 | |
| | | | | | | | I10 | |

Table 25. Pipeline Activity For MBCNDD, Branch Taken

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------|--------|--------|--------|--------|----|----|----|---|
| I1 | I1 | | | | | | | |
| I2 | I2 | I1 | | | | | | |
| I3 | I3 | I2 | I1 | | | | | |
| I4 | I4 | I3 | I2 | I1 | | | | |
| MBCNDD | MBCNDD | I4 | I3 | I2 | I1 | | | |
| I5 | I5 | MBCNDD | I4 | I3 | I2 | I1 | | |
| I6 | I6 | I5 | MBCNDD | I4 | I3 | I2 | I1 | |
| I7 | I7 | I6 | I5 | MBCNDD | I4 | I3 | I2 | |
| d1 | d1 | I7 | I6 | I5 | - | I4 | I3 | |
| d2 | d2 | d1 | I7 | I6 | I5 | - | I4 | |
| d3 | d3 | d2 | d1 | I7 | I6 | I5 | - | |
| | | d3 | d2 | d1 | I7 | I6 | I5 | |
| | | | d3 | d2 | d1 | I7 | I6 | |
| | | | | d3 | d2 | d1 | I7 | |
| | | | | | d3 | d2 | d1 | |
| | | | | | | d3 | d2 | |
| | | | | | | | d3 | |

Example 1

```

; if (State == 0.1)
; RampState = RampState || RAMPMASK
; else if (State == 0.01)
; CoastState = CoastState || COASTMASK
; else
; SteadyState = SteadyState || STEADYMASK
;
_ClalTask1:
MMOV32 MR0, @State
MCMPF32 MR0, #0.1           ; Affects flags for 1st MBCNDD (A)
MNOP
MNOP
MNOP
MBCNDD Skip1, NEQ           ; (A) If State != 0.1, go to Skip1
MNOP ; Always executed
MNOP ; Always executed
MNOP ; Always executed
MMOV32 MR1, @RampState      ; Execute if (A) branch not taken
MMOVXI MR2, #RAMPMASK       ; Execute if (A) branch not taken
MOR32 MR1, MR2              ; Execute if (A) branch not taken
MMOV32 @RampState, MR1      ; Execute if (A) branch not taken
MSTOP                      ; end of task if (A) branch not taken
Skip1:
MCMPF32 MR0, #0.01          ; Affects flags for 2nd MBCNDD (B)
MNOP
MNOP
MNOP
MBCNDD Skip2, NEQ           ; (B) If State != 0.01, go to Skip2
MNOP ; Always executed
MNOP ; Always executed
MNOP ; Always executed
MMOV32 MR1, @CoastState     ; Execute if (B) branch not taken
MMOVXI MR2, #COASTMASK      ; Execute if (B) branch not taken
MOR32 MR1, MR2              ; Execute if (B) branch not taken
MMOV32 @CoastState, MR1     ; Execute if (B) branch not taken
MSTOP
Skip2:
MMOV32 MR3, @SteadyState    ; Executed if (B) branch taken
MMOVXI MR2, #STEADYMASK     ; Executed if (B) branch taken
MOR32 MR3, MR2              ; Executed if (B) branch taken
MMOV32 @SteadyState, MR3    ; Executed if (B) branch taken
MSTOP

```

Example 2

```

; This example is the same as Example 1, except
; the code is optimized to take advantage of delay slots
;
; if (State == 0.1)
; RampState = RampState || RAMPMASK
; else if (State == 0.01)
; CoastState = CoastState || COASTMASK
; else
; SteadyState = SteadyState || STEADYMASK
;
_ClalTask2:
MMOV32 MR0, @State
MCMPF32 MR0, #0.1           ; Affects flags for 1st MBCNDD (A)
MCMPF32 MR0, #0.01          ; Check used by 2nd MBCNDD (B)
MTESTTF EQ                  ; Store EQ flag in TF for 2nd MBCNDD (B)
MNOP
MBCNDD Skip1, NEQ           ; (A) If State != 0.1, go to Skip1
MMOV32 MR1, @RampState      ; Always executed
MMOVXI MR2, #RAMPMASK       ; Always executed
MOR32 MR1, MR2              ; Always executed
MMOV32 @RampState, MR1      ; Execute if (A) branch not taken
MSTOP                      ; end of task if (A) branch not taken

Skip1:
MMOV32 MR3, @SteadyState
MMOVXI MR2, #STEADYMASK
MOR32 MR3, MR2
MBCNDD Skip2, NTF           ; (B) if State != .01, go to Skip2
MMOV32 MR1, @CoastState     ; Always executed
MMOVXI MR2, #COASTMASK      ; Always executed
MOR32 MR1, MR2              ; Always executed
MMOV32 @CoastState, MR1     ; Execute if (B) branch not taken
MSTOP                      ; end of task if (B) branch not taken

Skip2:
MMOV32 @SteadyState, MR3    ; Executed if (B) branch taken
MSTOP

```

See also

[MCCNDD 16BitDest, CNDF](#)
[MRCNDD CNDF](#)

MCCNDD 16BitDest {, CNDF} *Call Conditional Delayed*

Operands

| | |
|-----------|---|
| 16BitDest | 16-bit destination if condition is true |
| CNDF | Optional condition to be tested |

Opcode

```
LSW: dest dest dest dest
MSW: 0111 1001 1001 cndf
```

Description

If the specified condition is true, then store the return address in the RPC field of MSTF and make the call by adding the signed 16BitDest value to the MPC value. Otherwise, continue code execution without making the call. If the address overflows, it wraps around. Therefore a value of "0xFFFF" will put the MPC back to the MCCNDD instruction. Since the MPC is only 12 bits, unused bits the upper 4 bits of the destination address are ignored.

Please refer to the pipeline section for important information regarding this instruction.

```
if (CNDF == TRUE)
{
    RPC = return address;
    MPC += 16BitDest;
};
```

CNDF is one of the following conditions:

| Encode ⁽³⁾ | CNDF | Description | MSTF 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 OR 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 ⁽⁴⁾ | Unconditional with flag modification | None |

⁽³⁾ Values not shown are reserved.

⁽⁴⁾ This is the default operation if no CNDF field is specified. This condition will allow the ZF and NF flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Restrictions

The MCCNDD instruction is not allowed three instructions before or after a MBCNDD, MCCNDD, or MRCNDD instruction. Refer to the Pipeline section for more details.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

The MCCNDD instruction by itself is a single-cycle instruction. As shown in [Table 26](#), for each call 6 instruction slots are executed; three before the call instruction (I2-I4) and three after the call instruction (I5-I7). The total number of cycles for a call taken or not taken depends on the usage of these slots. That is, the number of cycles depends on how many slots are filled with a MNOP as well as which slots are filled. The effective number of cycles for a call can, therefore, range from 1 to 7 cycles. The number of cycles for a call taken may not be the same as for a call not taken.

Referring to the following code fragment and the pipeline diagrams in [Table 26](#) and [Table 27](#), the instructions before and after MCCNDD have the following properties:

- **I1**
 - I1 is the last instruction that can effect the CNDF flags for the MCCNDD instruction. The CNDF flags are tested in the D2 phase of the pipeline. That is, a decision is made whether to branch or not when MCCNDD is in the D2 phase.
 - There are no restrictions on the type of instruction for I1.
- **I2, I3 and I4**
 - The three instructions proceeding MCCNDD can change MSTF flags but will have no effect on whether the MCCNDD instruction makes the call or not. This is because the flag modification will occur after the D2 phase of the MCCNDD instruction.
 - These instructions must not be the following: MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.
- **I5, I6 and I7**
 - The three instructions following MBCNDD are always executed irrespective of whether the branch is taken or not.
 - These instructions must not be the following: MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.

```

<Instruction 1> ; I1 Last instruction that can affect flags for
                ; the MCCNDD operation
<Instruction 2> ; I2 Cannot be stop, branch, call or return
<Instruction 3> ; I3 Cannot be stop, branch, call or return
<Instruction 4> ; I4 Cannot be stop, branch, call or return

MCCNDD _func, NEQ ; Call to func if not equal to zero

                ; Three instructions after MCCNDD are always
                ; executed whether the call is taken or not

<Instruction 5> ; I5 Cannot be stop, branch, call or return
<Instruction 6> ; I6 Cannot be stop, branch, call or return
<Instruction 7> ; I7 Cannot be stop, branch, call or return
<Instruction 8> ; I8 The address of this instruction is saved
                ; in the RPC field of the MSTF register.
                ; Upon return this value is loaded into MPC
                ; and fetching continues from this point.
<Instruction 9> ; I9
....
_func:
<Destination 1> ; d1 Can be any instruction
<Destination 2> ; d2
<Destination 3> ; d3
<Destination 4> ; d4 Last instruction that can affect flags for
                ; the MRCNDD operation

<Destination 5> ; d5 Cannot be stop, branch, call or return
<Destination 6> ; d6 Cannot be stop, branch, call or return
<Destination 7> ; d7 Cannot be stop, branch, call or return

MRCNDD, UNC     ; Return to <Instruction 8>, unconditional

                ; Three instructions after MRCNDD are always
                ; executed whether the return is taken or not

<Destination 8> ; d8 Cannot be stop, branch, call or return
<Destination 9> ; d9 Cannot be stop, branch, call or return
<Destination 10> ; d10 Cannot be stop, branch, call or return
<Destination 11> ; d11
....
MSTOP

```

Table 26. Pipeline Activity For MCCNDD, Call Not Taken

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------|--------|--------|--------|--------|-----|-----|-----|---|
| I1 | I1 | | | | | | | |
| I2 | I2 | I1 | | | | | | |
| I3 | I3 | I2 | I1 | | | | | |
| I4 | I4 | I3 | I2 | I1 | | | | |
| MCCNDD | MCCNDD | I4 | I3 | I2 | I1 | | | |
| I5 | I5 | MCCNDD | I4 | I3 | I2 | I1 | | |
| I6 | I6 | I5 | MCCNDD | I4 | I3 | I2 | I1 | |
| I7 | I7 | I6 | I5 | MCCNDD | I4 | I3 | I2 | |
| I8 | I8 | I7 | I6 | I5 | - | I4 | I3 | |
| I9 | I9 | I8 | I7 | I6 | I5 | - | I4 | |
| I10 | I10 | I9 | I8 | I7 | I6 | I5 | - | |
| etc | | I10 | I9 | I8 | I7 | I6 | I5 | |
| | | | I10 | I9 | I8 | I7 | I6 | |
| | | | | I10 | I9 | I8 | I7 | |
| | | | | | I10 | I9 | I8 | |
| | | | | | | I10 | I9 | |
| | | | | | | | I10 | |

Table 27. Pipeline Activity For MCCNDD, Call Taken

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------------|--------|--------|--------|--------|----|----|----|---|
| I1 | I1 | | | | | | | |
| I2 | I2 | I1 | | | | | | |
| I3 | I3 | I2 | I1 | | | | | |
| I4 | I4 | I3 | I2 | I1 | | | | |
| MCCNDD | MCCNDD | I4 | I3 | I2 | I1 | | | |
| I5 | I5 | MCCNDD | I4 | I3 | I2 | I1 | | |
| I6 | I6 | I5 | MCCNDD | I4 | I3 | I2 | I1 | |
| I7 ⁽¹⁾ | I7 | I6 | I5 | MCCNDD | I4 | I3 | I2 | |
| d1 | d1 | I7 | I6 | I5 | - | I4 | I3 | |
| d2 | d2 | d1 | I7 | I6 | I5 | - | I4 | |
| d3 | d3 | d2 | d1 | I7 | I6 | I5 | - | |
| etc | | d3 | d2 | d1 | I7 | I6 | I5 | |
| | | | d3 | d2 | d1 | I7 | I6 | |
| | | | | d3 | d2 | d1 | I7 | |
| | | | | | d3 | d2 | d1 | |
| | | | | | | d3 | d2 | |
| | | | | | | | d3 | |

⁽¹⁾ The RPC value in the MSTF register will point to the instruction following I7 (instruction I8).

Example

;

See also

[MBCNDD #16BitDest, CNDF](#)
[MMOV32 mem32, MSTF](#)
[MMOV32 MSTF, mem32](#)
[MRCNDD CNDF](#)

MCMP32 MRa, MRb 32-bit Integer Compare for Equal, Less Than or Greater Than

Operands

| | |
|-----|---|
| MRa | CLA floating-point source register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1111 0010 0000
```

Description

Set ZF and NF flags on the result of MRa - MRb where MRa and MRb are 32-bit integers. For a floating point compare refer to [MCMFP32](#).

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
If(MRa == MRb) {ZF=1; NF=0;}
If(MRa > MRb) {ZF=0; NF=0;}
If(MRa < MRb) {ZF=0; NF=1;}
```

Pipeline

This is a single-cycle instruction.

Example

```
; Behavior of ZF and NF flags for different comparisons
;
; Given A = (int32)1
;       B = (int32)2
;       C = (int32)-7
;
MMOV32 MR0, @_A ; MR0 = 1 (0x00000001)
MMOV32 MR1, @_B ; MR1 = 2 (0x00000002)
MMOV32 MR2, @_C ; MR2 = -7 (0xFFFFFFFF9)
MCMP32 MR2, MR2 ; NF = 0, ZF = 1
MCMP32 MR0, MR1 ; NF = 1, ZF = 0
MCMP32 MR1, MR0 ; NF = 0, ZF = 0
```

See also

[MADD32 MRa, MRb, MRc](#)
[MSUB32 MRa, MRb, MRc](#)

MCMPF32 MRa, MRb 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

| | |
|-----|---|
| MRa | CLA floating-point source register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 0000 0000
```

Description

Set ZF and NF flags on the result of MRa - MRb. The MCMPF32 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified as follows:

```
If(MRa == MRb) {ZF=1; NF=0;}
If(MRa > MRb) {ZF=0; NF=0;}
If(MRa < MRb) {ZF=0; NF=1;}
```

Pipeline

This is a single-cycle instruction.

Example

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

MMOVIZ    MR1, #-2.0 ; MR1 = -2.0 (0xC0000000)
MMOVIZ    MR0, #5.0  ; MR0 = 5.0 (0x40A00000)
MCMPF32   MR1, MR0   ; ZF = 0, NF = 1
MCMPF32   MR0, MR1   ; ZF = 0, NF = 0
MCMPF32   MR0, MR0   ; ZF = 1, NF = 0
```

See also

[MCMPF32 MRa, #16FHi](#)
[MMAXF32 MRa, #16FHi](#)
[MMAXF32 MRa, MRb](#)
[MMINF32 MRa, #16FHi](#)
[MMINF32 MRa, MRb](#)

MCMPF32 MRa, #16FHi 32-bit Floating-Point Compare for Equal, Less Than or Greater Than

Operands

| | |
|--------|---|
| MRa | CLA floating-point source register (MR0 to MR3) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

```
LSW: I I I I I I I I I I I I I I I I
MSW: 0 1 1 1 1 0 0 0 1 1 0 0 0 0 a a
```

Description

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

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

The MCMPF32 instruction is performed as a logical compare operation. This is possible because of the IEEE floating-point format offsets the exponent. Basically the bigger the binary number, the bigger the floating-point value.

Special cases for inputs:

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

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified as follows:

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

Pipeline

This is a single-cycle instruction

Example 1

; Behavior of ZF and NF flags for different comparisons

```
MMOVIZ MR1, #-2.0 ; MR1 = -2.0 (0xC0000000)
MMOVIZ MR0, #5.0 ; MR0 = 5.0 (0x40A00000)
MCMPF32 MR1, #-2.2 ; ZF = 0, NF = 0
MCMPF32 MR0, #6.5 ; ZF = 0, NF = 1
MCMPF32 MR0, #5.0 ; ZF = 1, NF = 0
```

Example 2

```

; X is an array of 32-bit floating-point values
; and has len elements. Find the maximum value in
; the array and store it in Result
;
; Note: MCMFPF32 and MSWAPF can be replaced with MMAXF32
;
_ClalTask1:
    MMOV16 MAR1, #_X          ; Start address
    MUI16TOF32 MR0, @_len     ; Length of the array
    MNOP                     ; delay for MAR1 load
    MNOP                     ; delay for MAR1 load
    MMOV32 MR1, *MAR1[2]++    ; MR1 = X0

    LOOP
        MMOV32 MR2, *MAR1[2]++ ; MR2 = next element
        MCMFPF32 MR2, MR1      ; Compare MR2 with MR1
        MSWAPF MR1, MR2, GT    ; MR1 = MAX(MR1, MR2)
        MADD32 MR0, MR0, #-1.0 ; Decrement the counter
        MCMFPF32 MR0 #0.0      ; Set/clear flags for MBCNDD
        MNOP
        MNOP
        MNOP
        MBCNDD LOOP, NEQ       ; Branch if not equal to zero
        MMOV32 @_Result, MR1   ; Always executed
        MNOP                   ; Always executed
        MNOP                   ; Always executed
        MSTOP                  ; End of task

```

See also

[MCMFPF32 MRa, MRb](#)
[MMAXF32 MRa, #16FHi](#)
[MMAXF32 MRa, MRb](#)
[MMINF32 MRa, #16FHi](#)
[MMINF32 MRa, MRb](#)

MDEBUGSTOP

Debug Stop Task

Operands

| | |
|------|---|
| none | This instruction does not have any operands |
|------|---|

Opcode

```
LSW: 0000 0000 0000 0000
MSW: 0111 1111 0110 0000
```

Description

When CLA breakpoints are enabled, the MDEBUGSTOP instruction is used to halt a task so that it can be debugged. That is, MDEBUGSTOP is the CLA breakpoint. If CLA breakpoints are not enabled, the MDEBUGSTOP instruction behaves like a MNOP. Unlike the MSTOP, the MIRUN flag is not cleared and an interrupt is not issued. A single-step or run operation will continue execution of the task.

Restrictions

The MDEBUGSTOP instruction cannot be placed 3 instructions before or after a [MBCNDD](#), [MCCNDD](#) or [MRCNDD](#) instruction.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
;
```

See also

[MSTOP](#)

MEALLOW *Enable CLA Write Access to EALLOW Protected Registers*

Operands

| | |
|------|---|
| none | This instruction does not have any operands |
|------|---|

Opcode

```
LSW: 0000 0000 0000 0000
MSW: 0111 1111 1001 0000
```

Description

This instruction sets the MEALLOW bit in the CLA status register MSTF. When this bit is set, the CLA is allowed write access to EALLOW protected registers. To again protect against CLA writes to protected registers, use the MEDIS instruction.

MEALLOW and MEDIS only control CLA write access; reads are allowed even if MEALLOW has not been executed. MEALLOW and MEDIS are also independant from the main CPU's EALLOW/EDIS. This instruction does not modify the EALLOW bit in the main CPU's status register. The MEALLOW bit in MSTF only controls access for the CLA while the EALLOW bit in the ST1 register only controls access for the main CPU.

As with EALLOW, the MEALLOW bit is overridden via the JTAG port, allowing full control of register accesses during debug from Code Composer Studio.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; C header file including definition of
; the EPwm1Regs structure
;
; The ePWM TZSEL register is EALLOW protected
;
.cdecls C,LIST,"CLAShared.h"
...
_ClalTask1:
...
MEALLOW                ; Allow CLA write access
MMOV16 @_EPwm1Regs.TZSEL.all, MR3 ; Write to TZSEL
MEDIS                  ; Disallow CLA write access
...
...
MSTOP
```

See also

[MEDIS](#)

MEDIS

Disable CLA Write Access to EALLOW Protected Registers

Operands

| | |
|------|---|
| none | This instruction does not have any operands |
|------|---|

Opcode

```
LSW: 0000 0000 0000 0000
MSW: 0111 1111 1011 0000
```

Description

This instruction clears the MEALLOW bit in the CLA status register MSTF. When this bit is clear, the CLA is not allowed write access to EALLOW protected registers. To enable CLA writes to protected registers, use the MEALLOW instruction.

MEALLOW and MEDIS only control CLA write access; reads are allowed even if MEALLOW has not been executed. MEALLOW and MEDIS are also independent from the main CPU's EALLOW/EDIS. This instruction does not modify the EALLOW bit in the main CPU's status register. The MEALLOW bit in MSTF only controls access for the CLA while the EALLOW bit in the ST1 register only controls access for the main CPU.

As with EALLOW, the MEALLOW bit is overridden via the JTAG port, allowing full control of register accesses during debug from Code Composer Studio.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; C header file including definition of
; the EPwm1Regs structure
;
; The ePWM TZSEL register is EALLOW protected
;
.cdecls C,LIST,"CLAShared.h"
...
_ClalTask1:
...
MEALLOW                ; Allow CLA write access
MMOV16 @_EPwm1Regs.TZSEL.all, MR3 ; Write to TZSEL
MEDIS                  ; Disallow CLA write access
...
...
MSTOP
```

See also

[MEALLOW](#)

MEINVF32 MRa, MRb 32-bit Floating-Point Reciprocal Approximation

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1111 0000 0000
```

Description

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

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

After 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 MEINVF32 operation will not generate a negative zero, DeNorm or NaN value.

```
MRa = Estimate of 1/MRb;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MEINVF32 generates an underflow condition.
- LVF = 1 if MEINVF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```

; Calculate Num/Den using a Newton-Raphson algorithm for 1/Den
; Ye = Estimate(1/X)
; Ye = Ye*(2.0 - Ye*X)
; Ye = Ye*(2.0 - Ye*X)
;
_ClalTask1:
    MMOV32 MR1, @_Den      ; MR1 = Den
    MEINVF32 MR2, MR1      ; MR2 = Ye = Estimate(1/Den)
    MMPYF32 MR3, MR2, MR1  ; MR3 = Ye*Den
    MSUBF32 MR3, #2.0, MR3 ; MR3 = 2.0 - Ye*Den
    MMPYF32 MR2, MR2, MR3  ; MR2 = Ye = Ye*(2.0 - Ye*Den)
    MMPYF32 MR3, MR2, MR1  ; MR3 = Ye*Den
    || MMOV32 MR0, @_Num    ; MR0 = Num
    MSUBF32 MR3, #2.0, MR3 ; MR3 = 2.0 - Ye*Den
    MMPYF32 MR2, MR2, MR3  ; MR2 = Ye = Ye*(2.0 - Ye*Den)
    || MMOV32 MR1, @_Den    ; Reload Den To Set Sign
    MNEGF32 MR0, MR0, EQ   ; if(Den == 0.0) Change Sign Of Num
    MMPYF32 MR0, MR2, MR0  ; MR0 = Y = Ye*Num
    MMOV32 @_Dest, MR0     ; Store result
    MSTOP                  ; end of task

```

See also
[MEISQRTF32 MRa, MRb](#)

MEISQRTF32 MRa, MRb 32-bit Floating-Point Square-Root Reciprocal Approximation

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 0100 0000
```

Description

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

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

After 2 iterations of the Newton-Raphson algorithm, you will get an exact answer accurate to the 32-bit floating-point format. On each iteration the mantissa bit accuracy approximately doubles. The MEISQRTF32 operation will not generate a negative zero, DeNorm or NaN value.

```
MRa = Estimate of 1/sqrt (MRb);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MEISQRTF32 generates an underflow condition.
- LVF = 1 if MEISQRTF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```

; 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
;
_ClalTask3:
    MMOV32 MR0, @_x          ; MR0 = X
    MEISQRTF32 MR1, MR0      ; MR1 = Ye = Estimate(1/sqrt(X))
    MMOV32 MR1, @_x, EQ      ; if(X == 0.0) Ye = 0.0
    MMPYF32 MR3, MR0, #0.5   ; MR3 = X*0.5
    MMPYF32 MR2, MR1, MR3    ; MR2 = Ye*X*0.5
    MMPYF32 MR2, MR1, MR2    ; MR2 = Ye*Ye*X*0.5
    MSUBF32 MR2, #1.5, MR2   ; MR2 = 1.5 - Ye*Ye*X*0.5
    MMPYF32 MR1, MR1, MR2    ; MR1 = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    MMPYF32 MR2, MR1, MR3    ; MR2 = Ye*X*0.5
    MMPYF32 MR2, MR1, MR2    ; MR2 = Ye*Ye*X*0.5
    MSUBF32 MR2, #1.5, MR2   ; MR2 = 1.5 - Ye*Ye*X*0.5
    MMPYF32 MR1, MR1, MR2    ; MR1 = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    MMPYF32 MR0, MR1, MR0    ; MR0 = Y = Ye*X
    MMOV32 @_y, MR0          ; Store Y = sqrt(X)
    MSTOP                    ; end of task

```

See also
[MEINVF32 MRa, MRb](#)

MF32TOI16 MRa, MRb *Convert 32-bit Floating-Point Value to 16-bit Integer*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 1110 0000
```

Description

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

```
MRa(15:0) = F32TOI16(MRb);
MRa(31:16) = sign extension of MRa(15);
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ      MR0, #5.0    ; MR0      = 5.0 (0x40A00000)
MF32TOI16    MR1, MR0    ; MR1(15:0) = MF32TOI16(MR0) = 0x0005
              ; MR1(31:16) = Sign extension of MR1(15) = 0x0000
MMOVIZ      MR2, #-5.0   ; MR2      = -5.0 (0xC0A00000)
MF32TOI16    MR3, MR2    ; MR3(15:0) = MF32TOI16(MR2) = -5 (0xFFFFB)
              ; MR3(31:16) = Sign extension of MR3(15) = 0xFFFF
```

See also

[MF32TOI16R MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MF32TOI16R MRa, MRb *Convert 32-bit Floating-Point Value to 16-bit Integer and Round*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 0110 0000
```

Description

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

```
MRa(15:0) = F32TOI16round(MRb);
MRa(31:16) = sign extension of MRa(15);
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ MR0, #0x3FD9 ; MR0(31:16) = 0x3FD9
MMOVXI MR0, #0x999A ; MR0(15:0) = 0x999A
                    ; MR0 = 1.7 (0x3FD9999A)
MF32TOI16R MR1, MR0 ; MR1(15:0) = MF32TOI16round (MR0) = 2 (0x0002)
                    ; MR1(31:16) = Sign extension of MR1(15) = 0x0000
MMOVF32 MR2, #-1.7 ; MR2 = -1.7 (0xBF99999A)
MF32TOI16R MR3, MR2 ; MR3(15:0) = MF32TOI16round (MR2) = -2 (0xFFFE)
                    ; MR3(31:16) = Sign extension of MR2(15) = 0xFFFF
```

See also

[MF32TOI16 MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MF32TOI32 MRa, MRb *Convert 32-bit Floating-Point Value to 32-bit Integer*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 0110 0000
```

Description

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

```
MRa = F32TOI32(MRb);
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example 1

```
MMOVF32    MR2, #11204005.0    ; MR2 = 11204005.0 (0x4B2AF5A5)
MF32TOI32  MR3, MR2             ; MR3 = MF32TOI32(MR2) = 11204005 (0x00AAF5A5)
MMOVF32    MR0, #-11204005.0    ; MR0 = -11204005.0 (0xCB2AF5A5)
MF32TOI32  MR1, MR0             ; MR1 = MF32TOI32(MR0) = -11204005 (0xFF550A5B)
```

Example 2

```
; Given X, M and B are IQ24 numbers:
; X = IQ24(+2.5) = 0x02800000
; M = IQ24(+1.5) = 0x01800000
; B = IQ24(-0.5) = 0xFF800000
;
; Calculate Y = X * M + B
;
; Convert M, X and B from IQ24 to float
;
_ClalTask2:
    MI32TOF32 MR0, @_M           ; MR0 = 0x4BC00000
    MI32TOF32 MR1, @_X           ; MR1 = 0x4C200000
    MI32TOF32 MR2, @_B           ; MR2 = 0xCB000000
    MPPYF32   MR0, MR0, #0x3380 ; M = 1/(1*2^24) * iqm = 1.5 (0x3FC00000)
    MPPYF32   MR1, MR1, #0x3380 ; X = 1/(1*2^24) * iqx = 2.5 (0x40200000)
    MPPYF32   MR2, MR2, #0x3380 ; B = 1/(1*2^24) * iqb = -0.5 (0xBF000000)
    MPPYF32   MR3, MR0, MR1      ; M*X
    MADDF32   MR2, MR2, MR3      ; Y=MX+B = 3.25 (0x40500000)

; Convert Y from float32 to IQ24
    MPPYF32   MR2, MR2, #0x4B80 ; Y * 1*2^24
    MF32TOI32 MR2, MR2           ; IQ24(Y) = 0x03400000
    MMOV32    @_Y, MR2           ; store result
    MSTOP                                           ; end of task
```

See also

[MF32TOUI32 MRa, MRb](#)
[MI32TOF32 MRa, MRb](#)
[MI32TOF32 MRa, mem32](#)
[MUI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, mem32](#)

MF32TOUI16 MRa, MRb *Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 1010 0000
```

Description

Convert the 32-bit floating point value in MRb to an unsigned 16-bit integer value and truncate to zero. The result will be stored in MRa. To instead round the integer to the nearest even value use the MF32TOUI16R instruction.

```
MRa(15:0) = F32TOUI16(MRb);
MRa(31:16) = 0x0000;
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ      MR0, #9.0      ; MR0 = 9.0 (0x41100000)
MF32TOUI16   MR1, MR0      ; MR1(15:0) = MF32TOUI16(MR0) = 9 (0x0009)
                          ; MR1(31:16) = 0x0000
MMOVIZ      MR2, #-9.0     ; MR2 = -9.0 (0xC1100000)
MF32TOUI16   MR3, MR2      ; MR3(15:0) = MF32TOUI16(MR2) = 0 (0x0000)
                          ; MR3(31:16) = 0x0000
```

See also

[MF32TOI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MF32TOUI16R MRa, MRb Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer and Round
Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 1100 0000
```

Description

Convert the 32-bit floating-point value in MRb to an unsigned 16-bit integer and round to the closest even value. The result will be stored in MRa. To instead truncate the converted value, use the MF32TOUI16 instruction.

```
MRa(15:0) = MF32TOUI16round(MRb);
MRa(31:16) = 0x0000;
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ      MR0, #0x412C    ; MR0 = 0x412C
MMOVXI      MR0, #0xCCCD    ; MR0 = 0xCCCD ; MR0 = 10.8 (0x412CCCCD)
MF32TOUI16R MR1, MR0        ; MR1(15:0) = MF32TOUI16round(MR0) = 11 (0x000B)
                                   ; MR1(31:16) = 0x0000
MMOVF32     MR2, #-10.8     ; MR2 = -10.8 (0x0xC12CCCCD)
MF32TOUI16R MR3, MR2        ; MR3(15:0) = MF32TOUI16round(MR2) = 0 (0x0000)
                                   ; MR3(31:16) = 0x0000
```

See also

[MF32TOI16 MRa, MRb](#)
[MF32TOI16R MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MF32TOUI32 MRa, MRb Convert 32-bit Floating-Point Value to 16-bit Unsigned Integer

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 1010 0000
```

Description

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

```
MRa = F32TOUI32(MRb);
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ      MR0, #12.5    ; MR0 = 12.5 (0x41480000)
MF32TOUI32   MR0, MR0      ; MR0 = MF32TOUI32 (MR0) = 12 (0x0000000C)
MMOVIZ      MR1, #-6.5     ; MR1 = -6.5 (0xC0D00000)
MF32TOUI32   MR2, MR1      ; MR2 = MF32TOUI32 (MR1) = 0.0 (0x00000000)
```

See also

[MF32TOI32 MRa, MRb](#)
[MI32TOF32 MRa, MRb](#)
[MI32TOF32 MRa, mem32](#)
[MUI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, mem32](#)

MFRACF32 MRa, MRb *Fractional Portion of a 32-bit Floating-Point Value*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 0000 0000
```

Description

Returns in MRa the fractional portion of the 32-bit floating-point value in MRb

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ    MR2, #19.625 ; MR2 = 19.625 (0x419D0000)
MFRACF32  MR3, MR2     ; MR3 = MFRACF32(MR2) = 0.625 (0x3F200000)0
```

See also

MI16TOF32 MRa, MRb *Convert 16-bit Integer to 32-bit Floating-Point Value*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 1000 0000
```

Description

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

```
MRa = MI16TOF32(MRb);
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ    MR0, #0x0000    ; MR0(31:16) = 0.0 (0x0000)
MMOVXI    MR0, #0x0004    ; MR0(15:0) = 4.0 (0x0004)
MI16TOF32 MR1, MR0        ; MR1 = MI16TOF32 (MR0) = 4.0 (0x40800000)

MMOVIZ    MR2, #0x0000    ; MR2(31:16) = 0.0 (0x0000)
MMOVXI    MR2, #0xFFFC    ; MR2(15:0) = -4.0 (0xFFFC)
MI16TOF32 MR3, MR2        ; MR3 = MI16TOF32 (MR2) = -4.0 (0xC0800000)
MSTOP
```

See also

[MF32TOI16 MRa, MRb](#)
[MF32TOI16R MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MI16TOF32 MRa, mem16 *Convert 16-bit Integer to 32-bit Floating-Point Value*

Operands

| | |
|-------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| mem16 | 16-bit source memory location to be converted |

Opcode

LSW: mmmmm mmmmm mmmmm mmmmm
MSW: 0111 0101 00aa addr

Description

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

MRa = MI16TOF32[mem16];

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction:

Example

```
; Assume A = 4 (0x0004)
;           B = -4 (0xFFFC)

MI16TOF32 MR0, @_A ; MR0 = MI16TOF32(A) = 4.0 (0x40800000)
MI16TOF32 MR1, @_B ; MR1 = MI16TOF32(B) = -4.0 (0xC0800000)
```

See also

[MF32TOI16 MRa, MRb](#)
[MF32TOI16R MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MUI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MI32TOF32 MRa, mem32 *Convert 32-bit Integer to 32-bit Floating-Point Value*

Operands

| | |
|-------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| mem32 | 32-bit memory source for the MMOV32 operation. |

Opcode

LSW: `mmmm mmmm mmmm mmmm`
MSW: `0111 0100 01aa addr`

Description

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

MRa = MI32TOF32[mem32];

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; Given X, M and B are IQ24 numbers:
; X = IQ24(+2.5) = 0x02800000
; M = IQ24(+1.5) = 0x01800000
; B = IQ24(-0.5) = 0xFF800000
;
; Calculate Y = X * M + B
;
; Convert M, X and B from IQ24 to float
;

_ClalTask3:
    MI32TOF32 MR0, @_M      ; MR0 = 0x4BC00000
    MI32TOF32 MR1, @_X      ; MR1 = 0x4C200000
    MI32TOF32 MR2, @_B      ; MR2 = 0xCB000000
    MMPYF32 MR0, MR0, #0x3380 ; M = 1/(1*2^24) * iqm = 1.5 (0x3FC00000)
    MMPYF32 MR1, MR1, #0x3380 ; X = 1/(1*2^24) * iqx = 2.5 (0x40200000)
    MMPYF32 MR2, MR2, #0x3380 ; B = 1/(1*2^24) * iqb = -.5 (0xBF000000)
    MMPYF32 MR3, MR0, MR1     ; M*X
    MADDF32 MR2, MR2, MR3     ; Y=MX+B = 3.25 (0x40500000)

; Convert Y from float32 to IQ24
    MMPYF32 MR2, MR2, #0x4B80 ; Y * 1*2^24
    MF32TOI32 MR2, MR2        ; IQ24(Y) = 0x03400000
    MMOV32 @_Y, MR2           ; store result
    MSTOP                     ; end of task
```

See also

[MF32TOI32 MRa, MRb](#)
[MF32TOUI32 MRa, MRb](#)
[MI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, mem32](#)

MI32TOF32 MRa, MRb *Convert 32-bit Integer to 32-bit Floating-Point Value*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 1000 0000
```

Description

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

```
MRa = MI32TOF32(MRb);
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; Example1:
;
MMOVIZ    MR2, #0x1111 ; MR2(31:16) = 4369 (0x1111)
MMOVXI    MR2, #0x1111 ; MR2(15:0) = 4369 (0x1111)
; MR2 = +286331153 (0x11111111)
MI32TOF32 MR3, MR2     ; MR3 = MI32TOF32 (MR2) = 286331153.0 (0x4D888888)
```

See also

[MF32TOI32 MRa, MRb](#)
[MF32TOUI32 MRa, MRb](#)
[MI32TOF32 MRa, mem32](#)
[MUI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, mem32](#)

MLSL32 MRa, #SHIFT *Logical Shift Left*

Operands

| | |
|--------|---|
| MRa | CLA floating-point source/destination register (MR0 to MR3) |
| #SHIFT | Number of bits to shift (1 to 32) |

Opcode

```
LSW: 0000 0000 0shi ftaa
MSW: 0111 1011 1100 0000
```

Description

Logical shift left of MRa by the number of bits indicated. The number of bits can be 1 to 32.

```
MARa(31:0) = Logical Shift Left(MARa(31:0) by #SHIFT bits);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
; Given m2 = (int32)32
;          x2 = (int32)64
;          b2 = (int32)-128
;
; Calculate:
;          m2 = m2*2
;          x2 = x2*4
;          b2 = b2*8
;
_ClalTask3:
    MMOV32 MR0, @_m2    ; MR0 = 32 (0x00000020)
    MMOV32 MR1, @_x2    ; MR1 = 64 (0x00000040)
    MMOV32 MR2, @_b2    ; MR2 = -128 (0xFFFFF80)
    MLSL32 MR0, #1      ; MR0 = 64 (0x00000040)
    MLSL32 MR1, #2      ; MR1 = 256 (0x00000100)
    MLSL32 MR2, #3      ; MR2 = -1024 (0xFFFFFC00)
    MMOV32 @_m2, MR0    ; Store results
    MMOV32 @_x2, MR1
    MMOV32 @_b2, MR2
    MSTOP                ; end of task
```

See also

[MADD32 MRa, MRb, MRc](#)
[MASR32 MRa, #SHIFT](#)
[MAND32 MRa, MRb, MRc](#)
[MLSR32 MRa, #SHIFT](#)
[MOR32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)
[MSUB32 MRa, MRb, MRc](#)

MLSR32 MRa, #SHIFT *Logical Shift Right*

Operands

| | |
|--------|---|
| MRa | CLA floating-point source/destination register (MR0 to MR3) |
| #SHIFT | Number of bits to shift (1 to 32) |

Opcode

```
LSW: 0000 0000 0shi ftaa
MSW: 0111 1011 1000 0000
```

Description

Logical shift right of MRa by the number of bits indicated. The number of bits can be 1 to 32. Unlike the arithmetic shift (MASR32), the logical shift does not preserve the number's sign bit. Every bit in the operand is moved the specified number of bit positions, and the vacant bit-positions are filled in with zeros

```
MARa(31:0) = Logical Shift Right(MARa(31:0) by #SHIFT bits);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
; Illustrate the difference between MASR32 and MLSR32
```

```
MMOVIZ MR0, #0xAAAA ; MR0 = 0xAAAA5555
MMOVXI MR0, #0x5555
```

```
MMOV32 MR1, MR0 ; MR1 = 0xAAAA5555
MMOV32 MR2, MR0 ; MR2 = 0xAAAA5555
```

```
MASR32 MR1, #1 ; MR1 = 0xD5552AAA
MLSR32 MR2, #1 ; MR2 = 0x55552AAA
```

```
MASR32 MR1, #1 ; MR1 = 0xEAAA9555
MLSR32 MR2, #1 ; MR2 = 0x2AAA9555
```

```
MASR32 MR1, #6 ; MR1 = 0xFFAAA555
MLSR32 MR2, #6 ; MR2 = 0x00AAA555
```

See also

[MADD32 MRa, MRb, MRc](#)
[MASR32 MRa, #SHIFT](#)
[MAND32 MRa, MRb, MRc](#)
[MLSL32 MRa, #SHIFT](#)
[MOR32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)
[MSUB32 MRa, MRb, MRc](#)

MMACF32 MR3, MR2, MRd, MRe, MRf ||MMOV32 MRa, mem32 32-bit Floating-Point Multiply and Accumulate with Parallel Move

Operands

| | |
|-------|--|
| MR3 | floating-point destination/source register MR3 for the add operation |
| MR2 | CLA floating-point source register MR2 for the add operation |
| MRd | CLA floating-point destination register (MR0 to MR3) for the multiply operation MRd cannot be the same register as MRa |
| MRe | CLA floating-point source register (MR0 to MR3) for the multiply operation |
| MRf | CLA floating-point source register (MR0 to MR3) for the multiply operation |
| MRa | CLA floating-point destination register for the MMOV32 operation (MR0 to MR3). MRa cannot be MR3 or the same register as MRd. |
| mem32 | 32-bit source for the MMOV32 operation |

Opcode

```
LSW: mmm mmm mmm mmm
MSW: 0011 ffee ddaa addr
```

Description

Multiply and accumulate the contents of floating-point registers and move from register to memory. The destination register for the MMOV32 cannot be the same as the destination registers for the MMACF32.

```
MR3 = MR3 + MR2;
MRd = MRe * MRf;
MRa = [mem32];
```

Restrictions

The destination registers for the MMACF32 and the MMOV32 must be unique. That is, MRa cannot be MR3 and MRa cannot be the same register as MRd.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMACF32 (add or multiply) generates an underflow condition.
- LVF = 1 if MMACF32 (add or multiply) generates an overflow condition.

MMOV32 sets the NF and ZF flags as follows:

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

Pipeline

MMACF32 and MMOV32 complete in a single cycle.

Example 1

```

; Perform 5 multiply and accumulate operations:
;
; X and Y are 32-bit floating point arrays
;
; 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
;
_ClalTask1:
    MMIOV16 MAR0, #_X          ; MAR0 points to X array
    MMIOV16 MAR1, #_Y          ; MAR1 points to Y array
    MNOP                       ; Delay for MAR0, MAR1 load
    MNOP                       ; Delay for MAR0, MAR1 load
    ; <-- MAR0 valid
    MMIOV32 MR0, *MAR0[2]++    ; MR0 = X0, MAR0 += 2
    ; <-- MAR1 valid
    MMIOV32 MR1, *MAR1[2]++    ; MR1 = Y0, MAR1 += 2
    ;
    MMPYF32 MR2, MR0, MR1      ; MR2 = A = X0 * Y0
    || MMIOV32 MR0, *MAR0[2]++  ; In parallel MR0 = X1, MAR0 += 2
    MMIOV32 MR1, *MAR1[2]++    ; MR1 = Y1, MAR1 += 2
    ;
    MMPYF32 MR3, MR0, MR1      ; MR3 = B = X1 * Y1
    || MMIOV32 MR0, *MAR0[2]++  ; In parallel MR0 = X2, MAR0 += 2
    MMIOV32 MR1, *MAR1[2]++    ; MR1 = Y2, MAR2 += 2
    ;
    MMACF32 MR3, MR2, MR2, MR0, MR1 ; MR3 = A + B, MR2 = C = X2 * Y2
    || MMIOV32 MR0, *MAR0[2]++  ; In parallel MR0 = X3
    MMIOV32 MR1, *MAR1[2]++    ; MR1 = Y3 M
    ;
    MACF32 MR3, MR2, MR2, MR0, MR1 ; MR3 = (A + B) + C, MR2 = D = X3 * Y3
    || MMIOV32 MR0, *MAR0
    MMIOV32 MR1, *MAR1          ; MR1 = Y4
    ;
    MMPYF32 MR2, MR0, MR1      ; MR2 = E = X4 * Y4
    || MADD32 MR3, MR3, MR2     ; in parallel MR3 = (A + B + C) + D
    ;
    MADD32 MR3, MR3, MR2       ; MR3 = (A + B + C + D) + E
    MMIOV32 @_Result, MR3      ; Store the result
    MSTOP                     ; end of task

```


Example 2

```

; sum = X0*B0 + X1*B1 + X2*B2 + Y1*A1 + Y2*B2
;
;      X2 = X1
;      X1 = X0
;      Y2 = Y1 ; Y1 = sum
;
_ClaTask2:
    MMOV32    MR0, @_B2      ; MR0 = B2
    MMOV32    MR1, @_X2      ; MR1 = X2
    MMPYF32   MR2, MR1, MR0  ; MR2 = X2*B2
    || MMOV32    MR0, @_B1      ; MR0 = B1
    MMOVD32   MR1, @_X1      ; MR1 = X1, X2 = X1
    MMPYF32   MR3, MR1, MR0  ; MR3 = X1*B1
    || MMOV32    MR0, @_B0      ; MR0 = B0
    MMOVD32   MR1, @_X0      ; MR1 = X0, X1 = X0

;   MR3 = X1*B1 + X2*B2, MR2 = X0*B0
;   MR0 = A2
    MMACF32   MR3, MR2, MR2, MR1, MR0
    || MMOV32   MR0, @_A2 M

    MOV32     MR1, @_Y2      ; MR1 = Y2

;   MR3 = X0*B0 + X1*B1 + X2*B2, MR2 = Y2*A2
;   MR0 = A1
    MMACF32   MR3, MR2, MR2, MR1, MR0
    || MMOV32   MR0, @_A1

    MMOVD32   MR1, @_Y1      ; MR1 = Y1, Y2 = Y1
    MADDF32   MR3, MR3, MR2   ; MR3 = Y2*A2 + X0*B0 + X1*B1 + X2*B2
    || MMPYF32   MR2, MR1, MR0 ; MR2 = Y1*A1
    MADDF32   MR3, MR3, MR2   ; MR3 = Y1*A1 + Y2*A2 + X0*B0 + X1*B1 + X2*B2
    MMOV32    @_Y1, MR3      ; Y1 = MR3
    MSTOP                      ; end of task

```

See also
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRe, MRf](#)

MMAXF32 MRa, MRb 32-bit Floating-Point Maximum

Operands

| | |
|-----|---|
| MRa | CLA floating-point source/destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 0010 0000
```

Description

```
if(MRa < MRb) MRa = MRb;
```

Special cases for the output from the MMAXF32 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | 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(MRa == MRb) {ZF=1; NF=0;}
if(MRa > MRb) {ZF=0; NF=0;}
if(MRa < MRb) {ZF=0; NF=1;}
```

Pipeline

This is a single-cycle instruction.

Example 1

```
MMOVIZ MR0, #5.0 ; MR0 = 5.0 (0x40A00000)
MMOVIZ MR1, #-2.0 ; MR1 = -2.0 (0xC0000000)
MMOVIZ MR2, #-1.5 ; MR2 = -1.5 (0xBFC00000)
MMAXF32 MR2, MR1 ; MR2 = -1.5, ZF = NF = 0
MMAXF32 MR1, MR2 ; MR1 = -1.5, ZF = 0, NF = 1
MMAXF32 MR2, MR0 ; MR2 = 5.0, ZF = 0, NF = 1
MAXF32 MR0, MR2 ; MR2 = 5.0, ZF = 1, NF = 0
```

Example 2

```
; X is an array of 32-bit floating-point values
; Find the maximum value in an array X
; and store it in Result
;
_ClalTask1:
  MMOV16 MAR1, #_X ; Start address
  MUI16TOF32 MR0, @_len ; Length of the array
  MNOP ; delay for MAR1 load
  MNOP ; delay for MAR1 load
  MMOV32 MR1, *MAR1[2]++ ; MR1 = X0
LOOP
  MMOV32 MR2, *MAR1[2]++ ; MR2 = next element
  MMAXF32 MR1, MR2 ; MR1 = MAX(MR1, MR2)
  MADDF32 MR0, MR0, #-1.0 ; Decrement the counter
  MCMPF32 MR0 #0.0 ; Set/clear flags for MBCNDD
  MNOP
  MNOP
  MNOP
  MBCNDD LOOP, NEQ ; Branch if not equal to zero
  MMOV32 @_Result, MR1 ; Always executed
  MNOP ; Always executed
  MNOP ; Always executed
  MSTOP ; End of task
```

See also

[MCMPF32 MRa, MRb](#)
[MCMPF32 MRa, #16FHi](#)
[MMAXF32 MRa, #16FHi](#)
[MMINF32 MRa, MRb](#)
[MMINF32 MRa, #16FHi](#)

MMAXF32 MRa, #16FHi 32-bit Floating-Point Maximum

Operands

| | |
|--------|---|
| MRa | CLA floating-point source/destination register (MR0 to MR3) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

```
LSW: I I I I I I I I I I I I I I I I
MSW: 0 1 1 1 1 0 0 1 0 0 0 0 0 0 a a
```

Description

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

```
if(MRa < #16FHi:0) MRa = #16FHi:0;
```

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

Special cases for the output from the MMAXF32 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | 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(MRa == #16FHi:0) {ZF=1; NF=0;}
if(MRa > #16FHi:0) {ZF=0; NF=0;}
if(MRa < #16FHi:0) {ZF=0; NF=1;}
```

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ    MR0, #5.0 ; MR0 = 5.0 (0x40A00000)
MMOVIZ    MR1, #4.0 ; MR1 = 4.0 (0x40800000)
MMOVIZ    MR2, #-1.5 ; MR2 = -1.5 (0xBFC00000)
MMAXF32   MR0, #5.5 ; MR0 = 5.5, ZF = 0, NF = 1
MMAXF32   MR1, #2.5 ; MR1 = 4.0, ZF = 0, NF = 0
MMAXF32   MR2, #-1.0 ; MR2 = -1.0, ZF = 0, NF = 1
MMAXF32   MR2, #-1.0 ; MR2 = -1.5, ZF = 1, NF = 0
```

See also

[MMAXF32 MRa, MRb](#)
[MMINF32 MRa, MRb](#)
[MMINF32 MRa, #16FHi](#)

MMINF32 MRa, MRb 32-bit Floating-Point Minimum

Operands

| | |
|-----|---|
| MRa | CLA floating-point source/destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 0100 0000
```

Description

```
if(MRa > MRb) MRa = MRb;
```

Special cases for the output from the MMINF32 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | 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(MRa == MRb) {ZF=1; NF=0;}
if(MRa > MRb) {ZF=0; NF=0;}
if(MRa < MRb) {ZF=0; NF=1;}
```

Pipeline

This is a single-cycle instruction.

Example 1

```
MMOVIZ MR0, #5.0 ; MR0 = 5.0 (0x40A00000)
MMOVIZ MR1, #4.0 ; MR1 = 4.0 (0x40800000)
MMOVIZ MR2, #-1.5 ; MR2 = -1.5 (0xBFC00000)
MMINF32 MR0, MR1 ; MR0 = 4.0, ZF = 0, NF = 0
MMINF32 MR1, MR2 ; MR1 = -1.5, ZF = 0, NF = 0
MMINF32 MR2, MR1 ; MR2 = -1.5, ZF = 1, NF = 0
MMINF32 MR1, MR0 ; MR2 = -1.5, ZF = 0, NF = 1
```

Example 2

```
;
; X is an array of 32-bit floating-point values
; Find the minimum value in an array X
; and store it in Result
;

_ClalTask1:
MMOVI16    MAR1, #_X          ; Start address
MUI16TOF32 MR0, @_len         ; Length of the array
MNOP                          ; delay for MAR1 load
MNOP                          ; delay for MAR1 load
MMOV32     MR1, *MAR1[2]++    ; MR1 = X0
LOOP
MMOV32     MR2, *MAR1[2]++    ; MR2 = next element
MMINF32    MR1, MR2           ; MR1 = MAX(MR1, MR2)
MADDF32    MR0, MR0, #-1.0    ; Decrement the counter
MCMPPF32   MR0 #0.0           ; Set/clear flags for MBCNDD
MNOP
MNOP
MNOP
MBCNDD     LOOP, NEQ          ; Branch if not equal to zero
MMOV32     @_Result, MR1      ; Always executed
MNOP                          ; Always executed
MNOP                          ; Always executed
MSTOP
```

See also

[MMAXF32 MRa, MRb](#)
[MMAXF32 MRa, #16FHi](#)
[MMINF32 MRa, #16FHi](#)

MMINF32 MRa, #16FHi 32-bit Floating-Point Minimum

Operands

| | |
|--------|---|
| MRa | floating-point source/destination register (MR0 to MR3) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

```
LSW: IIII IIII IIII IIII
MSW: 0111 1001 0100 00aa
```

Description

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

```
if(MRa > #16FHi:0) MRa = #16FHi:0;
```

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

Special cases for the output from the MMINF32 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | 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(MRa == #16FHi:0) {ZF=1; NF=0;}
if(MRa > #16FHi:0) {ZF=0; NF=0;}
if(MRa < #16FHi:0) {ZF=0; NF=1;}
```

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ MR0, #5.0 ; MR0 = 5.0 (0x40A00000)
MMOVIZ MR1, #4.0 ; MR1 = 4.0 (0x40800000)
MMOVIZ MR2, #-1.5 ; MR2 = -1.5 (0xBFC00000)
MMINF32 MR0, #5.5 ; MR0 = 5.0, ZF = 0, NF = 1
MMINF32 MR1, #2.5 ; MR1 = 2.5, ZF = 0, NF = 0
MMINF32 MR2, #-1.0 ; MR2 = -1.5, ZF = 0, NF = 1
MMINF32 MR2, #-1.5 ; MR2 = -1.5, ZF = 1, NF = 0
```

See also

[MMAXF32 MRa, #16FHi](#)
[MMAXF32 MRa, MRb](#)
[MMINF32 MRa, MRb](#)

MMOV16 MARx, MRa, #16I Load the Auxiliary Register with MRa + 16-bit Immediate Value

Operands

| | |
|------|--|
| MARx | Auxiliary register MAR0 or MAR1 |
| MRa | CLA Floating-point register (MR0 to MR3) |
| #16I | 16-bit immediate value |

Opcode

```
LSW: IIII IIII IIII IIII (opcode of MMOV16 MAR0, MRa, #16I)
MSW: 0111 1111 1101 00AA
```

```
LSW: IIII IIII IIII IIII (opcode of MMOV16 MAR1, MRa, #16I)
MSW: 0111 1111 1111 00AA
```

Description

Load the auxiliary register, MAR0 or MAR1, with MRa(15:0) + 16-bit immediate value. Refer to the pipeline section for important information regarding this instruction.

MARx = MRa(15:0) + #16I;

Flags

This instruction does not modify flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction. The load of MAR0 or MAR1 will occur in the EXE phase of the pipeline. Any post increment of MAR0 or MAR1 using indirect addressing will occur in the D2 phase of the pipeline. Therefore the following applies when loading the auxiliary registers:

- I1 and I2**

The two instructions following MMOV16 will use MAR0/MAR1 before the update occurs. Thus these two instructions will use the old value of MAR0 or MAR1.

- I3**

Loading of an auxiliary register occurs in the EXE phase while updates due to post-increment addressing occur in the D2 phase. Thus I3 cannot use the auxiliary register or there will be a conflict. In the case of a conflict, the update due to address-mode post increment will win and the auxiliary register will not be updated with #_X.

- I4**

Starting with the 4th instruction MAR0 or MAR1 will be the new value loaded with MMOV16.

; Assume MAR0 is 50, MR0 is 10, and #_X is 20

```
MMOV16 MAR0, MR0, #_X      ; Load MAR0 with address of X (20) + MR0 (10)
<Instruction 1>              ; I1 Will use the old value of MAR0 (50)
<Instruction 2>              ; I2 Will use the old value of MAR0 (50)
<Instruction 3>              ; I3 Cannot use MAR0
<Instruction 4>              ; I4 Will use the new value of MAR0 (30)
<Instruction 5>              ; I5
```

Table 28. Pipeline Activity For MMOV16 MARx, MRa , #16I

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-----------------------|--------|--------|--------|--------|--------|--------|--------|---|
| MMOV16 MAR0, MR0, #_X | MMOV16 | | | | | | | |
| I1 | I1 | MMOV16 | | | | | | |
| I2 | I2 | I1 | MMOV16 | | | | | |
| I3 | I3 | I2 | I1 | MMOV16 | | | | |
| I4 | I4 | I3 | I2 | I1 | MMOV16 | | | |
| I5 | I5 | I4 | I3 | I2 | I1 | MMOV16 | | |
| I6 | I6 | I5 | I4 | I3 | I2 | I1 | MMOV16 | |

Example 1

```

; Calculate an offset into a sin/cos table
;
_ClalTask1:
    MMOV32 MR0,@_rad                ; MR0 = rad
    MMOV32 MR1,@_TABLE_SIZEDivTwoPi ; MR1 = TABLE_SIZE/(2*Pi)
    MMPYF32 MR1,MR0,MR1             ; MR1 = rad* TABLE_SIZE/(2*Pi)
    || MMOV32 MR2,@_TABLE_MASK       ; MR2 = TABLE_MASK
    MF32TOI32 MR3,MR1               ; MR3 = K=int(rad*TABLE_SIZE/(2*Pi))
    MAND32 MR3,MR3,MR2              ; MR3 = K & TABLE_MASK
    ML32L32 MR3,#1                  ; MR3 = K * 2

    MMOV16 MAR0,MR3,#_Cos0          ; MAR0 K*2+addr of table.Cos0
    MFRACF32 MR1,MR1               ; I1
    MMOV32 MR0,@_TwoPiDivTABLE_SIZE ; I2
    MMPYF32 MR1,MR1,MR0            ; I3
    || MMOV32 MR0,@_Coef3

    MMOV32 MR2,*MAR0[#-64]++        ; MR2 = *MAR0, MAR0 += (-64)
    ...
    ...
    MSTOP ; end of task

```

Example 2

```

; This task logs the last NUM_DATA_POINTS
; ADCRESULT1 values in the array VoltageCLA
;
; When the last element in the array has been
; filled, the task will go back to the
; the first element.
;
; Before starting the ADC conversions, force
; Task 8 to initialize the ConversionCount to zero
;
_ClalTask2:
    MMOVZ16    MR0, @_ConversionCount    ;I1 Current Conversion
    MMOV16 MAR1, MR0, #_VoltageCLA      ;I2 Next array location
    MUI16TOF32 MR0, MR0                 ;I3 Convert count to float32
    MADDF32    MR0, MR0, #1.0           ;I4 Add 1 to conversion count
    MCMPF32    MR0, #NUM_DATA_POINTS.0 ;I5 Compare count to max
    MF32TOUI16 MR0, MR0                 ;I6 Convert count to Uint16
    MNOP                          ;I7 Wait till I8 to read result
    MMOVZ16    MR2, @_AdcResult.ADCRESULT1 ;I8 Read ADCRESULT1
    MMOV16     *MAR1, MR2                ; Store ADCRESULT1
    MBCNDD     _RestartCount, GEQ        ; If count >= NUM_DATA_POINTS
    MMOVIZ     MR1, #0.0                 ; Always executed: MR1=0
    MNOP
    MNOP
    MMOV16     @_ConversionCount, MR0    ; If branch not taken
    MSTOP                      ; store current count
_RestartCount
    MMOV16     @_ConversionCount, MR1    ; If branch taken, restart count
    MSTOP                      ; end of task

; This task initializes the ConversionCount
; to zero
;
_ClalTask8:
    MMOVIZ MR0, #0.0
    MMOV16 @_ConversionCount, MR0
    MSTOP
_ClalTask8End:

```

See also

MMOV16 MARx, mem16 *Load MAR1 with 16-bit Value*

Operands

| | |
|-------|--|
| MARx | CLA auxiliary register MAR0 or MAR1 |
| mem16 | 16-bit destination memory accessed using indirect or direct addressing modes |

Opcode

```

LSW: mmm mmm mmm mmm (Opcode for MMOV16 MAR0, mem16)
MSW: 0111 0110 0000 addr

LSW: mmm mmm mmm mmm (Opcode for MMOV16 MAR1, mem16)
MSW: 0111 0110 0100 addr

```

Description

Load MAR0 or MAR1 with the 16-bit value pointed to by mem16. Refer to the pipeline section for important information regarding this instruction.

MAR1 = [mem16];

Flags

No flags MSTF flags are affected.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction. The load of MAR0 or MAR1 will occur in the EXE phase of the pipeline. Any post increment of MAR0 or MAR1 using indirect addressing will occur in the D2 phase of the pipeline. Therefore the following applies when loading the auxiliary registers:

- I1 and I2**

The two instructions following MMOV16 will use MAR0/MAR1 before the update occurs. Thus these two instructions will use the old value of MAR0 or MAR1.

- I3**

Loading of an auxiliary register occurs in the EXE phase while updates due to post-increment addressing occur in the D2 phase. Thus I3 cannot use the auxiliary register or there will be a conflict. In the case of a conflict, the update due to address-mode post increment will win and the auxiliary register will not be updated with #_X.

- I4**

Starting with the 4th instruction MAR0 or MAR1 will be the new value loaded with MMOV16.

; Assume MAR0 is 50 and @_X is 20

```

MMOV16 MAR0, @_X          ; Load MAR0 with the contents of X (20)
<Instruction 1>             ; I1 Will use the old value of MAR0 (50)
<Instruction 2>             ; I2 Will use the old value of MAR0 (50)
<Instruction 3>             ; I3 Cannot use MAR0
<Instruction 4>             ; I4 Will use the new value of MAR0 (20)
<Instruction 5>             ; I5
....

```

Table 29. Pipeline Activity For MMOV16 MAR0/MAR1, mem16

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|------------------|--------|--------|--------|--------|--------|--------|--------|---|
| MMOV16 MAR0, @_X | MMOV16 | | | | | | | |
| I1 | I1 | MMOV16 | | | | | | |
| I2 | I2 | I1 | MMOV16 | | | | | |
| I3 | I3 | I2 | I1 | MMOV16 | | | | |
| I4 | I4 | I3 | I2 | I1 | MMOV16 | | | |
| I5 | I5 | I4 | I3 | I2 | I1 | MMOV16 | | |
| I6 | I6 | I5 | I4 | I3 | I2 | I1 | MMOV16 | |

Example

```

; This task logs the last NUM_DATA_POINTS
; ADCRESULT1 values in the array VoltageCLA
;
; When the last element in the array has been
; filled, the task will go back to the
; the first element.
;
; Before starting the ADC conversions, force
; Task 8 to initialize the ConversionCount to zero
;
_Cla1Task2:
    MOVZ16    MR0, @_ConversionCount    ;I1 Current Conversion
    MOV16     MAR1, MR0, #_VoltageCLA    ;I2 Next array location
    MUI16TOF32 MR0, MR0                 ;I3 Convert count to float32
    MADDF32    MR0, MR0, #1.0           ;I4 Add 1 to conversion count
    MCMPPF32   MR0, #NUM_DATA_POINTS.0 ;I5 Compare count to max
    MF32TOUI16 MR0, MR0                 ;I6 Convert count to Uint16
    MNOP                          ;I7 Wait till I8 to read result
    MOVZ16     MR2, @_AdcResult.ADCRESULT1 ;I8 Read ADCRESULT1
    MOV16      *MAR1, MR2                ; Store ADCRESULT1
    MBCNDD     _RestartCount, GEQ        ; If count >= NUM_DATA_POINTS
    MOVIZ      MR1, #0.0                 ; Always executed: MR1=0
    MNOP
    MNOP
    MOV16      @_ConversionCount, MR0    ; If branch not taken MSTOP
                                          ; store current count

_RestartCount
    MOV16      @_ConversionCount, MR1    ; If branch taken, restart count
    MSTOP                                           ; end of task

; This task initializes the ConversionCount
; to zero
;
_Cla1Task8:
    MOVIZ      MR0, #0.0
    MOV16      @_ConversionCount, MR0
    MSTOP
_ClaT8End:

```

See also

MMOV16 mem16, MARx *Move 16-bit Auxiliary Register Contents to Memory*

Operands

| | |
|-------|--|
| mem16 | 16-bit destination memory accessed using indirect or direct addressing modes |
| MARx | CLA auxiliary register MAR0 or MAR1 |

Opcode

LSW: `mmmm mmmm mmmm mmmm` (Opcode for MMOV16 mem16, MAR0)
MSW: `0111 0110 1000 addr`

LSW: `mmmm mmmm mmmm mmmm` (Opcode for MMOV16 mem16, MAR1)
MSW: `0111 0110 1100 addr`

Description

Store the contents of MAR0 or MAR1 in the 16-bit memory location pointed to by mem16.

[mem16] = MAR0;

Flags

No flags MSTF flags are affected.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example
See also

MMOV16 mem16, MRa *Move 16-bit Floating-Point Register Contents to Memory*

Operands

| | |
|-------|--|
| mem16 | 16-bit destination memory accessed using indirect or direct addressing modes |
| MRa | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: mmmmm mmmmm mmmmm mmmmm
MSW: 0111 0101 11aa addr
```

Description

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

```
[mem16] = MRa(15:0);
```

Flags

No flags MSTF flags are affected.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; This task logs the last NUM_DATA_POINTS
; ADCRESULT1 values in the array VoltageCLA
;
; When the last element in the array has been
; filled, the task will go back to the
; the first element.
;
; Before starting the ADC conversions, force
; Task 8 to initialize the ConversionCount to zero
;
_ClalTask2:
    MMOVZ16    MR0, @_ConversionCount    ;I1 Current Conversion
    MMOV16     MAR1, MR0, #_VoltageCLA    ;I2 Next array location
    MUI16TOF32 MR0, MR0                  ;I3 Convert count to float32
    MADDF32    MR0, MR0, #1.0             ;I4 Add 1 to conversion count
    MCMPF32    MR0, #NUM_DATA_POINTS.0    ;I5 Compare count to max
    MF32TOUI16 MR0, MR0                  ;I6 Convert count to Uint16
    MNOP
    MMOVZ16    MR2, @_AdcResult.ADCRESULT1 ;I8 Read ADCRESULT1
    MMOV16     *MAR1, MR2                  ; Store ADCRESULT1
    MBCNDD     _RestartCount, GEQ         ; If count >= NUM_DATA_POINTS
    MMOVIZ     MR1, #0.0                  ; Always executed: MR1=0
    MNOP
    MNOP
    MMOV16     @_ConversionCount, MR0     ; If branch not taken MSTOP
                                           ; store current count
_RestartCount
    MMOV16     @_ConversionCount, MR1     ; If branch taken, restart count
    MSTOP
                                           ; end of task

; This task initializes the ConversionCount
; to zero
;
_ClalTask8:
    MMOVIZ MR0, #0.0
    MMOV16 @_ConversionCount, MR0
    MSTOP
_ClalTask8End:
```

See also

[MMOVIZ MRa, #16FHiHex](#)
[MMOVXI MRa, #16FLoHex](#)

MMOV32 mem32, MRa *Move 32-bit Floating-Point Register Contents to Memory*

Operands

| | |
|-------|--|
| MRa | floating-point register (MR0 to MR3) |
| mem32 | 32-bit destination memory accessed using indirect or direct addressing modes |

Opcode

LSW: mmmmm mmmmm mmmmm mmmmm
MSW: 0111 0100 11aa addr

Description

Move from MRa to 32-bit memory location indicated by mem32.

[mem32] = MRa;

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

No flags affected.

Pipeline

This is a single-cycle instruction.

Example

```
; Perform 5 multiply and accumulate operations:
;
; X and Y are 32-bit floating point arrays;
; 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
;
_ClalTask1:
    MMOV16    MAR0, #_X           ; MAR0 points to X array
    MMOV16    MAR1, #_Y           ; MAR1 points to Y array
    MNOP      ; Delay for MAR0, MAR1 load
    MNOP      ; Delay for MAR0, MAR1 load
    ; <-- MAR0 valid
    MMOV32    MR0, *MAR0[2]++     ; MR0 = X0, MAR0 += 2
    ; <-- MAR1 valid
    MMOV32    MR1, *MAR1[2]++     ; MR1 = Y0, MAR1 += 2
    MMPYF32   MR2, MR0, MR1       ; MR2 = A = X0 * Y0
    ; In parallel MR0 = X1, MAR0 += 2
    MMOV32    MR0, *MAR0[2]++     ; MR0 = X1, MAR0 += 2
    MMOV32    MR1, *MAR1[2]++     ; MR1 = Y1, MAR1 += 2
    MMPYF32   MR3, MR0, MR1       ; MR3 = B = X1 * Y1
    ; In parallel MR0 = X2, MAR0 += 2
    MMOV32    MR0, *MAR0[2]++     ; MR0 = X2, MAR0 += 2
    MMOV32    MR1, *MAR1[2]++     ; MR1 = Y2, MAR2 += 2
    ;
    MMACF32   MR3, MR2, MR2, MR0, MR1 ; MR3 = A + B, MR2 = C = X2 * Y2
    ; In parallel MR0 = X3
    MMOV32    MR0, *MAR0[2]++     ; MR0 = X3
    MMOV32    MR1, *MAR1[2]++     ; MR1 = Y3
    ;
    MMACF32   MR3, MR2, MR2, MR0, MR1 ; MR3 = (A + B) + C, MR2 = D = X3 * Y3
    ; In parallel MR0 = X4
    MMOV32    MR1, *MAR1
    ; MR1 = Y4
    MMPYF32   MR2, MR0, MR1       ; MR2 = E = X4 * Y4
    ; in parallel MR3 = (A + B + C) + D
    MADDF32   MR3, MR3, MR2       ; MR3 = (A + B + C + D) + E
    MADDF32   MR3, MR3, MR2
    MMOV32    @_Result, MR3       ; Store the result MSTOP ; end of task
```

See also

[MMOV32 mem32, MSTF](#)

MMOV32 mem32, MSTF *Move 32-bit MSTF Register to Memory*
Operands

| | |
|-------|--------------------------------|
| MSTF | floating-point status register |
| mem32 | 32-bit destination memory |

Opcode

LSW: mmmmm mmmmm mmmmm mmmmm
MSW: 0111 0111 0100 addr

Description

Copy the CLA's floating-point status register, MSTF, to memory.

[mem32] = MSTF;

Flags

This instruction does not modify flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example
See also

[MMOV32 mem32, MRa](#)

MMOV32 MRa, mem32 {, CNDF} *Conditional 32-bit Move*

Operands

| | |
|-------|---|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| mem32 | 32-bit memory location accessed using direct or indirect addressing |
| CNDF | optional condition. |

Opcode

```
LSW: mmm mmm mmm mmm
MSW: 0111 00cn dfaa addr
```

Description

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

```
if (CNDF == TRUE) MRa = [mem32];
```

CNDF is one of the following conditions:

| Encode ⁽¹⁾ | CNDF | Description | MSTF 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 OR 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 and NF 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

```
if (CNDF == UNCF)
{
    NF = MRa(31);
    ZF = 0;
    if (MRa(30:23) == 0) { ZF = 1; NF = 0; }
}
else No flags modified;
```

Pipeline

This is a single-cycle instruction.

Example

```

; Given A, B, X, M1 and M2 are 32-bit floating-point
; numbers
;
; if(A > B) calculate Y = X*M1
; if(A < B) calculate Y = X*M2
;
_ClalTask5:
    MMOV32    MR0, @_A
    MMOV32    MR1, @_B
    MCMPF32   MR0, MRB
    MMOV32    MR2, @_M1, EQ    ; if A > B, MR2 = M1
                                ;      Y = M1*X
    MMOV32    MR2, @_M2, NEQ   ; if A < B, MR2 = M2
                                ;      Y = M2*X

    MMOV32    MR3, @_X
    MMPYF32   MR3, MR2, MR3    ; Calculate Y
    MMOV32    @_Y, MR3         ; Store Y
    MSTOP                                ; end of task

```

See also

[MMOV32 MRa, MRb {, CNDF}](#)
[MMOVD32 MRa, mem32](#)

MMOV32 MRa, MRb {, CNDF} *Conditional 32-bit Move*

Operands

| | |
|------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| CNDF | optional condition. |

Opcode

```
LSW: 0000 0000 cndf bbaa
MSW: 0111 1010 1100 0000
```

Description

If the condition is true, then move the 32-bit value in MRb to the floating-point register indicated by MRa.

```
if (CNDF == TRUE) MRa = MRb;
```

CNDF is one of the following conditions:

| Encode ⁽³⁾ | CNDF | Description | MSTF 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 OR 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 ⁽⁴⁾ | Unconditional with flag modification | None |

⁽³⁾ Values not shown are reserved.

⁽⁴⁾ This is the default operation if no CNDF field is specified. This condition will allow the ZF, and NF 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

```
if (CNDF == UNCF)
{
    NF = MRa(31); ZF = 0;
    if (MRa(30:23) == 0) {ZF = 1; NF = 0;}
}
else No flags modified;
```

Pipeline

This is a single-cycle instruction.

Example

```

; Given: X = 8.0
;        Y = 7.0
;        A = 2.0
;        B = 5.0
; _ClaTask1
MMOV32  MR3, @_X      ; MR3 = X = 8.0
MMOV32  MR0, @_Y      ; MR0 = Y = 7.0
MMAXF32 MR3, MR0      ; ZF = 0, NF = 0, MR3 = 8.0
MMOV32  MR1, @_A, GT  ; true, MR1 = A = 2.0
MMOV32  MR1, @_B, LT  ; false, does not load MR1
MMOV32  MR2, MR1, GT  ; true, MR2 = MR1 = 2.0
MMOV32  MR2, MR0, LT  ; false, does not load MR2
MSTOP

```

See also

[MMOV32 MRa, mem32\[, CNDF\]](#)

MMOV32 MSTF, mem32 *Move 32-bit Value from Memory to the MSTF Register*

Operands

| | |
|-------|-------------------------------|
| MSTF | CLA status register |
| mem32 | 32-bit source memory location |

Opcode

LSW: mmmm mmmm mmmm mmmm
MSW: 0111 0111 0000 addr

Description

Move from memory to the CLA's status register MSTF. This instruction is most useful when nesting function calls (via MCCNDD).

MSTF = [mem32];

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|-----|-----|-----|-----|-----|
| Modified | Yes | Yes | Yes | Yes | Yes |

Loading the status register will overwrite all flags and the RPC field. The MEALLOW field is not affected.

Pipeline

This is a single-cycle instruction.

Example

See also

[MMOV32 mem32, MSTF](#)

MMOVD32 MRa, mem32 *Move 32-bit Value from Memory with Data Copy*

Operands

| | |
|-------|---|
| MRa | CLA floating-point register (MR0 to MR3) |
| mem32 | 32-bit memory location accessed using direct or indirect addressing |

Opcode

```
LSW: mmmmm mmmmm mmmmm mmmmm
MSW: 0111 0100 00aa addr
```

Description

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

```
MRa = [mem32];
[mem32+2] = [mem32];
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

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

Pipeline

This is a single-cycle instruction.

Example

```
; sum = X0*B0 + X1*B1 + X2*B2 + Y1*A1 + Y2*B2
;
;      X2 = X1
;      X1 = X0
;      Y2 = Y1
;      Y1 = sum
;
_ClalTask2:
    MMOVD32 MR0, @_B2      ; MR0 = B2
    MMOVD32 MR1, @_X2      ; MR1 = X2
    MMPYF32 MR2, MR1, MR0  ; MR2 = X2*B2
| | MMOVD32 MR0, @_B1      ; MR0 = B1
    MMOVD32 MR1, @_X1      ; MR1 = X1, X2 = X1
    MMPYF32 MR3, MR1, MR0  ; MR3 = X1*B1
| | MMOVD32 MR0, @_B0      ; MR0 = B0
    MMOVD32 MR1, @_X0      ; MR1 = X0, X1 = X0

; MR3 = X1*B1 + X2*B2, MR2 = X0*B0
; MR0 = A2
    MMACF32 MR3, MR2, MR2, MR1, MR0
| | MMOVD32 MR0, @_A2

    MMOVD32 MR1, @_Y2      ; MR1 = Y2

; MR3 = X0*B0 + X1*B1 + X2*B2, MR2 = Y2*A2
; MR0 = A1
    MMACF32 MR3, MR2, MR2, MR1, MR0
| | MMOVD32 MR0, @_A1

    MMOVD32 MR1, @_Y1      ; MR1 = Y1, Y2 = Y1
    MADDF32 MR3, MR3, MR2  ; MR3 = Y2*A2 + X0*B0 + X1*B1 + X2*B2
| | MMPYF32 MR2, MR1, MR0  ; MR2 = Y1*A1
    MADDF32 MR3, MR3, MR2  ; MR3 = Y1*A1 + Y2*A2 + X0*B0 + X1*B1 + X2*B2
    MMOVD32 @_Y1, MR3      ; Y1 = MR3
    MSTOP                  ; end of task
```

See also

[MMOV32 MRa, mem32 {,CNDF}](#)

MMOVF32 MRa, #32F *Load the 32-bits of a 32-bit Floating-Point Register*

Operands

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

```
MMOVIZ MRa, #16FHiHex  MMOVXI MRa, #16FLoHex
```

| | |
|------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| #32F | immediate float value represented in floating-point representation |

Opcode

```
LSW:  II II II II II II II II  (opcode of MMOVIZ MRa, #16FHiHex)
MSW:  0111 1000 0100 00aa
LSW:  II II II II II II II II  (opcode of MMOVXI MRa, #16FLoHex)
MSW:  0111 1000 1000 00aa
```

Description

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

Load the 32-bits of MRa 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.

```
MRa = #32F;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

Depending on #32FH, this instruction takes one or two cycles. If all of the lower 16-bits of the IEEE 32-bit floating-point format of #32F are zeros, then the assembler will convert MMOVF32 into only MMOVIZ instruction. If the lower 16-bits of the IEEE 32-bit floating-point format of #32F are not zeros, then the assembler will convert MMOVF32 into MMOVIZ and MMOVXI instructions.

Example

```
MMOVF32 MR1, #3.0      ; MR1 = 3.0 (0x40400000)
                        ; Assembler converts this instruction as
                        ; MMOVIZ MR1, #0x4040

MMOVF32 MR2, #0.0      ; MR2 = 0.0 (0x00000000)
                        ; Assembler converts this instruction as
                        ; MMOVIZ MR2, #0x0

MMOVF32 MR3, #12.265   ; MR3 = 12.625 (0x41443D71)
                        ; Assembler converts this instruction as
                        ; MMOVIZ MR3, #0x4144
                        ; MMOVXI MR3, #0x3D71
```

See also

[MMOVIZ MRa, #16FHi](#)
[MMOVXI MRa, #16FLoHex](#)
[MMOVI32 MRa, #32FHex](#)

MMOVI16 MARx, #16I Load the Auxiliary Register with the 16-bit Immediate Value

Operands

| | |
|------|---------------------------------|
| MARx | Auxiliary register MAR0 or MAR1 |
| #16I | 16-bit immediate value |

Opcode

```

LSW: I I I I I I I I I I I I I I (opcode of MMOVI16 MAR0, #16I)
MSW: 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0

LSW: I I I I I I I I I I I I I I (opcode of MMOVI16 MAR1, #16I)
MSW: 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0

```

Description

Load the auxiliary register, MAR0 or MAR1, with a 16-bit immediate value. Refer to the pipeline section for important information regarding this instruction.

MARx = #16I;

Flags

This instruction does not modify flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction. The immediate load of MAR0 or MAR1 will occur in the EXE phase of the pipeline. Any post increment of MAR0 or MAR1 using indirect addressing will occur in the D2 phase of the pipeline. Therefore the following applies when loading the auxiliary registers:

• I1 and I2

The two instructions following MMOVI16 will use MAR0/MAR1 before the update occurs. Thus these two instructions will use the old value of MAR0 or MAR1.

• I3

Loading of an auxiliary register occurs in the EXE phase while updates due to post-increment addressing occur in the D2 phase. Thus I3 cannot use the auxiliary register or there will be a conflict. In the case of a conflict, the update due to address-mode post increment will win and the auxiliary register will not be updated with #_X.

• I4

Starting with the 4th instruction MAR0 or MAR1 will be the new value loaded with MMOVI16.

; Assume MAR0 is 50 and #_X is 20

```

MMOVI16 MAR0, #_X          ; Load MAR0 with address of X (20)
<Instruction 1>              ; I1 Will use the old value of MAR0 (50)
<Instruction 2>              ; I2 Will use the old value of MAR0 (50)
<Instruction 3>              ; I3 Cannot use MAR0
<Instruction 4>              ; I4 Will use the new value of MAR0 (20)
<Instruction 5>              ; I5
....

```

Table 30. Pipeline Activity For MMOVI16 MAR0/MAR1, #16I

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------------|---------|---------|---------|---------|---------|---------|---------|---|
| MMOVI16 MAR0, #_X | MMOVI16 | | | | | | | |
| I1 | I1 | MMOVI16 | | | | | | |
| I2 | I2 | I1 | MMOVI16 | | | | | |
| I3 | I3 | I2 | I1 | MMOVI16 | | | | |
| I4 | I4 | I3 | I2 | I1 | MMOVI16 | | | |
| I5 | I5 | I4 | I3 | I2 | I1 | MMOVI16 | | |
| I6 | I6 | I5 | I4 | I3 | I2 | I1 | MMOVI16 | |

MMOVI32 MRa, #32FHex *Load the 32-bits of a 32-bit Floating-Point Register with the immediate*

Operands

| | |
|---------|---|
| MRa | floating-point register (MR0 to MR3) |
| #32FHex | A 32-bit immediate value that represents an IEEE 32-bit floating-point value. |

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

```
MMOVIZ MRa, #16FHiHex
MMOVXI MRa, #16FLoHex
```

Opcode

```
LSW: IIII IIII IIII IIII (opcode of MMOVIZ MRa, #16FHiHex)
MSW: 0111 1000 0100 00aa
```

```
LSW: IIII IIII IIII IIII (opcode of MMOVXI MRa, #16FLoHex)
MSW: 0111 1000 1000 00aa
```

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 MMOVF32 MRa, #32F instruction.

Load the 32-bits of MRa with the immediate 32-bit hex value represented by #32FHex.

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

```
MRa = #32FHex;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

Depending on #32FHex, this instruction takes one or two cycles. If all of the lower 16-bits of #32FHex are zeros, then assembler will convert MOV132 to the MMOVIZ instruction. If the lower 16-bits of #32FHex are not zeros, then assembler will convert MOV132 to a MMOVIZ and a MMOVXI instruction.

Example

```
MOV132  MR1, #0x40400000 ; MR1 = 0x40400000
                          ; Assembler converts this instruction as
                          ; MMOVIZ MR1, #0x4040

MOV132  MR2, #0x00000000 ; MR2 = 0x00000000
                          ; Assembler converts this instruction as
                          ; MMOVIZ MR2, #0x0

MOV132  MR3, #0x40004001 ; MR3 = 0x40004001
                          ; Assembler converts this instruction as
                          ; MMOVIZ MR3, #0x4000
                          ; MMOVXI MR3, #0x4001

MOV132  MR0, #0x00004040 ; MR0 = 0x00004040
                          ; Assembler converts this instruction as
                          ; MMOVIZ MR0, #0x0000
                          ; MMOVXI MR0, #0x4040
```

See also

[MMOVIZ MRa, #16FHi](#)
[MMOVXI MRa, #16FLoHex](#)
[MMOVF32 MRa, #32F](#)

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

Operands

| | |
|--------|---|
| MRa | floating-point register (MR0 to MR3) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

```
LSW: I I I I I I I I I I I I I I I I
MSW: 0 1 1 1 1 0 0 0 0 1 0 0 0 0 a a
```

Description

Load the upper 16-bits of MRa with the immediate value #16FHi and clear the low 16-bits of MRa.

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

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

```
MRa(31:16) = #16FHi;
MRa(15:0) = 0;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; Load MR0 and MR1 with -1.5 (0xBFC00000)
MMOVIZ MR0, #0xBFC0 ; MR0 = 0xBFC00000 (1.5)
MMOVIZ MR1, #-1.5 ; MR0 = -1.5 (0xBFC00000)

; Load MR2 with pi = 3.141593 (0x40490FDB)
MMOVIZ MR2, #0x4049 ; MR0 = 0x40490000
MMOVXI MR2, #0x0FDB ; MR0 = 0x40490FDB
```

See also

[MMOVF32 MRa, #32F](#)
[MMOVI32 MRa, #32FHex](#)
[MMOVXI MRa, #16FLoHex](#)

MMOVZ16 MRa, mem16 *Load MRx with 16-bit Value*

Operands

| | |
|-------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| mem16 | 16-bit source memory location |

Opcode

```
LSW: mmmmm mmmmm mmmmm mmmmm
MSW: 0111 0101 10aa addr
```

Description

Move the 16-bit value referenced by mem16 to the floating-point register indicated by MRa.

```
MRa(31:16) = 0;
MRa(15:0) = [mem16];
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = 0;
if (MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

MMOVXI MRa, #16FLoHex *Move Immediate to the Low 16-bits of a Floating-Point Register*

Operands

| | |
|-----------|--|
| MRa | CLA floating-point register (MR0 to MR3) |
| #16FLoHex | A 16-bit immediate hex value that represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits will not be modified. |

Opcode

```
LSW: I I I I I I I I I I I I I I
MSW: 0 1 1 1 1 0 0 0 1 0 0 0 0 0 a a
```

Description

Load the low 16-bits of MRa with the immediate value #16FLoHex. #16FLoHex represents the lower 16-bits of an IEEE 32-bit floating-point value. The upper 16-bits of MRa will not be modified. MMOVXI can be combined with the MMOVIZ instruction to initialize all 32-bits of a MRa register.

```
MRa(15:0) = #16FLoHex;
MRa(31:16) = Unchanged;
```

Flags

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; Load MR0 with pi = 3.141593 (0x40490FDB)
MMOVIZ      MR0, #0x4049    ; MR0 = 0x40490000
MMOVXI      MR0, #0x0FDB    ; MR0 = 0x40490FDB
```

See also

[MMOVIZ MRa, #16FHi](#)

MMPYF32 MRa, MRb, MRc 32-bit Floating-Point Multiply

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| MRc | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 00cc bbaa
MSW: 0111 1100 0000 0000
```

Description

Multiply the contents of two floating-point registers.

MRa = MRb * MRc;

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 generates an underflow condition.
- LVF = 1 if MMPYF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```
; Calculate Num/Den using a Newton-Raphson algorithm for 1/Den
; Ye = Estimate(1/X)
; Ye = Ye*(2.0 - Ye*X)
; Ye = Ye*(2.0 - Ye*X)
;
_ClalTask1:
    MMOV32    MR1, @_Den        ; MR1 = Den
    MEINVF32  MR2, MR1          ; MR2 = Ye = Estimate(1/Den)
    MMPYF32   MR3, MR2, MR1     ; MR3 = Ye*Den
    MSUBF32   MR3, #2.0, MR3    ; MR3 = 2.0 - Ye*Den
    MMPYF32   MR2, MR2, MR3     ; MR2 = Ye = Ye*(2.0 - Ye*Den)
    MMPYF32   MR3, MR2, MR1     ; MR3 = Ye*Den
||
    MMOV32    MR0, @_Num        ; MR0 = Num
    MSUBF32   MR3, #2.0, MR3    ; MR3 = 2.0 - Ye*Den
    MMPYF32   MR2, MR2, MR3     ; MR2 = Ye = Ye*(2.0 - Ye*Den)
||
    MMOV32    MR1, @_Den        ; Reload Den To Set Sign
    MNEGF32   MR0, MR0, EQ      ; if(Den == 0.0) Change Sign Of Num
    MMPYF32   MR0, MR2, MR0     ; MR0 = Y = Ye*Num
    MMOV32    @_Dest, MR0       ; Store result
    MSTOP
```

See also

[MMPYF32 MRa, #16FHi, MRb](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRe, MRf](#)
[MMPYF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)
[MMPYF32 MRd, MRe, MRf || MMOV32 mem32, MRa](#)
[MMPYF32 MRa, MRb, MRc || MSUBF32 MRd, MRe, MRf](#)
[MMACF32 MR3, MR2, MRd, MRe, MRf || MMOV32 MRa, mem32](#)

MMPYF32 MRa, #16FHi, MRb 32-bit Floating-Point Multiply

Operands

| | |
|--------|---|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| #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. |
| MRc | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: I I I I I I I I I I I I I I I I
MSW: 0 1 1 1 0 1 1 1 1 0 0 0 b a a a
```

Description

Multiply MRb with the floating-point value represented by the immediate operand. Store the result of the addition in MRa.

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

```
MRa = MRb * #16FHi:0;
```

This instruction can also be written as MMPYF32 MRa, MRb, #16FHi.

Flags

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

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 generates an underflow condition.
- LVF = 1 if MMPYF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example 1

```
; Same as example 2 but #16FHi is represented in float
MMOVIZ    MR3, #2.0      ; MR3 = 2.0 (0x40000000)
MMPYF32    MR0, #3.0, MR3 ; MR0 = 3.0 * MR3 = 6.0 (0x40C00000)
MMOV32     @_X, MR0      ; Save the result in variable X
```

Example 2

```
; Same as example 1 but #16FHi is represented in Hex
MMOVIZ    MR3, #2.0      ; MR3 = 2.0 (0x40000000)
MMPYF32    MR0, #0x4040, MR3 ; MR0 = 0x4040 * MR3 = 6.0 (0x40C00000)
MMOV32     @_X, MR0      ; Save the result in variable X
```

Example 3

```

; Given X, M and B are IQ24 numbers:
; X = IQ24(+2.5) = 0x02800000
; M = IQ24(+1.5) = 0x01800000
; B = IQ24(-0.5) = 0xFF800000
;
; Calculate Y = X * M + B
;
;
_Cla1Task2:
;
; Convert M, X and B from IQ24 to float
    MI32TOF32    MR0, @_M          ; MR0 = 0x4BC00000
    MI32TOF32    MR1, @_X          ; MR1 = 0x4C200000
    MI32TOF32    MR2, @_B          ; MR2 = 0xCB000000
    MMPYF32      MR0, MR0, #0x3380 ; M = 1/(1*2^24) * iqm = 1.5 (0x3FC00000)
    MMPYF32      MR1, MR1, #0x3380 ; X = 1/(1*2^24) * iqx = 2.5 (0x40200000)
    MMPYF32      MR2, MR2, #0x3380 ; B = 1/(1*2^24) * iqb = -.5 (0xBF000000)
    MMPYF32      MR3, MR0, MR1     ; M*X
    MADDF32      MR2, MR2, MR3     ; Y=MX+B = 3.25 (0x40500000)
; Convert Y from float32 to IQ24
    MMPYF32      MR2, MR2, #0x4B80 ; Y * 1*2^24
    MF32TOI32    MR2, MR2         ; IQ24(Y) = 0x03400000
    MMOV32      @_Y, MR2         ; store result
    MSTOP                          ; end of task

```

See also

[MMPYF32 MRa, MRb, #16FHi](#)
[MMPYF32 MRa, MRb, MRc](#)
[MMPYF32 MRa, MRb, MRc || MADDF32 MRd, MRe, MRf](#)

MMPYF32 MRa, MRb, #16FHi 32-bit Floating-Point Multiply

Operands

| | |
|--------|---|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |

Opcode

```
LSW: I I I I I I I I I I I I I I
MSW: 0 1 1 1 0 1 1 1 1 0 0 0 b a a a
```

Description

Multiply MRb with the floating-point value represented by the immediate operand. Store the result of the addition in MRa.

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

```
MRa = MRb * #16FHi:0;
```

This instruction can also be written as MMPYF32 MRa, #16FHi, MRb.

Flags

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

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 generates an underflow condition.
- LVF = 1 if MMPYF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example 1

```
;Same as example 2 but #16FHi is represented in float
MMOVIZ MR3, #2.0 ; MR3 = 2.0 (0x40000000)
MMPYF32 MR0, MR3, #3.0 ; MR0 = MR3 * 3.0 = 6.0 (0x40C00000)
MMOV32 @_X, MR0 ; Save the result in variable X
```

Example 2

```
;Same as above example but #16FHi is represented in Hex
MMOVIZ MR3, #2.0 ; MR3 = 2.0 (0x40000000)
MMPYF32 MR0, MR3, #0x4040 ; MR0 = MR3 * 0x4040 = 6.0 (0x40C00000)
MMOV32 @_X, MR0 ; Save the result in variable X
```

Example 3

```

; Given X, M and B are IQ24 numbers:
; X = IQ24(+2.5) = 0x02800000
; M = IQ24(+1.5) = 0x01800000
; B = IQ24(-0.5) = 0xFF800000
;
; Calculate Y = X * M + B
;
_Cla1Task2:
;
; Convert M, X and B from IQ24 to float
    MI32TOF32    MR0, @_M          ; MR0 = 0x4BC00000
    MI32TOF32    MR1, @_X          ; MR1 = 0x4C200000
    MI32TOF32    MR2, @_B          ; MR2 = 0xCB000000
    MMPYF32      MR0, #0x3380, MR0 ; M = 1/(1*2^24) * iqm = 1.5 (0x3FC00000)
    MMPYF32      MR1, #0x3380, MR1 ; X = 1/(1*2^24) * iqx = 2.5 (0x40200000)
    MMPYF32      MR2, #0x3380, MR2 ; B = 1/(1*2^24) * iqb = -.5 (0xBF000000)
    MMPYF32      MR3, MR0, MR1      ; M*X
    MADDF32      MR2, MR2, MR3      ; Y=MX+B = 3.25 (0x40500000)

; Convert Y from float32 to IQ24
    MMPYF32      MR2, #0x4B80, MR2 ; Y * 1*2^24
    MF32TOI32    MR2, MR2          ; IQ24(Y) = 0x03400000
    MMOV32       @_Y, MR2          ; store result
    MSTOP
; end of task

```

See also

[MMPYF32 MRa, #16FHi, MRb](#)
[MMPYF32 MRa, MRb, MRc](#)

MMPYF32 MRa, MRb, MRc||MADDF32 MRd, MRe, MRf 32-bit Floating-Point Multiply with Parallel Add

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register for MMPYF32 (MR0 to MR3) MRa cannot be the same register as MRd |
| MRb | CLA floating-point source register for MMPYF32 (MR0 to MR3) |
| MRc | CLA floating-point source register for MMPYF32 (MR0 to MR3) |
| MRd | CLA floating-point destination register for MADDF32 (MR0 to MR3) MRd cannot be the same register as MRa |
| MRe | CLA floating-point source register for MADDF32 (MR0 to MR3) |
| MRf | CLA floating-point source register for MADDF32 (MR0 to MR3) |

Opcode

LSW: 0000 ffee ddcc bbaa
MSW: 0111 1010 0000 0000

Description

Multiply the contents of two floating-point registers with parallel addition of two registers.

MRa = MRb * MRc;
MRd = MRe + MRf;

Restrictions

The destination register for the MMPYF32 and the MADDF32 must be unique. That is, MRa cannot be the same register as MRd.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 or MADDF32 generates an underflow condition.
- LVF = 1 if MMPYF32 or MADDF32 generates an overflow condition.

Pipeline

Both MMPYF32 and MADDF32 complete in a single cycle.

Example

```

; Perform 5 multiply and accumulate operations:
;
; X and Y are 32-bit floating point arrays
;
; 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
;
_ClalTask1:
    MMOVI16    MAR0, #_X                ; MAR0 points to X array
    MMOVI16    MAR1, #_Y                ; MAR1 points to Y array
    MNOP                          ; Delay for MAR0, MAR1 load
    MNOP                          ; Delay for MAR0, MAR1 load
                                ; <-- MAR0 valid
    MMOV32     MR0, *MAR0[2]++          ; MR0 = X0, MAR0 += 2
                                ; <-- MAR1 valid
    MMOV32     MR1, *MAR1[2]++          ; MR1 = Y0, MAR1 += 2

    MMPYF32    MR2, MR0, MR1            ; MR2 = A = X0 * Y0
    || MMOV32   MR0, *MAR0[2]++          ; In parallel MR0 = X1, MAR0 += 2
    MMOV32     MR1, *MAR1[2]++          ; MR1 = Y1, MAR1 += 2

    MMPYF32    MR3, MR0, MR1            ; MR3 = B = X1 * Y1
    || MMOV32   MR0, *MAR0[2]++          ; In parallel MR0 = X2, MAR0 += 2
    MMOV32     MR1, *MAR1[2]++          ; MR1 = Y2, MAR2 += 2

    MMACF32    MR3, MR2, MR2, MR0, MR1 ; MR3 = A + B, MR2 = C = X2 * Y2
    || MMOV32   MR0, *MAR0[2]++          ; In parallel MR0 = X3
    MMOV32     MR1, *MAR1[2]++          ; MR1 = Y3

    MMACF32    MR3, MR2, MR2, MR0, MR1 ; MR3 = (A + B) + C, MR2 = D = X3 * Y3
    || MMOV32   MR0, *MAR0              ; In parallel MR0 = X4
    MMOV32     MR1, *MAR1              ; MR1 = Y4

    MMPYF32    MR2, MR0, MR1            ; MR2 = E = X4 * Y4
    || MADDF32  MR3, MR3, MR2            ; in parallel MR3 = (A + B + C) + D

    MADDF32    MR3, MR3, MR2            ; MR3 = (A + B + C + D) + E
    MMOV32     @_Result, MR3            ; Store the result
    MSTOP                          ; end of task

```

See also

[MMACF32 MR3, MR2, MRd, MRc, MRf || MMOV32 MRa, mem32](#)

MMPYF32 MRd, MRe, MRf ||MMOV32 MRa, mem32 32-bit Floating-Point Multiply with Parallel Move

Operands

| | |
|-------|--|
| MRd | CLA floating-point destination register for the MMPYF32 (MR0 to MR3) MRd cannot be the same register as MRa |
| MRe | CLA floating-point source register for the MMPYF32 (MR0 to MR3) |
| MRf | CLA floating-point source register for the MMPYF32 (MR0 to MR3) |
| MRa | CLA floating-point destination register for the MMOV32 (MR0 to MR3) MRa cannot be the same register as MRd |
| mem32 | 32-bit memory location accessed using direct or indirect addressing. This will be the source of the MMOV32. |

Opcode

```
LSW: mmmn mmmn mmmn mmmn
MSW: 0000 ffee ddaa addr
```

Description

Multiply the contents of two floating-point registers and load another.

```
MRd = MRe * MRf;
MRa = [mem32];
```

Restrictions

The destination register for the MMPYF32 and the MMOV32 must be unique. That is, MRa cannot be the same register as MRd.

Flags

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

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 generates an underflow condition.
- LVF = 1 if MMPYF32 generates an overflow condition.

The MMOV32 Instruction will set the NF and ZF flags as follows:

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

Pipeline

Both MMPYF32 and MMOV32 complete in a single cycle.

Example 1

```
; Given M1, X1 and B1 are 32-bit floating point
; Calculate Y1 = M1*X1+B1
;
_ClalTask1:
    MMOV32    MR0, @M1        ; Load MR0 with M1
    MMOV32    MR1, @X1        ; Load MR1 with X1
    MMPYF32   MR1, MR1, MR0    ; Multiply M1*X1
    || MMOV32  MR0, @B1        ; and in parallel load MR0 with B1
    MADDF32   MR1, MR1, MR0    ; Add M*X1 to B1 and store in MR1
    MMOV32    @Y1, MR1        ; Store the result
    MSTOP
```

Example 2

```

; Given A, B and C are 32-bit floating-point numbers
; Calculate Y2 = (A * B)
;           Y3 = (A * B) * C
;
_ClalTask2:
    MMOV32    MR0, @A        ; Load MR0 with A
    MMOV32    MR1, @B        ; Load MR1 with B
    MMPYF32   MR1, MR1, MR0   ; Multiply A*B
    || MMOV32  MR0, @C        ; and in parallel load MR0 with C
    MMPYF32   MR1, MR1, MR0   ; Multiply (A*B) by C
    || MMOV32  @Y2, MR1       ; and in parallel store A*B
    MMOV32    @Y3, MR1       ; Store the result
    MSTOP                                           ; end of task

```

See also

[MMPYF32 MRd, MRe, MRf || MMOV32 mem32, MRa](#)
[MMACF32 MR3, MR2, MRd, MRe, MRf || MMOV32 MRa, mem32](#)

MMPYF32 MRd, MRe, MRf ||MMOV32 mem32, MRa 32-bit Floating-Point Multiply with Parallel Move

Operands

| | |
|-------|--|
| MRd | CLA floating-point destination register for the MMPYF32 (MR0 to MR3) |
| MRe | CLA floating-point source register for the MMPYF32 (MR0 to MR3) |
| MRf | CLA floating-point source register for the MMPYF32 (MR0 to MR3) |
| mem32 | 32-bit memory location accessed using direct or indirect addressing. This will be the destination of the MMOV32. |
| MRa | CLA floating-point source register for the MMOV32 (MR0 to MR3) |

Opcode

```
LSW: mmmm mmmm mmmm mmmm
MSW: 0100 ffee ddaa addr
```

Description

Multiply the contents of two floating-point registers and move from memory to register.

```
MRd = MRe * MRf;
[mem32] = MRa;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 generates an underflow condition.
- LVF = 1 if MMPYF32 generates an overflow condition.

Pipeline

MMPYF32 and MMOV32 both complete in a single cycle.

Example

```
; Given A, B and C are 32-bit floating-point numbers
; Calculate Y2 = (A * B)
;           Y3 = (A * B) * C
;
_ClalTask2:
    MMOV32    MR0, @A        ; Load MR0 with A
    MMOV32    MR1, @B        ; Load MR1 with B
    MMPYF32   MR1, MR1, MR0   ; Multiply A*B
    || MMOV32  MR0, @C        ; and in parallel load MR0 with C
    MMPYF32   MR1, MR1, MR0   ; Multiply (A*B) by C
    || MMOV32  @Y2, MR1       ; and in parallel store A*B
    MMOV32    @Y3, MR1       ; Store the result
    MSTOP
```

See also

[MMPYF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)
[MMACF32 MR3, MR2, MRd, MRe, MRf || MMOV32 MRa, mem32](#)

MMPYF32 MRa, MRb, MRc || MSUBF32 MRd, MRe, MRf 32-bit Floating-Point Multiply with Parallel Subtract

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register for MMPYF32 (MR0 to MR3) MRa cannot be the same register as MRd |
| MRb | CLA floating-point source register for MMPYF32 (MR0 to MR3) |
| MRc | CLA floating-point source register for MMPYF32 (MR0 to MR3) |
| MRd | CLA floating-point destination register for MSUBF32 (MR0 to MR3) MRd cannot be the same register as MRa |
| MRe | CLA floating-point source register for MSUBF32 (MR0 to MR3) |
| MRf | CLA floating-point source register for MSUBF32 (MR0 to MR3) |

Opcode

```
LSW: 0000 ffee ddcc bbaa
MSW: 0111 1010 0100 0000
```

Description

Multiply the contents of two floating-point registers with parallel subtraction of two registers.

```
MRa = MRb * MRc;
MRd = MRe - MRf;
```

Restrictions

The destination register for the MMPYF32 and the MSUBF32 must be unique. That is, MRa cannot be the same register as MRd.

Flags

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

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MMPYF32 or MSUBF32 generates an underflow condition.
- LVF = 1 if MMPYF32 or MSUBF32 generates an overflow condition.

Pipeline

MMPYF32 and MSUBF32 both complete in a single cycle.

Example

```
; Given A, B and C are 32-bit floating-point numbers
; Calculate Y2 = (A * B)
;           Y3 = (A - B)
;
_ClalTask2:
    MMOV32    MR0, @A           ; Load MR0 with A
    MMOV32    MR1, @B           ; Load MR1 with B
    MMPYF32   MR2, MR0, MR1     ; Multiply (A*B)
    || MSUBF32 MR3, MR0, MR1     ; and in parallel Sub (A-B)
    MMOV32    @Y2, MR2          ; Store A*B
    MMOV32    @Y3, MR3          ; Store A-B
    MSTOP                     ; end of task
```

See also

[MSUBF32 MRa, MRb, MRc](#)
[MSUBF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)
[MSUBF32 MRd, MRe, MRf || MMOV32 mem32, MRa](#)

MNEGF32 MRa, MRb{, CNDF} *Conditional Negation*

Operands

| | |
|------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| CNDF | condition tested |

Opcode

```
LSW: 0000 0000 cndf bbaa
MSW: 0111 1010 1000 0000
```

Description

```
if (CNDF == true) {MRa = - MRb; }
else {MRa = MRb; }
```

CNDF is one of the following conditions:

| Encode ⁽⁵⁾ | CNDF | Description | MSTF 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 OR 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 ⁽⁶⁾ | Unconditional with flag modification | None |

⁽⁵⁾ Values not shown are reserved.

⁽⁶⁾ This is the default operation if no CNDF field is specified. This condition will allow the ZF, and NF 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

Pipeline

This is a single-cycle instruction.

Example 1

```
; Show the basic operation of MNEGF32
;
MMOVIZ    MR0, #5.0      ; MR0 = 5.0 (0x40A00000)
MMOVIZ    MR1, #4.0      ; MR1 = 4.0 (0x40800000)
MMOVIZ    MR2, #-1.5     ; MR2 = -1.5 (0xBFC00000)
MMPYF32   MR3, MR1, MR2  ; MR3 = -6.0
MMPYF32   MR0, MR0, MR1  ; MR0 = 20.0
MMOVIZ    MR1, #0.0
MCMPPF32  MR3, MR1       ; NF = 1
MNEGF32   MR3, MR3, LT   ; if NF = 1, MR3 = 6.0
MCMPPF32  MR0, MR1       ; NF = 0
MNEGF32   MR0, MR0, GEQ  ; if NF = 0, MR0 = -20.0
```

Example 2

```

; Calculate Num/Den using a Newton-Raphson algorithm for 1/Den
; Ye = Estimate(1/X)
; Ye = Ye*(2.0 - Ye*X)
; Ye = Ye*(2.0 - Ye*X)
;
_ClalTask1:
    MMOV32    MR1, @_Den      ; MR1 = Den
    MEINVF32  MR2, MR1       ; MR2 = Ye = Estimate(1/Den)
    MMPYF32   MR3, MR2, MR1  ; MR3 = Ye*Den
    MSUBF32   MR3, #2.0, MR3 ; MR3 = 2.0 - Ye*Den
    MMPYF32   MR2, MR2, MR3  ; MR2 = Ye = Ye*(2.0 - Ye*Den)
    MMPYF32   MR3, MR2, MR1  ; MR3 = Ye*Den
    || MMOV32  MR0, @_Num     ; MR0 = Num
    MSUBF32   MR3, #2.0, MR3 ; MR3 = 2.0 - Ye*Den
    MMPYF32   MR2, MR2, MR3  ; MR2 = Ye = Ye*(2.0 - Ye*Den)
    || MMOV32  MR1, @_Den     ; Reload Den To Set Sign
    MNEGF32   MR0, MR0, EQ    ; if(Den == 0.0) Change Sign Of Num
    MMPYF32   MR0, MR2, MR0  ; MR0 = Y = Ye*Num
    MMOV32    @_Dest, MR0    ; Store result
    MSTOP
    ; end of task

```

See also
[MABSF32 MRa, MRb](#)

MNOP *No Operation*

Operands

| | |
|------|---|
| none | This instruction does not have any operands |
|------|---|

Opcode

```
LSW: 0000 0000 0000 0000
MSW: 0111 1111 1010 0000
```

Description

Do nothing. This instruction is used to fill required pipeline delay slots when other instructions are not available to fill the slots.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; X is an array of 32-bit floating-point values
; Find the maximum value in an array X
; and store it in Result
;
_ClalTask1:
    MMOV16    MAR1, #_X          ; Start address
    MUI16TOF32 MR0, @_len        ; Length of the array
    MNOP                      ; delay for MAR1 load
    MNOP                      ; delay for MAR1 load
    MMOV32    MR1, *MAR1[2]++    ; MR1 = X0
LOOP
    MMOV32    MR2, *MAR1[2]++    ; MR2 = next element
    MMAXF32   MR1, MR2           ; MR1 = MAX(MR1, MR2)
    MADD32    MR0, MR0, #-1.0    ; Decrement the counter
    MCMPF32   MR0 #0.0           ; Set/clear flags for MBCNDD
    MNOP                      ; Too late to affect MBCNDD
    MNOP                      ; Too late to affect MBCNDD
    MNOP                      ; Too late to affect MBCNDD
    MBCNDD    LOOP, NEQ          ; Branch if not equal to zero
    MMOV32    @_Result, MR1      ; Always executed
    MNOP                      ; Pad to separate MBCNDD and MSTOP
    MNOP                      ; Pad to separate MBCNDD and MSTOP
    MSTOP                      ; End of task
```

See also

MOR32 MRa, MRb, MRc *Bitwise OR*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| MRc | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 00cc bbaa
MSW: 0111 1100 1000 0000
```

Description

Bitwise OR of MRb with MRc.

```
MARa(31:0) = MARb(31:0) OR MRc(31:0);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ    MR0, #0x5555 ; MR0 = 0x5555AAAA
MMOVXI    MR0, #0xAAAA

MMOVIZ    MR1, #0x5432 ; MR1 = 0x5432FEDC
MMOVXI    MR1, #0xFEDC

; 0101 OR 0101 = 0101 (5)
; 0101 OR 0100 = 0101 (5)
; 0101 OR 0011 = 0111 (7)
; 0101 OR 0010 = 0111 (7)
; 1010 OR 1111 = 1111 (F)
; 1010 OR 1110 = 1110 (E)
; 1010 OR 1101 = 1111 (F)
; 1010 OR 1100 = 1110 (E)
MOR32 MR2, MR1, MR0 ; MR3 = 0x5555FEFE
```

See also

[MAND32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)

MRCNDD {CNDF} *Return Conditional Delayed*
Operands

| | |
|------|---------------------|
| CNDF | optional condition. |
|------|---------------------|

Opcode

```
LSW: 0000 0000 0000 0000
MSW: 0111 1001 1010 cndf
```

Description

If the specified condition is true, then the RPC field of MSTF is loaded into MPC and fetching continues from that location. Otherwise program fetches will continue without the return.

Please refer to the pipeline section for important information regarding this instruction.

```
if (CNDF == TRUE) MPC = RPC;
```

CNDF is one of the following conditions:

| Encode ⁽⁷⁾ | CNDF | Description | MSTF 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 OR 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 ⁽⁸⁾ | Unconditional with flag modification | None |

⁽⁷⁾ Values not shown are reserved.

⁽⁸⁾ This is the default operation if no CNDF field is specified. This condition will allow the ZF and NF flags to be modified when a conditional operation is executed. All other conditions will not modify these flags.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

The MRCNDD instruction by itself is a single-cycle instruction. As shown in [Table 31](#), for each return 6 instruction slots are executed; three before the return instruction (d5-d7) and three after the return instruction (d8-d10). The total number of cycles for a return taken or not taken depends on the usage of these slots. That is, the number of cycles depends on how many slots are filled with a MNOP as well as which slots are filled. The effective number of cycles for a return can, therefore, range from 1 to 7 cycles. The number of cycles for a return taken may not be the same as for a return not taken.

Referring to the following code fragment and the pipeline diagrams in [Table 31](#) and [Table 32](#), the instructions before and after MRCNDD have the following properties:

```

;
;
<Instruction 1>      ; I1 Last instruction that can affect flags for
                    ; the MCCNDD operation
<Instruction 2>      ; I2 Cannot be stop, branch, call or return
<Instruction 3>      ; I3 Cannot be stop, branch, call or return
<Instruction 4>      ; I4 Cannot be stop, branch, call or return

MCCNDD _func, NEQ    ; Call to func if not equal to zero
                    ; Three instructions after MCCNDD are always
                    ; executed whether the call is taken or not
<Instruction 5>      ; I5 Cannot be stop, branch, call or return
<Instruction 6>      ; I6 Cannot be stop, branch, call or return
<Instruction 7>      ; I7 Cannot be stop, branch, call or return
<Instruction 8>      ; I8 The address of this instruction is saved
                    ; in the RPC field of the MSTF register.
                    ; Upon return this value is loaded into MPC
                    ; and fetching continues from this point.
<Instruction 9>      ; I9
<Instruction 10>     ; I10
....
....
_func:
<Destination 1>     ; d1 Can be any instruction
<Destination 2>     ; d2
<Destination 3>     ; d3
<Destination 4>     ; d4 Last instruction that can affect flags for
                    ; the MRCNDD operation
<Destination 5>     ; d5 Cannot be stop, branch, call or return
<Destination 6>     ; d6 Cannot be stop, branch, call or return
<Destination 7>     ; d7 Cannot be stop, branch, call or return

MRCNDD, NEQ          ; Return to <Instruction 8> if not equal to zero
                    ; Three instructions after MRCNDD are always
                    ; executed whether the return is taken or not
<Destination 8>     ; d8 Cannot be stop, branch, call or return
<Destination 9>     ; d9 Cannot be stop, branch, call or return
<Destination 10>    ; d10 Cannot be stop, branch, call or return
<Destination 11>    ; d11
<Destination 12>    ; d12
....
....
MSTOP
....

```

- **d4**

- d4 is the last instruction that can effect the CNDF flags for the MRCNDD instruction. The CNDF flags are tested in the D2 phase of the pipeline. That is, a decision is made whether to return or not when MRCNDD is in the D2 phase.
- There are no restrictions on the type of instruction for d4.

- **d5, d6 and d7**

- The three instructions proceeding MRCNDD can change MSTF flags but will have no effect on whether the MRCNDD instruction makes the return or not. This is because the flag modification will occur after the D2 phase of the MRCNDD instruction.
- These instructions must not be the following: MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.

- **d8, d9 and d10**
 - The three instructions following MRCNDD are always executed irrespective of whether the return is taken or not.
 - These instructions must not be the following: MSTOP, MDEBUGSTOP, MBCNDD, MCCNDD or MRCNDD.

Table 31. Pipeline Activity For MRCNDD, Return Not Taken

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------|--------|--------|--------|--------|-----|-----|-----|---|
| d4 | d4 | d3 | d2 | d1 | I7 | I6 | I5 | |
| d5 | d5 | d4 | d3 | d2 | d1 | I7 | I6 | |
| d6 | d6 | d5 | d4 | d3 | d2 | d1 | I7 | |
| d7 | d7 | d6 | d5 | d4 | d3 | d2 | d1 | |
| MRCNDD | MRCNDD | d7 | d6 | d5 | d4 | d3 | d2 | |
| d8 | d8 | MRCNDD | d7 | d6 | d5 | d4 | d3 | |
| d9 | d9 | d8 | MRCNDD | d7 | d6 | d5 | d4 | |
| d10 | d10 | d9 | d8 | MRCNDD | d7 | d6 | d5 | |
| d11 | d11 | d10 | d9 | d8 | - | d7 | d6 | |
| d12 | d12 | d11 | d10 | d9 | d8 | - | d7 | |
| etc.... | | d12 | d11 | d10 | d9 | d8 | - | |
| | | | d12 | d11 | d10 | d9 | d8 | |
| | | | | d12 | d11 | d10 | d9 | |
| | | | | | d12 | d11 | d10 | |
| | | | | | | d12 | d11 | |
| | | | | | | | d12 | |

Table 32. Pipeline Activity For MRCNDD, Return Taken

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|-------------|--------|--------|--------|--------|-----|-----|-----|---|
| d4 | d4 | d3 | d2 | d1 | I7 | I6 | I5 | |
| d5 | d5 | d4 | d3 | d2 | d1 | I7 | I6 | |
| d6 | d6 | d5 | d4 | d3 | d2 | d1 | I7 | |
| d7 | d7 | d6 | d5 | d4 | d3 | d2 | d1 | |
| MRCNDD | MRCNDD | d7 | d6 | d5 | d4 | d3 | d2 | |
| d8 | d8 | MRCNDD | d7 | d6 | d5 | d4 | d3 | |
| d9 | d9 | d8 | MRCNDD | d7 | d6 | d5 | d4 | |
| d10 | d10 | d9 | d8 | MRCNDD | d7 | d6 | d5 | |
| I8 | I8 | d10 | d9 | d8 | - | d7 | d6 | |
| I9 | I9 | I8 | d10 | d9 | d8 | - | d7 | |
| I10 | I10 | I9 | I8 | d10 | d9 | d8 | - | |
| etc.... | | I10 | I9 | I8 | d10 | d9 | d8 | |
| | | | I10 | I9 | I8 | d10 | d9 | |
| | | | | I10 | I9 | I8 | d10 | |
| | | | | | I10 | I9 | I8 | |
| | | | | | | I10 | I9 | |
| | | | | | | | I10 | |

Example ;

See also [MBCNDD #16BitDest, CNDF](#)
[MCCNDD 16BitDest, CNDF](#)
[MMOV32 mem32, MSTF](#)
[MMOV32 MSTF, mem32](#)

MSETFLG FLAG, VALUE *Set or clear selected floating-point status flags*

Operands

| | |
|-------|--|
| FLAG | 8 bit mask indicating which floating-point status flags to change. |
| VALUE | 8 bit mask indicating the flag value; 0 or 1. |

Opcode

LSW: FFFF FFFF VVVV VVVV
MSW: 0111 1001 1100 0000

Description

The MSETFLG instruction is used to set or clear selected floating-point status flags in the MSTF 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:

| reserved | RNDF32 | TF | reserved | reserved | ZF | NF | LUF | LVF |
|----------|--------|----|----------|----------|----|----|-----|-----|
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

The VALUE field indicates the value the flag should be set to; 0 or 1.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|-----|-----|-----|-----|-----|
| Modified | Yes | Yes | Yes | Yes | Yes |

Any flag can be modified by this instruction. The MEALLOW and RPC fields cannot be modified with this instruction.

Pipeline

This is a single-cycle instruction.

Example

To make it easier and legible, the assembler accepts a FLAG=VALUE syntax for the MSETFLG operation as shown below:

```
MSETFLG RNDF32=0, TF=0, NF=1; FLAG = 11000100; VALUE = 00XXX1XX;
```

See also

[MMOV32 mem32, MSTF](#)
[MMOV32 MSTF, mem32](#)

MSTOP

Stop Task

Operands

| | |
|------|---|
| none | This instruction does not have any operands |
|------|---|

Opcode

```
LSW: 0000 0000 0000 0000
MSW: 0111 1111 1000 0000
```

Description

The MSTOP instruction must be placed to indicate the end of each task. In addition, placing MSTOP in unused memory locations within the CLA program RAM can be useful for debugging and preventing run away CLA code. When MSTOP enters the D2 phase of the pipeline, the MIRUN flag for the task is cleared and the associated interrupt is flagged in the PIE vector table.

There are three special cases that can occur when single-stepping a task such that the MPC reaches the MSTOP instruction.

1. If you are single-stepping or halted in "task A" and "task B" comes in before the MPC reaches the MSTOP, then "task B" will start if you continue to step through the MSTOP instruction. Basically if "task B" is pending before the MPC reaches MSTOP in "task A" then there is no issue in "task B" starting and no special action is required.
2. In this case you have single-stepped or halted in "task A" and the MPC has reached the MSTOP with no tasks pending. If "task B" comes in at this point, it will be flagged in the MIFR register but it may or may not start if you continue to single-step through the MSTOP instruction of "task A". It depends on exactly when the new task comes in. To reliably start "task B" perform a soft reset and reconfigure the MIER bits. Once this is done, you can start single-stepping "task B".
3. Case 2 can be handled slightly differently if there is control over when "task B" comes in (for example using the IACK instruction to start the task). In this case you have single-stepped or halted in "task A" and the MPC has reached the MSTOP with no tasks pending. Before forcing "task B", run free to force the CLA out of the debug state. Once this is done you can force "task B" and continue debugging.

Restrictions

The MSTOP instruction cannot be placed 3 instructions before or after a [MBCNDD](#), [MCCNDD](#) or [MRCNDD](#) instruction.

Flags

This instruction does not modify flags in the MSTF register.

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction. [Table 33](#) shows the pipeline behavior of the MSTOP instruction. The MSTOP instruction cannot be placed with 3 instructions of a [MBCNDD](#), [MCCNDD](#) or [MRCNDD](#) instruction.

Table 33. Pipeline Activity For MSTOP

| Instruction | F1 | F2 | D1 | D2 | R1 | R2 | E | W |
|------------------------------------|-------|-------|-------|-------|----|----|----|---|
| I1 | I1 | | | | | | | |
| I2 | I2 | I1 | | | | | | |
| I3 | I3 | I2 | I1 | | | | | |
| MSTOP | MSTOP | I3 | I2 | I1 | | | | |
| I4 | I4 | MSTOP | I3 | I2 | I1 | | | |
| I5 | I5 | I4 | MSTOP | I3 | I2 | I1 | | |
| I6 | I6 | I5 | I4 | MSTOP | I3 | I2 | I1 | |
| New Task Arbitrated and Piroitized | - | - | - | - | - | I3 | I2 | |
| New Task Arbitrated and Piroitized | - | - | - | - | - | - | I3 | |
| I1 | I1 | - | - | - | - | - | - | - |
| I2 | I2 | I1 | - | - | - | - | - | - |
| I3 | I3 | I2 | I1 | - | - | - | - | - |
| I4 | I4 | I3 | I2 | I1 | - | - | - | - |
| I5 | I5 | I4 | I3 | I2 | I1 | - | - | - |
| I6 | I6 | I5 | I4 | I3 | I2 | I1 | - | - |
| I7 | I7 | I6 | I5 | I4 | I3 | I2 | I1 | - |
| etc | | | | | | | | |

Example

```

; Given A = (int32)1
;      B = (int32)2
;      C = (int32)-7
;
; Calculate Y2 = A - B - C
_Cla1Task3:
    MMOV32    MR0, @_A      ; MR0 = 1 (0x00000001)
    MMOV32    MR1, @_B      ; MR1 = 2 (0x00000002)
    MMOV32    MR2, @_C      ; MR2 = -7 (0xFFFFFFFF9)
    MSUB32    MR3, MR0, MR1 ; A + B
    MSUB32    MR3, MR3, MR2 ; A + B + C = 6 (0x00000006)
    MMOV32    @_y2, MR3     ; Store y2
    MSTOP                     ; End of task

```

See also
[MDEBUGSTOP](#)

MSUB32 MRa, MRb, MRc 32-bit Integer Subtraction

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point destination register (MR0 to MR3) |
| MRc | CLA floating-point destination register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 00cc bbaa
MSW: 0111 1100 1110 0000
```

Description

32-bit integer addition of MRb and MRc.

```
MARa(31:0) = MARb(31:0) - MRc(31:0);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified as follows:

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
; Given A = (int32)1
;         B = (int32)2
;         C = (int32)-7
;
;
Calculate Y2 = A - B - C
;
_ClalTask3:
    MOV32    MR0, @_A        ; MR0 = 1 (0x00000001)
    MOV32    MR1, @_B        ; MR1 = 2 (0x00000002)
    MOV32    MR2, @_C        ; MR2 = -7 (0xFFFFFFFF9)
    MSUB32    MR3, MR0, MR1   ; A + B
    MSUB32    MR3, MR3, MR2   ; A + B + C = 6 (0x00000006)
    MOV32    @_y2, MR3       ; Store y2
    MSTOP                                ; End of task
```

See also

[MADD32 MRa, MRb, MRc](#)
[MAND32 MRa, MRb, MRc](#)
[MASR32 MRa, #SHIFT](#)
[MLSL32 MRa, #SHIFT](#)
[MLSR32 MRa, #SHIFT](#)
[MOR32 MRa, MRb, MRc](#)
[MXOR32 MRa, MRb, MRc](#)

MSUBF32 MRa, MRb, MRc 32-bit Floating-Point Subtraction

Operands

| | |
|-----|---|
| MRa | CLA floating-point destination register (MR0 to R1) |
| MRb | CLA floating-point source register (MR0 to R1) |
| MRc | CLA floating-point source register (MR0 to R1) |

Opcode

```
LSW: 0000 0000 00cc bbaa
MSW: 0111 1100 0100 0000
```

Description

Subtract the contents of two floating-point registers

MRa = MRb - MRc;

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MSUBF32 generates an underflow condition.
- LVF = 1 if MSUBF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```
; Given A, B and C are 32-bit floating-point numbers
; Calculate Y2 = A + B - C
;
_ClalTask5:
    MMOV32    MR0, @_A      ; Load MR0 with A
    MMOV32    MR1, @_B      ; Load MR1 with B
    MADDF32   MR0, MR1, MR0 ; Add A + B
    || MMOV32  MR1, @_C      ; and in parallel load C
    MSUBF32   MR0, MR0, MR1 ; Subtract C from (A + B)
    MMOV32    @Y, MR0       ; (A+B) - C
    MSTOP                                ; end of task
```

See also

MSUBF32 MRa, #16FHi, MRb
MSUBF32 MRd, MRe, MRf || MMOV32 MRa, mem32
MSUBF32 MRd, MRe, MRf || MMOV32 mem32, MRa
MMPYF32 MRa, MRb, MRc || MSUBF32 MRd, MRe, MRf

MSUBF32 MRa, #16FHi, MRb 32-bit Floating Point Subtraction

Operands

| | |
|--------|---|
| MRa | CLA floating-point destination register (MR0 to R1) |
| #16FHi | A 16-bit immediate value that represents the upper 16-bits of an IEEE 32-bit floating-point value. The low 16-bits of the mantissa are assumed to be all 0. |
| MRb | CLA floating-point source register (MR0 to R1) |

Opcode

```
LSW: I I I I I I I I I I I I I I I I
MSW: 0 1 1 1 1 0 0 0 0 0 0 0 b a a a
```

Description

Subtract MRb from the floating-point value represented by the immediate operand. Store the result of the addition in MRa.

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

```
MRa = #16FHi:0 - MRb;
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MSUBF32 generates an underflow condition.
- LVF = 1 if MSUBF32 generates an overflow condition.

Pipeline

This is a single-cycle instruction.

Example

```
; 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
;
_ClalTask3:
    MOV32    MR0, @_x          ; MR0 = X
    MEISQRTF32 MR1, MR0        ; MR1 = Ye = Estimate(1/sqrt(X))
    MOV32    MR1, @_x, EQ       ; if(X == 0.0) Ye = 0.0
    MPYF32   MR3, MR0, #0.5     ; MR3 = X*0.5
    MPYF32   MR2, MR1, MR3      ; MR2 = Ye*X*0.5
    MPYF32   MR2, MR1, MR2      ; MR2 = Ye*Ye*X*0.5
    MSUBF32  MR2, #1.5, MR2     ; MR2 = 1.5 - Ye*Ye*X*0.5
    MPYF32   MR1, MR1, MR2      ; MR1 = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    MPYF32   MR2, MR1, MR3      ; MR2 = Ye*X*0.5
    MPYF32   MR2, MR1, MR2      ; MR2 = Ye*Ye*X*0.5
    MSUBF32  MR2, #1.5, MR2     ; MR2 = 1.5 - Ye*Ye*X*0.5
    MPYF32   MR1, MR1, MR2      ; MR1 = Ye = Ye*(1.5 - Ye*Ye*X*0.5)
    MPYF32   MR0, MR1, MR0      ; MR0 = Y = Ye*X
    MOV32    @_y, MR0          ; Store Y = sqrt(X)
    MSTOP
```

See also

[MSUBF32 MRa, MRb, MRc](#)
[MSUBF32 MRd, MRe, MRf || MOV32 MRa, mem32](#)
[MSUBF32 MRd, MRe, MRf || MOV32 mem32, MRa](#)
[MPYF32 MRa, MRb, MRc || MSUBF32 MRd, MRe, MRf](#)

MSUBF32 MRd, MRe, MRf ||MMOV32 MRa, mem32 *32-bit Floating-Point Subtraction with Parallel Move*

Operands

| | |
|-------|--|
| MRd | CLA floating-point destination register (MR0 to MR3) for the MSUBF32 operation MRd cannot be the same register as MRa |
| MRe | CLA floating-point source register (MR0 to MR3) for the MSUBF32 operation |
| MRf | CLA floating-point source register (MR0 to MR3) for the MSUBF32 operation |
| MRa | CLA floating-point destination register (MR0 to MR3) for the MMOV32 operation MRa cannot be the same register as MRd |
| mem32 | 32-bit memory location accessed using direct or indirect addressing. Source for the MMOV32 operation. |

Opcode

LSW: mmm mmm mmm mmm
MSW: 0010 ffee ddaa addr

Description

Subtract the contents of two floating-point registers and move from memory to a floating-point register.

MRd = MRe - MRf;
MRa = [mem32];

Restrictions

The destination register for the MSUBF32 and the MMOV32 must be unique. That is, MRa cannot be the same register as MRd.

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MSUBF32 generates an underflow condition.
- LVF = 1 if MSUBF32 generates an overflow condition.

The MMOV32 Instruction will set the NF and ZF flags as follows:

Pipeline

Both MSUBF32 and MMOV32 complete in a single cycle.

Example

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

See also

[MSUBF32 MRa, MRb, MRc](#)
[MSUBF32 MRa, #16FHi, MRb](#)
[MMPYF32 MRa, MRb, MRc || MSUBF32 MRd, MRe, MRf](#)

MSUBF32 MRd, MRe, MRf ||MMOV32 mem32, MRa 32-bit Floating-Point Subtraction with Parallel Move

Operands

| | |
|-------|--|
| MRd | CLA floating-point destination register (MR0 to MR3) for the MSUBF32 operation |
| MRe | CLA floating-point source register (MR0 to MR3) for the MSUBF32 operation |
| MRf | CLA floating-point source register (MR0 to MR3) for the MSUBF32 operation |
| mem32 | 32-bit destination memory location for the MMOV32 operation |
| MRa | CLA floating-point source register (MR0 to MR3) for the MMOV32 operation |

Opcode

LSW: `nnnn nnnn nnnn nnnn`
MSW: `0110 ffee ddaa addr`

Description

Subtract the contents of two floating-point registers and move from a floating-point register to memory.

$MRd = MRe - MRf;$
 $[mem32] = MRa;$

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | Yes | Yes |

The MSTF register flags are modified as follows:

- LUF = 1 if MSUBF32 generates an underflow condition.
- LVF = 1 if MSUBF32 generates an overflow condition.

Pipeline

Both MSUBF32 and MMOV32 complete in a single cycle.

Example

See also

[MSUBF32 MRa, MRb, MRc](#)
[MSUBF32 MRa, #16FHi, MRb](#)
[MSUBF32 MRd, MRe, MRf || MMOV32 MRa, mem32](#)
[MMPYF32 MRa, MRb, MRc || MSUBF32 MRd, MRe, MRf](#)

MSWAPF MRa, MRb {, CNDF} *Conditional Swap*

Operands

| | |
|------|---|
| MRa | CLA floating-point register (MR0 to MR3) |
| MRb | CLA floating-point register (MR0 to MR3) |
| CNDF | Optional condition tested based on the MSTF flags |

Opcode

```
LSW: 0000 0000 CNDF bbaa
MSW: 0111 1011 0000 0000
```

Description

Conditional swap of MRa and MRb.

```
if (CNDF == true) swap MRa and MRb;
```

CNDF is one of the following conditions:

| Encode ⁽¹⁾ | CNDF | Description | MSTF 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 OR 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 and NF 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

No flags affected

Pipeline

This is a single-cycle instruction.

Example

```

; X is an array of 32-bit floating-point values
; and has len elements. Find the maximum value in
; the array and store it in Result
;
; Note: MCMPPF32 and MSWAPF can be replaced by MMAXF32
;
_ClalTask1:
    MMOV16    MAR1, #_X          ; Start address
    MUI16TOF32 MR0, @_len        ; Length of the array
    MNOP                      ; delay for MAR1 load
    MNOP                      ; delay for MAR1 load
    MMOV32    MR1, *MAR1[2]++    ; MR1 = X0
LOOP
    MMOV32    MR2, *MAR1[2]++    ; MR2 = next element
    MCMPPF32  MR2, MR1           ; Compare MR2 with MR1
    MSWAPF    MR1, MR2, GT       ; MR1 = MAX(MR1, MR2)
    MADD32    MR0, MR0, #-1.0    ; Decrement the counter
    MCMPPF32  MR0, #0.0         ; Set/clear flags for MBCNDD
    MNOP
    MNOP
    MNOP
    MBCNDD    LOOP, NEQ         ; Branch if not equal to zero
    MMOV32    @_Result, MR1     ; Always executed
    MNOP                      ; Always executed
    MNOP                      ; Always executed
    MSTOP                      ; End of task

```

See also

MTESTTF CNDF *Test MSTF Register Flag Condition*

Operands

| | |
|------|---------------------------------------|
| CNDF | condition to test based on MSTF flags |
|------|---------------------------------------|

Opcode

```
LSW: 0000 0000 0000 cndf
MSW: 0111 1111 0100 0000
```

Description

Test the CLA floating-point condition and if true, set the MSTF[TF] flag. If the condition is false, clear the MSTF[TF] flag. This is useful for temporarily storing a condition for later use.

```
if (CNDF == true) TF = 1;
else TF = 0;
```

CNDF is one of the following conditions:

| Encode ⁽³⁾ | CNDF | Description | MSTF 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 OR 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 ⁽⁴⁾ | Unconditional with flag modification | None |

⁽³⁾ Values not shown are reserved.

⁽⁴⁾ This is the default operation if no CNDF field is specified. This condition will allow the ZF and NF 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 MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|-----|----|----|-----|-----|
| Modified | Yes | No | No | No | No |

```
TF = 0;
if (CNDF == true) TF = 1;
```

Note: If (CNDF == UNC or UNCF), the TF flag will be set to 1.

Pipeline

This is a single-cycle instruction.

Example

```

; if (State == 0.1)
;   RampState = RampState || RAMPMASK
; else if (State == 0.01)
;   CoastState = CoastState || COASTMASK
; else
;   SteadyState = SteadyState || STEADYMASK
;
_ClalTask2:
  MMOV32    MR0, @_State
  MCMPF32   MR0, #0.1      ; Affects flags for 1st MBCNDD (A)
  MCMPF32   MR0, #0.01     ; Check used by 2nd MBCNDD (B)
  MTESTTF   EQ             ; Store EQ flag in TF for 2nd MBCNDD (B)
  MNOP
  MBCNDD    _Skip1, NEQ     ; (A) If State != 0.1, go to Skip1
  MMOV32    MR1, @_RampState ; Always executed
  MMOVXI    MR2, #RAMPMASK  ; Always executed
  MOR32     MR1, MR2        ; Always executed
  MMOV32    @_RampState, MR1 ; Execute if (A) branch not taken
  MSTOP     ; end of task if (A) branch not taken

_Skip1:
  MMOV32    MR3, @_SteadyState
  MMOVXI    MR2, #STEADYMASK
  MOR32     MR3, MR2
  MBCNDD    _Skip2, NTF     ; (B) if State != .01, go to Skip2
  MMOV32    MR1, @_CoastState ; Always executed
  MMOVXI    MR2, #COASTMASK  ; Always executed
  MOR32     MR1, MR2        ; Always executed
  MMOV32    @_CoastState, MR1 ; Execute if (B) branch not taken
  MSTOP     ; end of task if (B) branch not taken

_Skip2:
  MMOV32    @_SteadyState, MR3 ; Executed if (B) branch taken
  MSTOP

```

See also

MUI16TOF32 MRa, mem16 Convert unsigned 16-bit integer to 32-bit floating-point value

Operands

| | |
|-------|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| mem16 | 16-bit source memory location |

Opcode

```

LSW: mmm mmm mmm mmm
MSW: 0111 0101 01aa addr

```

Description

When converting F32 to I16/UI16 data format, the MF32TOI16/UI16 operation truncates to zero while the MF32TOI16R/UI16R operation will round to nearest (even) value.

```
MRa = UI16TOF32[mem16];
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example
See also

[MF32TOI16 MRa, MRb](#)
[MF32TOI16R MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, MRb](#)

MUI16TOF32 MRa, MRb Convert unsigned 16-bit integer to 32-bit floating-point value

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1110 1110 0000
```

Description

Convert an unsigned 16-bit integer to a 32-bit floating-point value. When converting float32 to I16/UI16 data format, the MF32TOI16/UI16 operation truncates to zero while the MF32TOI16R/UI16R operation will round to nearest (even) value.

```
MRa = UI16TOF32[MRb];
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVXI MR1, #0x800F ; MR1(15:0) = 32783 (0x800F)
MUI16TOF32 MR0, MR1 ; MR0 = UI16TOF32 (MR1(15:0))
                     ; = 32783.0 (0x47000F00)
```

See also

[MF32TOI16 MRa, MRb](#)
[MF32TOI16R MRa, MRb](#)
[MF32TOUI16 MRa, MRb](#)
[MF32TOUI16R MRa, MRb](#)
[MI16TOF32 MRa, MRb](#)
[MI16TOF32 MRa, mem16](#)
[MUI16TOF32 MRa, mem16](#)

MUI32TOF32 MRa, mem32 *Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value*

Operands

| | |
|-------|---|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| mem32 | 32-bit memory location accessed using direct or indirect addressing |

Opcode

```
LSW: mmmm mmmm mmmm mmmm
MSW: 0111 0100 10aa addr
```

Description

```
MRa = UI32TOF32[mem32];
```

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
; Given x2, m2 and b2 are Uint32 numbers:
;
; x2 = Uint32(2) = 0x00000002
; m2 = Uint32(1) = 0x00000001
; b2 = Uint32(3) = 0x00000003
;
; Calculate y2 = x2 * m2 + b2
;
_ClalTask1:
    MUI32TOF32 MR0, @_m2      ; MR0 = 1.0 (0x3F800000)
    MUI32TOF32 MR1, @_x2      ; MR1 = 2.0 (0x40000000)
    MUI32TOF32 MR2, @_b2      ; MR2 = 3.0 (0x40400000)
    MMPYF32 MR3, MR0, MR1     ; M*X
    MADDF32 MR3, MR2, MR3     ; Y=MX+B = 5.0 (0x40A00000)
    MF32TOUI32 MR3, MR3       ; Y = Uint32(5.0) = 0x00000005
    MMOV32 @_y2, MR3          ; store result
    MSTOP                     ; end of task
```

See also

[MF32TOI32 MRa, MRb](#)
[MF32TOUI32 MRa, MRb](#)
[MI32TOF32 MRa, mem32](#)
[MI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, MRb](#)

MUI32TOF32 MRa, MRb *Convert Unsigned 32-bit Integer to 32-bit Floating-Point Value*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 0000 bbaa
MSW: 0111 1101 1100 0000
```

Description

MRa = UI32TOF32 [MRb];

Flags

This instruction does not affect any flags:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|----|----|-----|-----|
| Modified | No | No | No | No | No |

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ    MR3, #0x8000 ; MR3(31:16) = 0x8000
MMOVXI    MR3, #0x1111 ; MR3(15:0) = 0x1111
                ; MR3 = 2147488017
MUI32TOF32 MR3, MR3    ; MR3 = MUI32TOF32 (MR3) = 2147488017.0 (0x4F000011)
```

See also

[MF32TOI32 MRa, MRb](#)
[MF32TOUI32 MRa, MRb](#)
[MI32TOF32 MRa, mem32](#)
[MI32TOF32 MRa, MRb](#)
[MUI32TOF32 MRa, mem32](#)

MXOR32 MRa, MRb, MRc *Bitwise Exclusive Or*

Operands

| | |
|-----|--|
| MRa | CLA floating-point destination register (MR0 to MR3) |
| MRb | CLA floating-point source register (MR0 to MR3) |
| MRc | CLA floating-point source register (MR0 to MR3) |

Opcode

```
LSW: 0000 0000 00cc bbaa
MSW: 0111 1100 1010 0000
```

Description

Bitwise XOR of MRb with MRc.

```
MARa(31:0) = MARb(31:0) XOR MRc(31:0);
```

Flags

This instruction modifies the following flags in the MSTF register:

| Flag | TF | ZF | NF | LUF | LVF |
|----------|----|-----|-----|-----|-----|
| Modified | No | Yes | Yes | No | No |

The MSTF register flags are modified based on the integer results of the operation.

```
NF = MRa(31);
ZF = 0;
if(MRa(31:0) == 0) { ZF = 1; }
```

Pipeline

This is a single-cycle instruction.

Example

```
MMOVIZ MR0, #0x5555 ; MR0 = 0x5555AAAA
MMOVXI MR0, #0xAAAA

MMOVIZ MR1, #0x5432 ; MR1 = 0x5432FEDC
MMOVXI MR1, #0xFEDC

; 0101 XOR 0101 = 0000 (0)
; 0101 XOR 0100 = 0001 (1)
; 0101 XOR 0011 = 0110 (6)
; 0101 XOR 0010 = 0111 (7)
; 1010 XOR 1111 = 0101 (5)
; 1010 XOR 1110 = 0100 (4)
; 1010 XOR 1101 = 0111 (7)
; 1010 XOR 1100 = 0110 (6)

MXOR32 MR2, MR1, MR0 ; MR3 = 0x01675476
```

See also

[MAND32 MRa, MRb, MRc](#)
[MOR32 MRa, MRb, MRc](#)

Appendix A CLA and CPU Arbitration

Typically, CLA activity is independent of the CPU activity. Under the circumstance where both the CLA and the CPU are attempting to access memory or a peripheral register within the same interface concurrently, an arbitration procedure will occur. This appendix describes this arbitration.

A.1 CLA and CPU Arbitration

Typically, CLA activity is independent of the CPU activity. Under the circumstance where both the CLA and the CPU are attempting to access memory or a peripheral register within the same interface concurrently, an arbitration procedure will occur. The one exception is the ADC result registers which do not create a conflict when read by both the CPU and the CLA simultaneously even if different addresses are accessed. Any combined accesses between the different interfaces, or where the CPU access is outside of the interface that the CLA is accessing do not create a conflict.

The interfaces that can have conflict arbitration are:

- CLA Message RAMs
- CLA Program Memory
- CLA Data RAMs

A.1.1 CLA Message RAMs

Message RAMs consist of two blocks. These blocks are for passing data between the main CPU and the CLA. No opcode fetches are allowed from the message RAMs. The two message RAMs have the following characteristics:

- **CLA to CPU Message RAM:**

The following accesses are allowed:

- CPU reads
- CLA reads and writes
- CPU debug reads and writes

The following accesses are ignored

- CPU writes

Priority of accesses are (highest priority first):

1. CLA write
2. CPU debug write
3. CPU data read, program read, CPU debug read
4. CLA data read

- **CPU to CLA Message RAM:**

The following accesses are allowed:

- CPU reads and writes
- CLA reads
- CPU debug reads and writes

The following accesses are ignored

- CLA writes

Priority of accesses are (highest priority first):

1. CLA read
2. CPU data write, program write, CPU debug write
3. CPU data read, CPU debug read
4. CPU program read

A.1.2 CLA Program Memory

The behavior of the program memory depends on the state of the MMEMCFG[PROGE] bit. This bit controls whether the memory is mapped to CLA space or CPU space.

- **MMEMCFG[PROGE] == 0**

In this case the memory is mapped to the CPU. The CLA will be halted and no tasks should be incoming.

- Any CLA fetch will be treated as an illegal opcode condition as described in [Section 3.4](#). This condition will not occur if the proper procedure is followed to map the program memory.
- CLA reads and writes cannot occur
- The memory block behaves as any normal SRAM block mapped to CPU memory space.

Priority of accesses are (highest priority first):

1. CPU data write, program write, debug write
2. CPU data read, program read, debug read
3. CPU fetch, program read

- **MMEMCFG[PROGE] == 1**

In this case the memory block is mapped to CLA space. The CPU can only make debug accesses.

- CLA reads and writes cannot occur
- CLA fetches are allowed
- CPU fetches return 0 which is an illegal opcode and will cause an ITRAP interrupt.
- CPU data reads and program reads return 0
- CPU data writes and program writes are ignored

Priority of accesses are (highest priority first):

1. CLA fetch
2. CPU debug write
3. CPU debug read

NOTE: Because the CLA fetch has higher priority than CPU debug reads, it is possible for the CLA to permanently block debug accesses if the CLA is executing in a loop. This might occur when initially developing CLA code due to a bug. To avoid this issue, the program memory will return all 0x0000 for CPU debug reads (ignore writes) when the CLA is running. When the CLA is halted or idle then normal CPU debug read and write access can be performed.

A.1.3 CLA Data Memory

There are two independent data memory blocks. The behavior of the data memory depends on the state of the MMEMCFG[RAM0E] and MMEMCFG[RAM1E] bits. These bits determine whether the memory blocks are mapped to CLA space or CPU space.

- **MMEMCFG[RAMxE] == 0**

In this case the memory block is mapped to the CPU.

- CLA fetches cannot occur to this block.
- CLA reads return 0
- CLA writes are ignored
- The memory block behaves as any normal SARAM block mapped to the CPU memory space.

Priority of accesses are (highest priority first):

1. CPU data write, program write, debug write
2. CPU data read, program read, debug read
3. CPU fetch, program read

- **MMEMCFG[RAMxE] == 1**

In this case the memory block is mapped to CLA space. The CPU can only make debug accesses.

- CLA fetches cannot occur to this block.
- CLA read and CLA writes are allowed.
- CPU fetches return 0
- CPU data reads and program reads return 0
- CPU data writes and program writes are ignored

Priority of accesses are (highest priority first):

1. CLA write
2. CPU debug write
3. CPU debug read
4. CLA read

A.1.4 Peripheral Registers (ePWM, HRPWM, Comparator)

Accesses to the registers follow these rules:

- If both the CPU and CLA request access at the same time, then the CLA will have priority and the main CPU is stalled.
- If a CPU access is in progress and another CPU access is pending, then the CLA will have priority over the pending CPU access. In this case the CLA access will begin when the current CPU access completes.
- While a CPU access is in progress any incoming CLA access will be stalled.
- While a CLA access is in progress any incoming CPU access will be stalled.
- A CPU write operation has priority over a CPU read operation.
- A CLA write operation has priority over a CLA read operation.
- If the CPU is performing a read-modify-write operation and the CLA performs a write to the same location, the CLA write may be lost if the operation occurs in-between the CPU read and write. For this reason, you should not mix CPU and CLA accesses to same location.

Appendix B Revision History

This document has been revised because of the following technical change(s).

Table 34. Revisions to this Document

| Location | Edits, Deletes, Additions |
|-------------------------------|--|
| Section 4.3.3 | For bits 15-12, value 0010, changed ePWM5 to ePWM4 is the input for interrupt task 4. (EPWM4_INT) for bit 15-12 description. |

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

| Products | | Applications | |
|-----------------------------|--|----------------------------|--|
| Amplifiers | amplifier.ti.com | Audio | www.ti.com/audio |
| Data Converters | dataconverter.ti.com | Automotive | www.ti.com/automotive |
| DLP® Products | www.dlp.com | Communications and Telecom | www.ti.com/communications |
| DSP | dsp.ti.com | Computers and Peripherals | www.ti.com/computers |
| Clocks and Timers | www.ti.com/clocks | Consumer Electronics | www.ti.com/consumer-apps |
| Interface | interface.ti.com | Energy | www.ti.com/energy |
| Logic | logic.ti.com | Industrial | www.ti.com/industrial |
| Power Mgmt | power.ti.com | Medical | www.ti.com/medical |
| Microcontrollers | microcontroller.ti.com | Security | www.ti.com/security |
| RFID | www.ti-rfid.com | Space, Avionics & Defense | www.ti.com/space-avionics-defense |
| RF/IF and ZigBee® Solutions | www.ti.com/lprf | Video and Imaging | www.ti.com/video |
| | | Wireless | www.ti.com/wireless-apps |