:

TriMedia32 Architecture

© 2000 TriMedia Technologies
All rights reserved.

See Terms and Conditions on the next page.

Preliminary Specification

October 1, 2000, Version 1.0.13

TERMS AND CONDITIONS

TriMedia Technologies reserves the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. TriMedia Technologies assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or most work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or most work right infringement, unless otherwise specified. Applications that are described herein for any of these products are for illustrative purposes only. TriMedia Technologies makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

TriMedia Technologies products are not designed for use in life support appliances, devices, or systems where malfunction of a TriMedia Technologies product can reasonably be expected to result in a personal injury. TriMedia Technologies customers using or selling TriMedia Technologies products for use in such applications do so at their own risk and agree to fully indemnify TriMedia Technologies for any damages resulting from improper use or sale.

TriMedia Technologies registers eligible circuits under the Semiconductor Chips Protection Act.

DEFINITIONS

Data Sheet Identification	Product Status	Definition
Objective Specification	Formative or in Design	This data sheet contains the design target or goal specifications for product development. Specifications may change in any manner without notice.
Preliminary Specification	Preproduction Product	This data sheet contains preliminary data, and supplementary data will be published at a later date. TriMedia Technologies reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product Specification	Full Production	This data sheet contains Final Specifications. TriMedia Technologies reserves the right to make changes at any time without notice, in order to improve the design and supply the best possible product.

© 2000 TriMedia Technologies

All rights reserved.

Printed in U.S.A.

TriMedia Technologies, 1840 McCarthy Blvd., Milpitas, CA 95035-7425

Table of Contents

Chapter 1 DSPCPU32 1-1
1.1 Basic Architecture Concepts 1-1
1.1.1 Register Model 1-1
1.1.2 Basic DSPCPU32 Execution Model 1-2
1.1.3 PCSW Overview 1-3
1.1.4 SPC and DPC—Source and Destination Program Counter 1-4
1.1.5 CCCOUNT—Clock Cycle Counter 1-5
1.1.6 Boolean Representation 1-5
1.1.7 Integer Representation 1-5
1.1.8 Floating Point Representation 1-5
1.1.9 Addressing Modes 1-6
1.1.10 Software Compatibility 1-7
1.2 Instruction Set Overview 1-7
1.2.1 Guarding (Conditional Execution) 1-7
1.2.2 Load and Store Operations 1-7
1.2.3 Compute Operations 1-8
1.2.4 Special-Register Operations 1-9
1.2.5 Control-Flow Operations 1-9
1.3 TM32 Instruction Issue Rules 1-10
1.4 Memory Map 1-11
1.4.1 DRAM Aperture 1-12
1.4.2 MMIO Aperture 1-12
1.4.3 Aperture1 1-13
1.5 Special Event Handling 1-14
1.5.1 Reset and Start 1-14
1.5.2 EXC (Exceptions) 1-15
1.5.3 INT and NMI (Maskable and Non-Maskable Interrupts) 1-15
1.6 Timers 1-19
1.7 Debug Support 1-20
1.7.1 Instruction Breakpoints 1-20
1.7.2 Data Breakpoints 1-21
1.8 System 1-22
1.8.1 TM32_CTL 1-22
1.8.2 TM32_MODID 1-23
1.8.3 Endianness 1-23
1.9 Powerdown 1-24
1.9.1 Partial Powerdown 1-24
1.9.2 Full Powerdown 1-24

Chapter 2 Custom Operations for Multimedia 2-1
2.1 Custom Operations Overview 2-1
2.1.1 Custom Operation Motivation 2-1
2.1.2 Introduction to Custom Operations 2-2
2.1.3 Example Uses of Custom Ops 2-4
2.2 Example 1: Byte-Matrix Transposition 2-4
2.3 Example 2: MPEG Image Reconstruction 2-7
2.4 Example 3: Motion-Estimation Kernel 2-11
2.4.1 A Simple Transformation 2-11
2.4.2 More Unrolling 2-13

Chapter 3 Cache Architecture 3-1
3.1 Introduction 3-1
3.2 DRAM Aperture 3-1
3.3 Data Cache 3-2
3.3.1 General Cache Parameters 3-2
3.3.2 Address Mapping 3-3
3.3.3 Miss Processing Order 3-3
3.3.4 Replacement Policies, Coherency 3-3
3.3.5 Alignment, Partial-Word Transfers, Endian-ness 3-4
3.3.6 Dual Ports 3-4
3.3.7 Cache Locking 3-4
3.3.8 Non-cacheable Region 3-5
3.3.9 Special Data Cache Operations 3-6
3.3.10 Memory Operation Ordering 3-8
3.3.11 Operation Latency 3-9
3.3.12 MMIO Aperture References 3-9
3.3.13 Aperture1 References 3-9
3.3.14 CPU Stall Conditions 3-9
3.3.15 Data Cache Initialization 3-9
3.4 Instruction Cache 3-10
3.4.1 General Cache Parameters 3-10
3.4.2 Address Mapping 3-11
3.4.3 Miss Processing Order 3-11
3.4.4 Replacement Policy 3-11
3.4.5 Location of Program Code 3-11
3.4.6 Branch Units 3-11
3.4.7 Coherency: Special iclr Operation 3-12
3.4.8 Reading Tags and Cache Status 3-12
3.4.9 Cache Locking 3-13
3.4.10 Instruction Cache Initialization and Boot Sequence 3-14
3.5 LRU Algorithm 3-14
3.5.1 Two-Way Algorithm 3-14
3.5.2 Four-Way Algorithm 3-15
3.5.3 LRU Initialization 3-15
3.5.4 LRU Bit Definitions 3-15
3.5.5 LRU for the Dual-Ported Cache 3-15
3.6 Cache Coherency 3-16
3.6.1 Example 1: Data-Cache/Input-Unit Coherency 3-16
3.6.2 Example 2: Data-Cache/Output-Unit Coherency 3-16
3.6.3 Example 3: Instruction-Cache/Data-Cache Coherency 3-16

 © TriMedia Technologies, Inc.

3.6.4 Example 4: Instruction-Cache/Input-Unit Coherency 3-17
3.7 Performance Evaluation Support 3-17
3.8 MMIO Register Summary 3-18

Chapter 4 On-Chip Semaphore Assist Device 4-1

4.1 SEM Device Specification 4-1
4.2 Constructing a 12-Bit ID 4-2
4.3 Which SEM to Use 4-2
4.4 Usage Notes 4-2

Instruction Set A-1

A.1 Alphabetic Operation List A-1
A.2 Operation List By Function A-2
alloc A-3
allocd A-4
allocr A-5
allocx A-6
asl A-7
asli A-8
asr A-9
asri A-10
bitand A-11
bitandinv A-12
bitinv A-13
bitor A-14
bitxor A-15
borrow A-16
carry A-17
curcycles A-18
cycles A-19
dcb A-20
dinvalid A-21
dspiabs A-22
dspiadd A-23
dspidualabs A-24
dspidualadd A-25
dspidualmul A-26
dspidualsub A-27
dspimul A-28
dspisub A-29
dspuadd A-30
dspumul A-31
dspuquadaddui A-32
dspusub A-33
dualasr A-34

dualiclipi A-35
dualuclipi A-36
fabsval A-37
fabsvalflags A-38
fadd A-39
faddflags A-40
fdiv A-41
fdivflags A-42
feql A-43
feqlflags A-44
fgeq A-45
fgeqflags A-46
fgtr A-47
fgtrflags A-48
fleq A-49
fleqflags A-50
fles A-51
flesflags A-52
fmul A-53
fmulflags A-54
fneq A-55
fneqflags A-56
fsign A-57
fsignflags A-58
fsqrt A-59
fsqrtflags A-60
fsub A-61
fsubflags A-62
funshift1 .... A-63
funshift2 .... A-64
funshift3 .... A-65
h_dspiabs A-66
h_dspidualabs A-67
h_iabs A-68
h_st16d A-69
h_st32d A-70
h_st8d A-71
hicycles A-72
iabs A-73
iadd A-74
iaddi A-75
iavgonep A-76
ibytesel A-77
iclipi A-78
iclr A-79
ident A-80

 © TriMedia Technologies, Inc.

ieql A-81
ieqli A-82
ifir16 .... A-83
ifir8ii A-84
ifir8ui A-85
ifixieee A-86
ifixieeeflags A-87
ifixrz A-88
ifixrzflags A-89
iflip A-90
ifloat A-91
ifloatflags A-92
ifloatrz A-93
ifloatrzflags A-94
igeq A-95
igeqi A-96
igtr A-97
igtri A-98
iimm A-99
ijmpf A-100
ijmpi A-101
ijmpt A-102
ild16 .... A-103
ild16d A-104
ild16r A-105
ild16x A-106
ild8 .... A-107
ild8d A-108
ild8r A-109
ileq A-110
ileqi A-111
iles A-112
ilesi A-113
imax A-114
imin A-115
imul A-116
imulm A-117
ineg A-118
ineq A-119
ineqi A-120
inonzero A-121
isub A-122
isubi A-123
izero A-124
jmpf A-125
jmpi A-126

jmpt A-127
ld32 ..... A-128
ld32d A-129
ld32r A-130
ld32x A-131
lsl A-132
lsli A-133
lsr A-134
lsri A-135
mergedual16lsb A-136
mergelsb A-137
mergemsb A-138
nop A-139
pack16lsb A-140
pack16msb A-141
packbytes A-142
pref A-143
pref16x A-144
pref32x A-145
prefd A-146
prefr A-147
quadavg A-148
quadumax A-149
quadumin A-150
quadumulmsb A-151
rdstatus A-152
rdtag A-153
readdpc A-154
readpcsw A-155
readspc A-156
rol A-157
roli A-158
sex16 ..... A-159
sex8 ..... A-160
st16 ..... A-161
st16d A-162
st32 ..... A-163
st32d A-164
st8 ..... A-165
st8d A-166
ubytesel A-167
uclipi A-168
uclipu A-169
ueql A-170
ueqli A-171
ufir16 ..... A-172

 © TriMedia Technologies, Inc.

ufir8uu A-173
ufixieee A-174
ufixieeeflags A-175
ufixrz A-176
ufixrzflags A-177
ufloat A-178
ufloatflags A-179
ufloatrz A-180
ufloatrzflags A-181
ugeq A-182
ugeqi A-183
ugtr A-184
ugtri A-185
uimm A-186
uld16 .... A-187
uld16d A-188
uld16r A-189
uld16x A-190
uld8 .... A-191
uld8d A-192
uld8r A-193
uleq A-194
uleqi A-195
ules A-196
ulesi A-197
ume8ii A-198
ume8uu A-199
umin A-200
umul A-201
umulm A-202
uneq A-203
uneqi A-204
writedpc A-205
writepcsw A-206
writespc A-207
zex16 .... A-208
zex8 .... A-209
.... A-210

MMIO Registers B-1

Signal Interface C-1
C.1 Signals C-1
C.1.1 Timing C-4
C.1.2 Interface Description C-4

Chapter 1 DSPCPU32

1.1 Basic Architecture Concepts

This section documents the system programmer or ‘bare-machine’ view of the TM32 CPU (or DSPCPU32).

1.1.1 Register Model

Figure 1-1 shows the TM32’s 128 general purpose registers, r0...r127. In addition to the hardware program counter, PC, there are 4 user-accessible special purpose registers, PCSW, DPC (destination program counter), SPC (source program counter), and CCCOUNT. Table 1-1 lists the registers and their purposes.

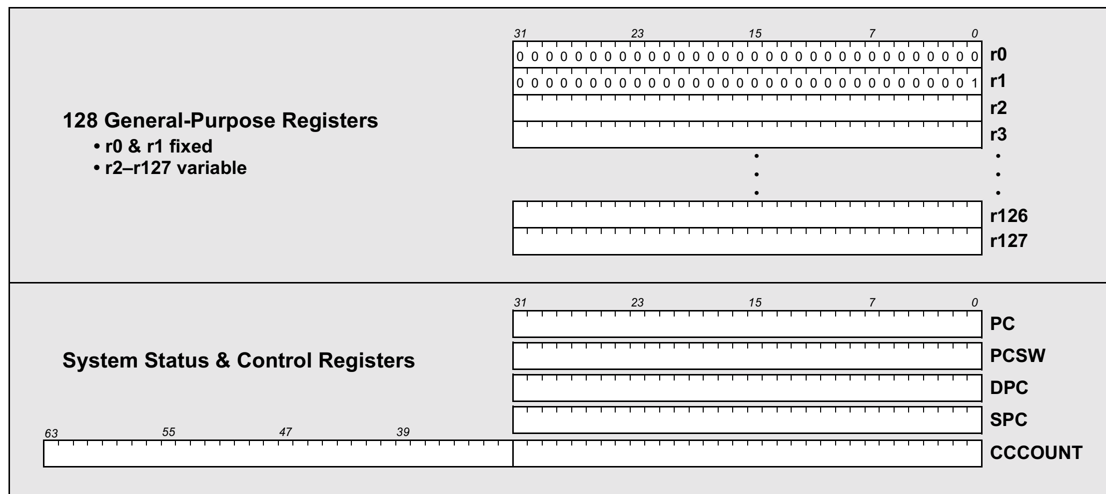


Figure 1-1 TM32 registers.

Register r0 always contains the integer value '0', corresponding to the boolean value 'FALSE' or the single-precision floating point value +0.0. Register r1 always contains the integer value '1' ('TRUE'). The programmer is NOT allowed to write to r0 or r1.

Note: Writing to r0 or r1 may cause reads from r0 or r1 scheduled in adjacent clock cycles to return unpredictable values. The standard assembler prevents/forbids the use of r0 or r1 as a destination register.

Registers r2 through r127 are true general purpose registers; the hardware does not imply their use in any way, though compiler or programmer conventions may assign particular roles to particular registers. The

DPC and SPC relate to interrupt and exception handling and are treated in Section 1.1.4. The PCSW (Program Control and Status Word) register is treated in Section 1.1.3. CCCOUNT, the 64-bit clock cycle counter is treated in Section 1.1.5. The program counter specifies the address of the instruction that is currently being executed[1]. It can be accessed by reading TM32_PC (MMIO offset 0x10,004c). Reading this read only register from TM32 will return the address of the instruction containing the load operation.

Table 1-1 TM32 registers

Register	Size	Details
r0	32 bits	Always reads as 0x0; must not be used as destination of operations
r1	32 bits	Always reads as 0x1; must not be used as destination of operations
r2–r127	32 bits	126 general-purpose registers
PC	32 bits	Program counter
PCSW	32 bits	Program control & status word
DPC	32 bits	Destination program counter; latches target of taken branch that is interrupted
SPC	32 bits	Source program counter; latches target of taken branch that is not interrupted
CCCOUNT	64 bits	Counts clock cycles since reset

1.1.2 Basic DSPCPU32 Execution Model

The DSPCPU32 issues one ‘long instruction’ every clock cycle. Each instruction consists of several operations (five operations for the TM32 microprocessor). Each operation is comparable to a RISC machine instruction, except that the execution of an operation is conditional upon the content of a general purpose register. Examples of operations are:

```
IF r10 iadd r11 r12 → r13
```

(if r10 true, add r11 and r12 and write sum in r13)

```
IF r10 ld32d(4) r15 → r16
```

(if r10 true, load 32 bits from mem[r15+4] into r16)

```
IF r20 jmpf r21 r22
```

(if r20 true and r21 false, jump to address in r22)

Each operation has a specific, known execution latency in clock cycles. For example, iadd takes 1 cycle; thus the result of an iadd operation started in clock cycle i is available for use as an argument to operations issued in cycle $i+1$ or later. The other operations issued in cycle i cannot use the result of iadd. The ld32d operation has a latency of 3 cycles. The result of an ld32d operation started in cycle j is available for use by other operations issued in cycle $j+3$ or later. Branches, such as the jmpf example above have three delay slots. This means that if a branch operation in cycle k is taken, all operations in the instructions in cycle $k+1$, $k+2$ and $k+3$ are still executed.

In the above examples, r10 and r20 control conditional execution of the operations. Also known as ‘guarding’, here r10 and r20 contain the operation ‘guard’. See Section 1.2.1.

Certain restrictions exist in the choice of what operations can be packed into an instruction. For example, the DSPCPU32 allows no more than two load/store class operations to be packed into a single instruction.

1. Multiple instuctions will be in the TM32 pipeline at any given time. PC specifies the instruction that is or would be in the second execute stage

Also, no more than five results (of previously started operations) can be written during any one cycle. The packing of operations is not normally done by the programmer. Instead, the *instruction scheduler* (See TriMedia SDE Reference Manual) takes care of converting the parallel intermediate format code into packed instructions ready for the assembler. The rules are formally described in the *machine description file* used by the instruction scheduler and other tools.

1.1.3 PCSW Overview

Figure 1-2 shows the PCSW register. The TM32 value of PCSW on reset is 0x800.

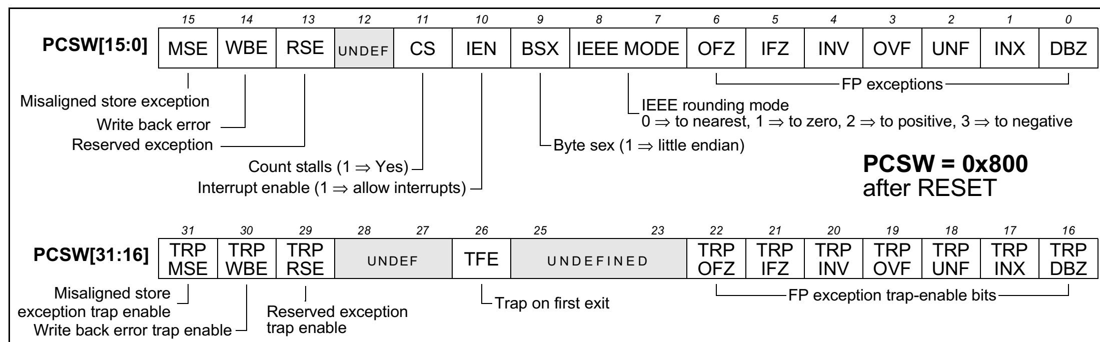


Figure 1-2 TM32 PCSW (Program Control and Status Word) register

For compatibility, any undefined PCSW fields should never be modified.

Note that the DSPCPU32 architecture has no condition codes or integer arithmetic status flags. Integer operations that generate out-of-range results deliver an operation specific bit pattern. For example, see dspiadd in Appendix A. Predicate operations exist that take the place of integer status flags in a classical architecture. Multiword arithmetic is supported by the 'carry' operation which generates a '0' or '1' depending on the carry that would be generated if its arguments were summed.

FP-Related Fields. The IEEE mode field determines the IEEE rounding mode of all floating point operations, with the exception of a few floating point conversion operations that use fixed rounding mode. For example, see ifixrz, ifloatrz, ifixrz and ifloatrz in Appendix A.

The *FP exception flags* are 'sticky bits' that are set as a side effect of floating-point computations. Each floating point operation can set one or more of the flags if it incurs the corresponding exception. The flags can only be reset by direct software manipulation of the PCSW (using the writepcsw operation). The bits have the meanings shown in Table 1-2.

The *FP exception trap enable bits* determine which FP exception flags invoke CPU exception handling. An exception is requested if the intersection of the exception flags and trap enable flags is non-zero. The acceptance and handling of exceptions is described in Section 1.5.

BSX (Bytesex). The DSPCPU32 has a switchable bytesex. The BSX flag in the PCSW can be written by software. Load/store operations observe little- or big-endian byte ordering based on the current setting of BSX.

Table 1-2 PCSW FP exception flag definitions

Flag	Function
INV	Standard IEEE invalid flag
OVF	Standard IEEE overflow flag
UNF	Standard IEEE underflow flag

Table 1-2 PCSW FP exception flag definitions

Flag	Function
INX	Standard IEEE inexact flag
DBZ	Standard IEEE divide-by-zero flag
OFZ	'Output flushed to zero' set if an operation caused a denormalized result
IFZ	'Input flushed to zero' set if an operation was applied to one or more denormalized operands

IEN (Interrupt Enable). The IEN flag disables or enables interrupt processing for most interrupt sources. Only NMI (non-maskable interrupt) bypasses IEN. The acceptance and handling of interrupts is described in Section 1.5.3.

CS (Count Stalls). The CS flag determines the mode of CCCOUNT, the 64-bit clock cycle counter. If CS = '1', the cycle counter increments on all clock cycles. If CS = '0', the clock cycle counter only increments on non-stall cycles. See also Section 1.1.5. After RESET, CS is set to '1'.

MSE and TRPMSE (Misaligned-Store Exception). The MSE bit will be set when the processor detects a store operation to an address that is not aligned. For example, a 32-bit store executed with an address that is not a multiple of four will cause MSE to be set. The TRPMSE bit enables the DSPCPU32 to raise misaligned address exceptions. An exception is requested if the intersection of MSE and TRPMSE is non-zero. The acceptance and handling of exceptions is described in Section 1.5.

Unaligned load operations do not cause an exception, because load operations can be speculative (i.e. their result is thrown away).

When the DSPCPU32 generates an unaligned address, the low order address bit(s) (one bit in the case of a 16-bit load, two bits for a 32-bit load) are forced to zero and the load/store is executed from this aligned address.

WBE and TRPWBE (Write Back Error). The WBE flag will be set whenever a program attempts to write back more than 5 results simultaneously. This is indicative of a programming error, likely caused by the scheduler or assembler. The TRPWBE bit enables the corresponding exception.

RSE, TRPRSE (Reserved Exception). RSE and TRPRSE are reserved for diagnostic purposes and not described here.

TFE (Trap on First Exit). The TFE bit is a support bit for the debugger. The TFE bit is set by the debugger prior to taking a (non-interruptible) jump to the application program. On the next interruptible jump (the first interruptible jump in the application being debugged), an exception is requested because the TFE bit is set. The acceptance and handling of exception processing is described in Section 1.5. It is the responsibility of the exception handler software to clear the TFE bit. The hardware does not clear or set TFE.

Note: *Corner-case:* Whenever a hardware update (e.g. an exception being raised) and a software update (through writepcsw) of the PCSW coincide, the new value of the PCSW will be the value that is written by the writepcsw instruction, except for those bits that the hardware is currently updating (which will reflect the hardware value).

1.1.4 SPC and DPC—Source and Destination Program Counter

The SPC and DPC registers are support registers for exception processing. The DPC is updated during every interruptible jump with the target address of that interruptible jump. If an exception is taken at an interruptible jump, the value in the DPC register can be used by the exception handling routine as the return address to resume the program at the place of interruption.

The SPC register is updated during every interruptible jump that is not interrupted by an exception. Thus on an interrupted interruptible jump, the SPC register is not updated. The SPC register allows the exception

© TriMedia Technologies, Inc.

handling routine to determine the start address of the decision tree (a block of uninterruptible, scheduled TM32 code) that was executing when the exception was taken (see also Section 1.5).

Note: *Corner-case:* Whenever a hardware update (during an interruptible jump) and a software update (through writedpc or writespc) coincide, the software update takes precedence.

1.1.5 CCCOUNT—Clock Cycle Counter

CCCOUNT is a 64-bit counter that counts clock cycles since RESET. Cycle counting can occur in two modes, depending on PCSW.CS. If PCSW.CS = ‘1’, the cycle count increments on every CPU clock cycle. If PCSW.CS = ‘0’, the clock cycle count only increments on non-stall CPU cycles.

CCCOUNT is implemented as a master counter/slave register pair. The master 64-bit counter gets updated continuously. The value of the CCCOUNT slave register is updated with the current master cycle count during successful interruptible jumps only. The *cycles* and *hicycles* DSPCPU32 operations return the content of the 32 LSBs and 32 MSBs, respectively, of the slave register. This ensures that the value returned by hicycles and cycles is coherent, as long as there is no intervening interruptible jump, which makes these operations suitable for 64-bit high resolution timing from C source code programs. The *curcycles* DSPCPU32 operation returns the 32 LSBs of the master counter. The latter operation can be used for instruction cycle precise timing. When used, it must be precisely placed, probably at the assembly code level.

1.1.6 Boolean Representation

The bit pattern generated by boolean valued operations (ileq, fleq etc.) is '00...00' (FALSE) or '00...01' (TRUE). When interpreting a bit pattern as a boolean value, only the LSB is taken into account, i.e. 'xx..x0' is interpreted as FALSE and 'xx..x1' is interpreted as TRUE. In particular, wherever a general purpose register is used as a ‘guard’, the LSB determines whether execution of the guarded operation takes place.

1.1.7 Integer Representation

The architecture supports the notion of 'unsigned integers' and 'signed integers.' Signed integers use the standard two’s-complement representation.

Arithmetic on integers does not generate traps. If a result is not representable, the bit pattern returned is operation specific, as defined in the individual operation description section. The typical cases are:

- Wrap around for regular add- and subtract-type operations.
- Clamping against the minimum or maximum representable value for DSP-type operations.
- Returning the least significant 32-bit value of a 64-bit result (e.g., integer/unsigned multiply).

1.1.8 Floating Point Representation

The TM32 architecture supports single precision (32-bit) IEEE-754 floating point arithmetic.

All arithmetic conforms to the IEEE-754 standard in flush-to-zero mode.

All floating point compute operations round according to the current setting of the *PCSW IEEE mode* field. The current setting of the field determines result rounding (to nearest, to zero, to positive infinity, to negative infinity). Conversions from float to integer/unsigned are available in two forms: a PCSW rounding-mode-observing form and an ANSI-C-specific-rounding form. The ANSI-C-specific form forces round to

zero regardless of the PCSW IEEE rounding mode. Conversion from integer/unsigned to float always observes the IEEE rounding mode.

Floating point exceptions are supported with two mechanisms. Each individual floating point operation (e.g. fadd) has a counterpart operation (faddflags) that computes the exception flag values. These operations can be used for precise exception identification[1]. The second mechanism uses the 'sticky' exception bits in the PCSW that collect aggregate exception events. The PCSW exception bits can selectively invoke CPU exception handling. See Section 1.5.2.

Table 1-3 shows the representation choices that were made in TM32's floating point implementation.

Table 1-3 Special Float Value Representation

Item	Representation
+inf	0x7f800000
-inf	0xff800000
self generated qNaN	0xffffffff
result of operation on any NaN argument	argument | 0x00400000 (forcing the NaN to be quiet)
signalling NaN	never generated by TM32, accepted as per IEEE-754

1.1.9 Addressing Modes

The addressing modes shown in Table 1-4 are supported by the DSPCPU32 architecture (store operations allow only displacement mode).

Table 1-4 Addressing Modes

Mode	Suffix	Applies to	Name
R[i] + scaled(#j)	d	Load & Store	Displacement
R[i] + R[k]	r	Load only	Index
R[i] + scaled(R[k])	x	Load only	Scaled index

In these addressing modes, R[i] indicates one of the general purpose registers. The scale factor applied (1/2/4) is

Table 1-5 Minimum values for implementation-dependent addressing mode components

Parameter	Minimum Range
'i' and 'k'	0..127 (i.e., each implementation has at least 128 registers)
'j'	-64..63 (i.e., displacements will be at least 7 bits long and signed)

equal to the size of the item loaded or stored, i.e. 1 for a byte operation, two for a 16-bit operation and four for a 32-bit operation. The range of valid 'i', 'j' and 'k' values may differ between implementations of the architecture; the minimum values for implementation-dependent characteristics are shown in Table 1-5.

1. This mechanism allows precise exception identification in the context of our multi-issue microprocessor core—where many floating point operations may issue simultaneously—at the expense of additional operations generated by the compiler. It also allows the compiler to issue compute operations speculatively and compute exceptions precisely.

 © TriMedia Technologies, Inc.

Note that the assembly code specifies the true displacement, and not the value to be scaled. For example, ‘ld32d(–8) r3’ loads a 32-bit value from address (r3 – 8). This is encoded in the binary operation pattern as a –2 in the seven-bit field by the assembler. At runtime, the scale factor four is applied to reconstruct the intended displacement of –8.

1.1.10 Software Compatibility

The DSPCPU32 architecture expressly does not support binary compatibility between family members. The ANSI C compiler ensures that all family members are compatible at the source-code level.

1.2 Instruction Set Overview

1.2.1 Guarding (Conditional Execution)

In the TM32 architecture, all operations can be optionally 'guarded'. A guarded operation executes conditionally, depending on the value in the ‘guard' register. For example, a guarded add is written as:

```
IF R23 iadd R14 R10 → R13
```

This should be taken to mean

```
if R23 then R13 ← R14 + R10.
```

The ’if R23' clause controls the execution of the operation based on the LSB of R23. Hence, depending on the LSB of R23, R13 is either unchanged or set to contain the integer sum of R14 and R10.

Guarding applies to all DSPCPU32 operations, except iimm and uimm (load-immediate). It controls the effect on all programmer-visible states of the system, i.e. register values, memory content, exception raising and device state.

1.2.2 Load and Store Operations

Memory is byte addressable. Loads and stores must be ‘naturally aligned’, i.e. a 16-bit load or store must target an address that is a multiple of 2. A 32-bit load or store must target an address that is a multiple of 4. The BSX bit in the PCSW determines the byte order of loads and stores. For example, see ld32 and st32 in Appendix A.

Only 32-bit load and store operations are allowed to access MMIO registers in the MMIO address aperture (see Section 1.4). The results are undefined for other loads and stores. A load from a non-existent MMIO register returns an undefined result. A store to a non-existent MMIO register times out and then does not happen. There are no other side effects of an access to a nonexistent MMIO register. The state of the BSX bit has no effect on the result of MMIO or Aperture1 accesses.

Loads are allowed to be issued speculatively. Loads outside the range of valid data memory addresses for the active process return an implementation-dependent value and do not generate an exception. Misaligned loads also return an implementation dependent value and do not generate an exception.

If a pair of memory operations involves one or more common bytes in memory, the effect on the common bytes is as defined in Table 1-6.

Table 1-6 Behavior of loads and stores with coincident addresses

Condition	Behavior
$T_{store} < T_{load}$	If a store is issued before a load, the value loaded contains the new bytes.
$T_{load} < T_{store}$	If a load is issued before a store, the value loaded contains the old bytes.
$T_{store1} < T_{store2}$	If store1 is issued before store2, the resulting value contains the bytes of store2.
$T_{store} = T_{load}$	If a load and store are issued in the same clock cycle, the result is UNDEFINED.
$T_{store1} = T_{store2}$	If two stores are issued in the same clock cycle, the resulting stored value is undefined.

Table 1-4 shows the supported addressing modes. The minimum values of implementation-dependent addressing-mode components are shown in Table 1-5.

Note: The index and scaled-index modes are not allowed with store opcodes, due to the hardware restriction that each operation have at most 2 source operand registers and 1 condition register. Stores use 1 operand register for the value to be stored leaving only 1 register to form an address.

The scale factor applied (1/2/4) in the scaled addressing modes is equal to the size of the item loaded or stored, i.e. 1 for a byte operation, 2 for a 16-bit operation and 4 for a 32-bit operation.

Table 1-7 lists the available load and store mnemonics for the three addressing modes.

Table 1-7 Load and store mnemonics

Operation	Displacement	Index	Scaled-Index
8-bit signed load	ild8d	ild8r	—
8-bit unsigned load	uld8d	uld8r	—
16-bit signed load	ild16d	ild16r	ild16x
16-bit unsigned load	uld16d	uld16r	uld16x
32-bit load	ld32d	ld32r	ld32x
8-bit store	st8d	—	—
16-bit store	st16d	—	—
32-bit store	st32d	—	—

Example usage of load and store operations:

```
IF r10 ild16d(12) r12 → r13
```

If the LSB of r10 is set, load 16 bits starting at address (r12+12) using the byte ordering indicated in PCSW.BSX, sign-extend the value to 32 bits and store the result in r13.

```
IF r10 st32d(40) r12 r13
```

If the LSB of r10 is set, store the 32-bit value from r13 to the address (r12+40) using the byte ordering indicated in PCSW.BSX.

1.2.3 Compute Operations

Compute operations are register-to-register operations. The specified operation is performed on one or two source registers and the result is written to the destination register.

Immediate Operations. Immediate operations load an immediate constant (specified in the opcode) and produce a result in the destination register.

Floating-Point Compute Operations. Floating-point compute operations are register-to-register operations. The specified operation is performed on one or two source registers and the result is written to the destination register. Unless otherwise mentioned all floating point operations observe the rounding mode bits defined in the PCSW register. All floating-point operations not ending in ‘flags’ update the PCSW exception flags. All operations ending in ‘flags’ compute the exception flags as if the operation were executed and return the flag values (in the same format as in the PCSW); the exception flags in the PCSW itself remain unchanged.

Multimedia Operations. These special compute operations are like normal compute operations, but the specified operations are not usually found in general purpose CPUs. These operations provide special support for multimedia applications.

1.2.4 Special-Register Operations

Special register operations operate on the special registers: PCSW, DPC, SPC and CCCOUNT.

1.2.5 Control-Flow Operations

Control-flow operations change the value of the program counter. Conditional jumps test the value in a register and, based on this value, change the program counter to the address contained in a second register or continue execution with the next instruction. Unconditional jumps always change the program counter to the specified immediate address.

Control-flow operations can be interruptible or non-interruptible. Execution of an interruptible jump is the only occasion where TM32 allows special event handling to take place (see Section 1.5).

1.3 TM32 Instruction Issue Rules

The TM32 VLIW CPU allows issue of 5 operations in each clock cycle according to a set of specific issue rules. The issue rules impose issue time constraints and a result writeback constraint. Any set of operations that meets all constraints constitutes a legal TM32 instruction.

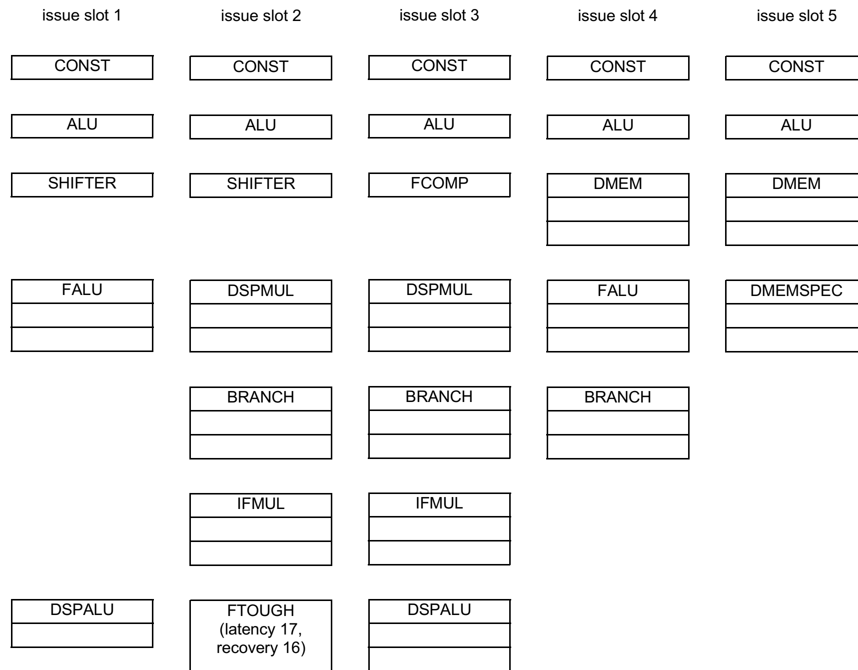


Figure 1-3 TM32 issue slots, functional units, and latency.

A more extensive description and a few special case issue rules and limitations can be found in the TriMedia SDE documentation.

Issue time constraints:

- an operation implies a need for a functional unit type (as documented in Appendix A)
- each operation requires an issue slot that has an instance of the appropriate functional unit type attached
- functional units should be ‘recovered’ from any prior operation issues

Writeback constraint:

- No more than 5 results should be simultaneously written to the register file at any point in time (writeback occurs ‘latency’ cycles after issue)

Figure 1-3 shows all functional units of TM32, including the relation to issue slots, and each functional unit’s latency (e.g. 1 for CONST, 3 for FALU, etc.). With the exception of FTOUGH, each functional unit can accept an operation every clock cycle, i.e. has a recovery time of 1. The binding of operations to func-

tional unit types is summarized in Table 1-8. In Appendix A, each operation lists the precise functional unit and unit latency.

Table 1-8 Functional unit operations

unit type	operation category
const	immediate operations
alu	32-bit arithmetic, logical, pack/unpack
dspalu	dual 16-bit, quad 8-bit multimedia arithmetic
dspmul	dual 16-bit and quad 8-bit multimedia multiplies
dmem	loads/stores
dmemspec	cache coherency, cache control, prefetch
shifter	multi-bit shift
branch	control flow
falu	floating point arithmetic & conversions
ifmul	32-bit integer and floating point multiplies
fcomp	single cycle floating point compares
ftough	iterative floating point square root and division

1.4 Memory Map

The memory map seen by the DSPCPU32 contains 3 apertures as seen in Figure 1-4. Accesses outside the 3 apertures return a ‘0’ on loads, and have no effect for stores.

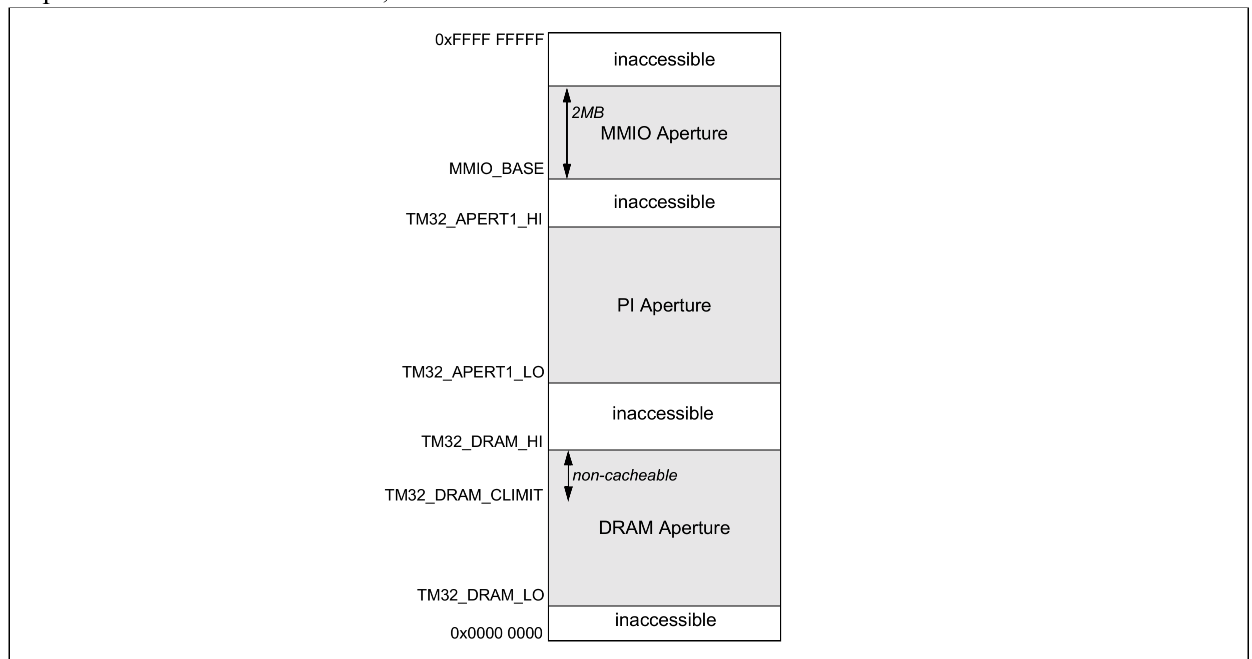


Figure 1-4 TM32 memory map.

Each aperture is independently positionable. Apertures should not be set to overlap or extend across the 0xffff,ffff limit of 32 bit addressing conventions.

1.4.1 DRAM Aperture

The DRAM Aperture is of variable size. It must be a multiple of 64 kBytes, and can only be positioned at 64 kBytes origin. The origin value is determined by the content of the TM32_DRAM_LO MMIO register inside TM32. The last byte of the DRAM aperture is at address TM32_DRAM_HI-1.

The DRAM aperture is divided into two parts:

- cacheable area from TM32_DRAM_LO to TM32_DRAM_CLIMIT-1
- non-cacheable area from TM32_DRAM_CLIMIT to TM32_DRAM_HI-1

The TM32 CPU can access the entire DRAM aperture data with all load and store instructions. Loads and stores in the cacheable area use the data cache. Loads and stores in the non-cacheable area bypass the data-cache and go directly to memory across the memory interface. Execution is supported from the entire DRAM aperture, and always uses the instruction cache.

TM32_DRAM_LOW, TM32_DRAM_HIGH, and TM32_DRAM_CLIMIT are all 64kByte aligned and are read only for TM32 itself while being read/write for external PI-masters. On reset, TM32_DRAM_LOW is set to 0x0000,0000, TM32_DRAM_HI to 0x0100,0000, and TM32_DRAM_CLIMIT to 0x00ff,0000.

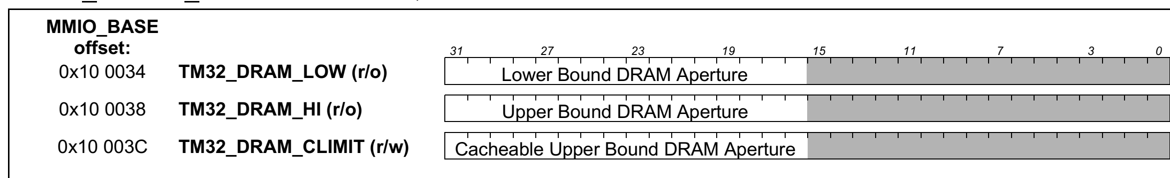


Figure 1-5 DRAM Aperture Registers

1.4.2 MMIO Aperture

The MMIO Aperture is 2 MBytes in size. Its origin is determined by the value on the 'tm32_pci_mmio_base' core input pins (MMIO_BASE). The origin must be a multiple of 2 MBytes. The origin value is only allowed to be changed when the TM32 CPU is stopped.

In typical systems, the MMIO_BASE is provided by a PCI block and MMIO_BASE will be the same as the PCI BASE_14 value, i.e. the origin of the second aperture in PCI space.

TM32 can only perform 32 bit aligned loads and stores to the MMIO aperture. Execution from this aperture, or access with sizes other than 32 bits, is not supported.

Figure 1-6 summarizes the layout of the MMIO aperture. A TM32 access to a 32 bit location in the MMIO aperture will either access the Icache tags/LRU bits, a TM32 internal MMIO register, or a PI-bus system device register.

MMIO locations with offset 0x0 - 0x00,ffff constitute a 64 kByte read-only Icache LRU/tags area. This area is only visible to the a TM32 itself, and is not accessible by other PI-bus masters. Refer to Section 3.4.8 for details.

Accesses to MMIO locations with offset 0x10,0000 - 0x10,1fff resolve to core internal MMIO registers. The full set of TM32 internal registers in this range are defined in Appendix B. A write to a non-existent register address has no effect. A read from a non-existent register returns zeroes, and has no side-effects.

Any 32 bit access to a MMIO location outside the Icache tags or core internal register area causes a PI-bus access.

Note: to provide the basic mechanisms for security, some of the TM32 core internal MMIO registers are R/O (read-only) to the TM32 CPU, and can only be written by PI masters (in particular the internal MIPS CPU, external PCI host and boot block). Writing to such registers from the TM32 CPU has no effect. No TM32 CPU exception is raised - the value written is discarded.

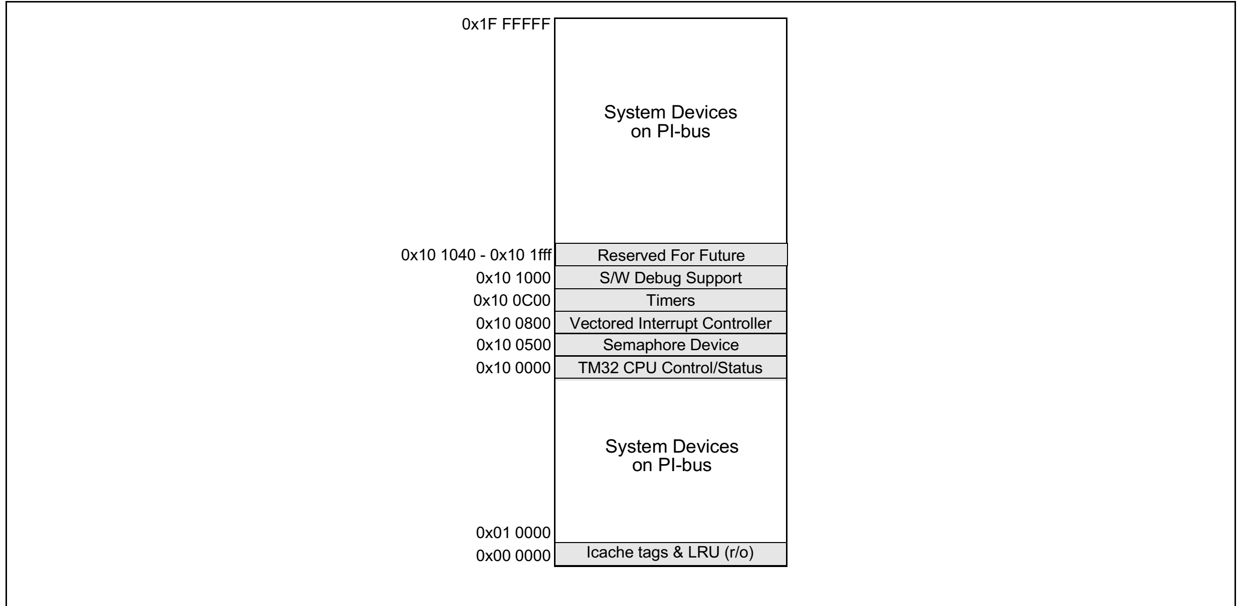


Figure 1-6 TM32 MMIO Overview

1.4.3 Aperture1

The Aperture1 is of variable size and intended to provide access to memory mapped peripherals outside of the MMIO aperture. It must be a multiple of 64 kBytes, and can only be positioned at 64 kBytes origin. The origin value is determined by the content of the TM32_APERT1_LO MMIO register inside TM32. The last byte of Aperture1 is at address TM32_APERT1_HI-1. If TM32_APERT1_HI is less than or equal to TM32_APERT1_LO, Aperture1 is disabled.

The DSPCPU32 can access Aperture1 data with all load and store instructions. Such loads and stores always bypass the datacache, and go directly to the PI-bus. Note that the TM32 CPU always reads a full 32-bit word across the PI-bus on 8 and 16 bit load operations, and internally selects the appropriate byte(s) based on address and endian mode. Writes of 8 and 16 bis will use byte enables.

TM32_APERT1_LOW and TM32_APERT1_HI are both 64kByte aligned and are read only for TM32 itself while being read/write for external PI-masters. On reset, TM32_APERT1_LOW is set to 0x1C00,0000 and TM32_APERT1_HI to 0x2000,0000.

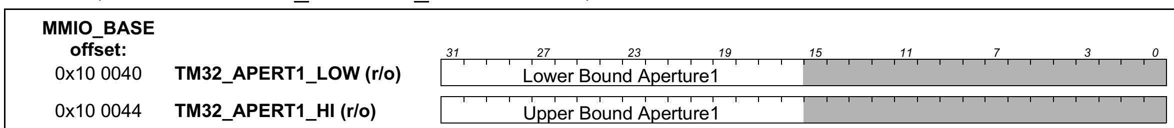


Figure 1-7 Aperture1 Registers

1.5 Special Event Handling

The TM32 microprocessor responds to the special events shown in Table 1-9, ordered by priority.

With the exception of RESET, which is enabled at all times, the architecture of the DSPCPU32 allows special event handling to begin only during an *interruptible jump* operation (ijmpt, ijmpf or ijmpi) that succeeds (i.e., is a taken jump). EXC, NMI and INT handling can be initiated during handling of an EXC or an INT, but *only* during successful interruptible jumps.

Table 1-9 Special Events and Event Vectors

Event	Vector
RESET	(Highest priority) vector to TM32_START_ADR (taken after receiving start command)
EXC	(All exceptions) vector to EXCVEC (programmable)
NMI, INT	(Non-maskable interrupt, maskable interrupt) use the programmed vector (one of 32 vectors depending on the interrupt source)

The *instruction scheduler* uses interruptible jumps exclusively for inter-decision tree jumps. Hence, within a decision tree, no special-event processing can be initiated. If a tree-to-tree jump is taken, special-event processing is allowed. Since the only registers live at this point (i.e., that contain useful data) are the *global registers* allocated by the ANSI C compiler, only a subset of the registers needs to be preserved by the event handlers. Refer to the TriMedia SDE Reference Manual for details on which registers can be in use. The DSPCPU32 register state can be described by the contents of this subset of general purpose registers and the contents of the PCSW and the DPC value (the target of the inter-tree jump).

The priority resolution mechanism built into the DSPCPU32 hardware dispatches the highest-priority, non-masked special-event request at the time of a successful interruptible jump operation. In view of the simple, real-time-oriented nature of the mechanisms provided, only limited nesting of events should be allowed.

1.5.1 Reset and Start

RESET is the highest priority special event. There are two ways to reset the TM32 CPU core. Both have identical effect.

- hardware reset, by asserting and then negating the tm32_reset_n input to the core.
- MMIO reset, by writing the code for ’stop and reset’ to the TM32_CTL MMIO register

Upon either reset, the TM32 CPU puts all registers and cache control in its defined initial state. It does NOT start program execution.

After reset, program execution must be started as follows by either a boot block or a host processor:

- write the desired frequency ratio and the code for ‘enable pll’ and ‘stop and reset’ to the TM32_CTL register
- delay for approx. 10 uSec to allow the internal CPU PLL to lock and stabilize
- configure DRAM, MMIO, and Aperture1 apertures
- write the desired TM32 CPU start address to the TM32_START_ADDR MMIO register
- write the code for ’start’ and the same frequency ratio and ‘enable pll’ to the TM32_CTL MMIO regis-

© TriMedia Technologies, Inc.

ter.

Execution then starts at the address contained in the TM32_START_ADDR MMIO register. TM32_START_ADDR is 64 byte (instruction cache line) aligned and is read only for TM32 itself while being read/write for external PI-masters. On reset, TM32_START_ADR is set to 0x0000,0000 and TM32_APERT1_HI to 0x2000,0000.

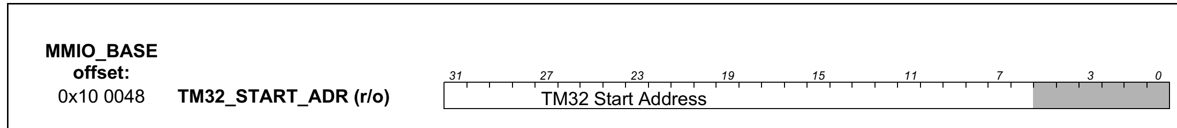


Figure 1-8 TM32 Start Address

1.5.2 EXC (Exceptions)

The DSPCPU32 enters EXC special-event processing under the following conditions:

1. RESET is de-asserted.
2. The intersection PCSW[15,6:0] & PCSW[31,22:16] is non-empty or PCSW.TFE is set.
3. A successful interruptible jump is in the final jump execution stage.

DSPCPU32 hardware takes the following actions on the initiation of EXC processing:

1. DPC is assigned the intended destination address of the successful jump.
2. Instruction processing starts at EXCVEC.

All other actions are the responsibility of the EXC handler software. Note that no other special event processing will take place until the handler decides to execute an interruptible jump that succeeds.

1.5.3 INT and NMI (Maskable and Non-Maskable Interrupts)

The on-chip Vectored Interrupt Controller (VIC) provides 32 INT request input hardware lines. The interrupt controller prioritizes and maps attention requests from several different peripherals onto successive INT requests to the DSPCPU32.

INT special event processing will occur under the following conditions:

1. RESET is de-asserted.
2. The intersection PCSW[15,6:0] & PCSW[31,22:16] is empty and PCSW.TFE is not set.
3. The intersection of IPENDING and IMASK is non-empty.
4. The interrupt is at level NMI or PCSW.IEN = 1.
5. A successful interruptible jump is in the final jump execution stage.

DSPCPU32 hardware takes the following actions on the initiation of NMI or INT processing:

1. DPC gets assigned the intended destination address of the successful jump.
2. Instruction processing starts at the appropriate interrupt vector.

All other actions are the responsibility of the INT handler software. Note that no other special event processing will take place until the handler decides to execute an interruptible jump that succeeds.

1.5.3.1 Interrupt vectors

Each of the 32 interrupt sources can be assigned an arbitrary interrupt vector (the address of the first instruction of the interrupt handler). A vector is setup by writing the address to one of the MMIO locations

shown in Figure 1-9. The state of the MMIO vector locations is undefined after RESET. (Addresses of the MMIO vector registers are offset with respect to MMIO_BASE.)

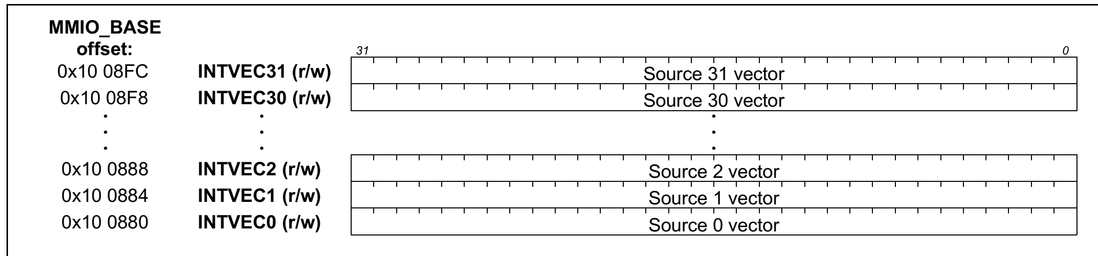


Figure 1-9 Interrupt vector locations in MMIO address space.

Note: Programmers, see the *TriMedia Cookbook* (Book 2 of TriMedia SDE documentation) for information on writing interrupt handlers.

1.5.3.2 Interrupt modes

DSPCPU32 interrupt sources can be programmed to operate in either *level-sensitive* or *edge-triggered* mode. Operation in edge-triggered or level-sensitive mode is determined by a bit in the ISETTING MMIO locations corresponding to the source, as defined in Figure 1-10. On RESET, all ISETTING registers are cleared.

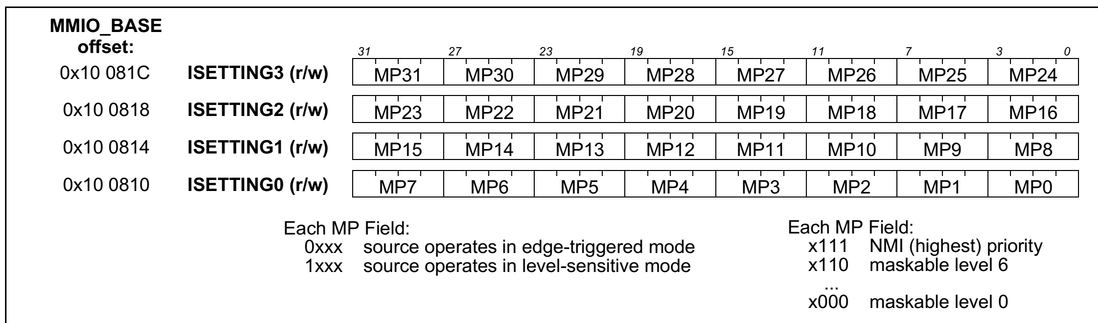


Figure 1-10 Interrupt mode and priority MMIO locations and formats.

In edge-triggered mode, the leading edge of the signal on the device interrupt request line causes the VIC (Vectored Interrupt Controller) to set the *interrupt pending* flag corresponding to the device source number. Note that, for active high signals, the leading edge is the positive edge, whereas for active low request signals (such as PCI INTA#), the negative edge is the leading edge. The interrupt remains pending until one of two events occurs:

- The VIC successfully dispatches the vector corresponding to the source to the TM32 CPU, or
- TM32 CPU software clears the interrupt-pending flag by a direct write to the ICLEAR location.

No interrupt acknowledge to ICLEAR is needed for devices operating in edge-triggered mode, since the vector dispatch clears the IPENDING request. The device itself may however need a device-specific interrupt acknowledge to clear the requesting condition. Edge-triggered mode is *not recommended* for devices that can signal multiple simultaneous interrupt conditions. The on-chip timers *must* be operated in edge triggered mode.

In level-sensitive mode, the device requests an interrupt by asserting the VIC source request line. The device holds the request until the device interrupt handler performs a device interrupt acknowledge. It is *highly recommended* that all off-chip and on-chip sources, with the exception of the timers, operate in level-sensitive mode.

1.5.3.3 Device interrupt acknowledge

All devices capable of generating level-triggered interrupts have interrupt acknowledge bits in their memory mapped control registers for this purpose. An interrupt acknowledge is performed by a store to such control register, with a ‘1’ in the bit position(s) corresponding to the desired acknowledge flags.

Programmers note: the store operation that performs the interrupt acknowledge should be issued at least 2 cycles before the (interruptible) jump that ends an interrupt handler. This ensures that the same interrupt is not dispatched twice due to request de-assertion clock delays.

1.5.3.4 Interrupt priorities

Each interrupt source can be programmed to request one out of eight levels of priorities. The highest priority level (level 7) corresponds to requesting an NMI—an interrupt that cannot be masked by the DSPCPU32 PCSW.IEN bit. The other levels request regular interrupts, that can be masked as a group by the PCSW.IEN flag. Level six represents the highest priority normal interrupt level and level zero represents the lowest. Refer to Figure 1-10 for details of programming the priority level.

The VIC arbitrates the highest-priority pending interrupt requestor. Sources programmed to request at the same level are treated with a fixed priority, from source number 0 (highest) to 31 (lowest). At such time as the DSPCPU32 is willing to process special events, the vector of highest priority NMI source will be dispatched. If no NMI is pending, and the DSPCPU32 allows regular interrupts (PCSW.IEN is asserted), the vector of the highest priority regular source is dispatched. Once a vector is dispatched, the corresponding interrupt pending flag is de-asserted (edge triggered mode sources only).

1.5.3.5 Interrupt masking

A single MMIO register (IMASK in Figure 1-11) allows masking of an arbitrary subset of the interrupt sources. Masking applies to both regular as well as NMI level requestors. Masking is used by software to disable unused devices and/or to implement nested interrupt handling. In the latter case, each interrupt handler can stack the old IMASK content for later restoration and insert a new mask that only allows the interrupts it is willing to handle. For level-triggered device handlers, IMASK should also exclude the device itself to prevent repeated handler activation.

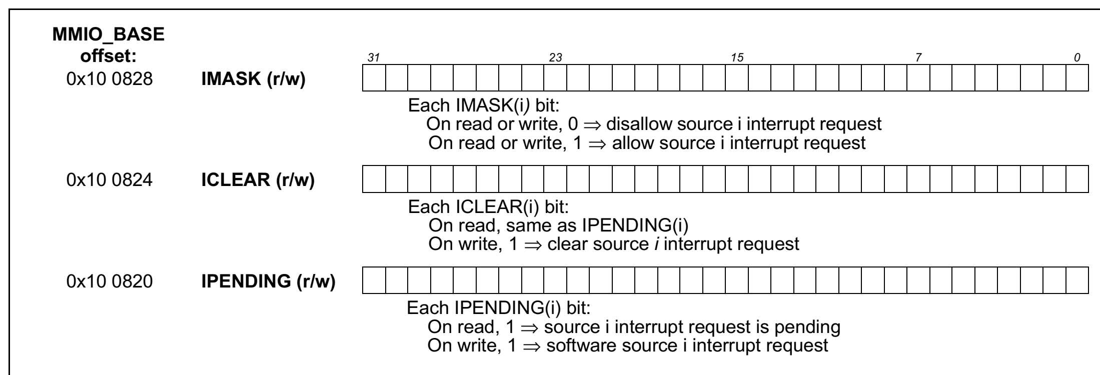


Figure 1-11 Interrupt controller request, clear, and mask MMIO

Each interrupt source device typically has its own interrupt enable flag(s) that determine whether certain key device events lead to the request of an interrupt. In addition, the PCSW.IEN flag determines whether the DSPCPU32 is willing to handle regular interrupts. Non maskable interrupts ignore the state of this flag.

All three mechanisms are necessary: the PCSW.IEN flag is used to implement critical sections of code during which the RTOS (real-time operating system) is unable to handle regular interrupts. The IMASK is

used to allow full control over interrupt handler nesting. The device interrupt flags set the operational mode of the device.

When RESET is asserted, IPENDING, ICLEAR, and IMASK are set to all zeroes. (MMIO register addresses shown in Figure 1-11 are offset addresses with respect to MMIO_BASE.)

1.5.3.6 Software interrupts and acknowledgment

The IPENDING register shown in Figure 1-11 can be read to observe the currently pending interrupts. Each bit read depends on the mode of the source:

- For a level-sensitive source, a bit value corresponds to the current state of the device interrupt request line.
- For an edge-triggered interrupt, a '1' is read if and only if an interrupt request occurred and the corresponding vector has not yet been dispatched.

Software can request an interrupt for sources operating in edge-triggered mode. Writes to the IPENDING register assert an interrupt request for all sources where a 1 occurred in the bit position of the written value. The state of sources where a 0 occurred in the written value is unchanged. Writes have no effect on level-sensitive mode sources. The interrupt request, if not masked, will occur at the next successful interruptible jump. This differs from the conventional software interrupt-like semantics of many architectures. Any of the 32 sources can be requested in software. In normal operation however, software-requested interrupts should be limited to source vectors not allocated for hardware devices. Note that another PCI master can request interrupts by manipulating the IPENDING location in the MMIO aperture. This is useful for interprocessor communication.

The ICLEAR register reads the same as the IPENDING register. Writes to the ICLEAR register serve to clear pending flags for edge-triggered mode sources. All IPENDING flags corresponding to bit positions in which '1's are written are cleared. IPENDING flags corresponding to bit positions in which '0's are written are not affected. Writes have no effect on level-sensitive mode sources. When a pending interrupt bit is being cleared through a write to the ICLEAR register at the same time that the hardware is trying to set that interrupt bit, the hardware takes precedence.

1.5.3.7 NMI sequentialization

In most applications, it is desirable not to nest NMIs. The NMI interrupt handler can accomplish this by saving the old IMASK content and clearing IMASK before the first interruptible jump is executed by the NMI handler.

Table 1-10 Interrupt source assignments

SOURCE NAME	SRC NUM	MODE	SOURCE DESCRIPTION
INTREQ0..4	0..4	either	generic interrupt request
TIMER1	5	edge	general-purpose timer
TIMER2	6	edge	general-purpose timer
TIMER3	7	edge	general-purpose timer
SYSTIMER	8	edge	reserved for debugger
INTREQ9..31	9..31	either	generic interrupt request

 © TriMedia Technologies, Inc.

1.5.3.8 Interrupt source assignment

Table 1-10 shows the assignment of devices to interrupt source numbers, as well as the recommended operating mode (edge or level triggered).

1.6 Timers

The DSPCPU32 contains four programmable timer/counters, all with the same function. The first three (TIMER1, TIMER2, TIMER3) are intended for general use. The fourth timer/counter (SYSTIMER) is reserved for use by the system software and should not be used by applications.

Each timer has three registers as shown in Figure 1-12. The MMIO register addresses shown are offset addresses with respect to the timer's base address.

Table 1-11 Timer base MMIO address

TIMER1	MMIO_BASE+0x10,0C00
TIMER2	MMIO_BASE+0x10,0C20
TIMER3	MMIO_BASE+0x10,0C40
SYSTIMER	MMIO_BASE+0x10,0C60

Table 1-12 Timer source selections

Source Name	Source Bits Value	Source Description
CLOCK	0	CPU clock
PRESCALE	1	prescaled CPU clock
TRI_TIMER_CLK	2	external clock pin
DATABREAK	3	data breakpoints
INSTBREAK	4	instruction breakpoints
CACHE1	5	cache event 1
CACHE2	6	cache event 2
EXT0	7	external pin0
EXT1	8	external pin1
EXT2	9	external pin2
EXT3	10	external pin3
EXT4	11	external pin4
EXT5	12	external pin5
—	13-15	undefined

Each timer/counter can be set to count one of the event types specified in Table 1-12. Note that the DATABREAK event is special, in that the timer/counter may increment by zero, one or two in each clock cycle. For all other event types, increments are by zero or one. The CACHE1 and CACHE2 events serve as cache performance monitoring support. The actual event selected for CACHE1 and CACHE2 is deter-

mined by the MEM_EVENTS MMIO register, see Section 3.7. If a TM32 pin signal (VI-CLK, etc.) is selected as an event, positive-going edges on the signal are counted.

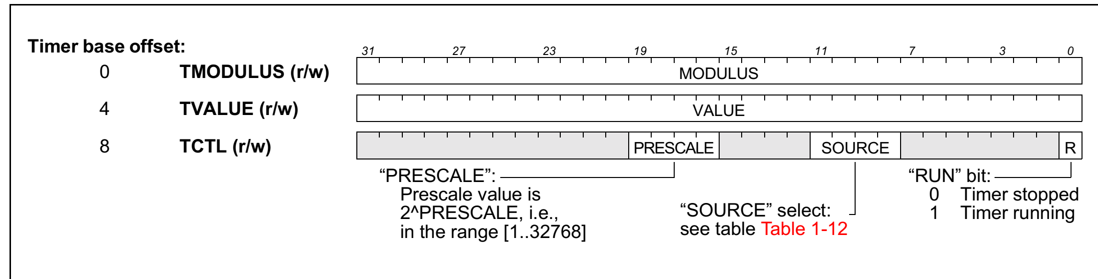


Figure 1-12 Timer register definitions.

Each timer increments its value until the modulus is reached. On the clock cycle where the incremented value would equal or exceed the modulus, the value wraps around to zero or one (in the case of an increment by two), and an interrupt is generated as defined in Table 1-10. The timer interrupt source mode should be set as edge-sensitive. No software interrupt acknowledge to the timer device is necessary.

Counting starts and continues as long as the run bit is set.

Loading a new modulus does not affect the contents of the value register. If a store operation to either the modulus or value register results in value and modulus being the same, no interrupt will be generated. If the run bit is set, the next value will be modulus+1 or modulus+2, and the counter will have to loop around before an interrupt is generated.

A modulus value of zero causes a wrap-around as if the modulus value was 2^{32}.

On RESET, the TCTL registers are cleared, and the value of the TMODULUS and TVALUE registers is undefined.

1.7 Debug Support

This section describes the special debug support offered by the DSPCPU32. Instruction and data breakpoints can be defined through a set of registers in the MMIO register space. When a breakpoint is matched, an event is generated that can be used as a timer source (see Section 1.6). The timer TMODULUS has to be set to generate a DSPCPU32 interrupt after the desired number of breakpoint matches.

1.7.1 Instruction Breakpoints

The instruction-breakpoint control register is shown in Figure 1-13. On RESET, the BICTL register is cleared. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

The instruction-breakpoint address-range registers are shown in Figure 1-14. After RESET, the value of these registers is undefined. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

When the IC bit in the breakpoint control register is set to '1', instruction breakpoints are activated. Any instruction address issued by the TM32 core is compared against the low and high address-range values. The IAC bit in the breakpoint control register determines whether the instruction address needs to be inside or outside of the range defined by the low and high address-range registers. A successful comparison takes place when either:

 © TriMedia Technologies, Inc.

- IAC = ‘0’ and low ≤ iaddr ≤ high, or
- IAC = ‘1’ and iaddr < low or iaddr > high.

On a successful comparison, an instruction breakpoint event is generated, which can be used as a clock input to a timer. After counting the programmed number of instruction breakpoint events, the timer will generate an interrupt request.

Figure 1-13 Instruction-breakpoint control register.

Figure 1-14 Instruction-breakpoint address-range registers.

Figure 1-15 Data-breakpoint address-range and value-compare

1.7.2 Data Breakpoints

The data-breakpoint address-range and compare-value registers are shown in Figure 1-15. After RESET, the value of the data breakpoint registers is undefined. (MMIO-register addresses shown are offset with respect to MMIO_BASE.)

The data-breakpoint control register is shown in Figure 1-16. On RESET, the BDCTL register is cleared. (The register address shown is offset with respect to MMIO_BASE.)

Figure 1-16 Data-breakpoint control register.

When the DC bits in the data breakpoint control register are not set to ‘0’, data breakpoints are activated. When the value of the DC bits is ‘1’ or ‘3’, any data address from load operations (if the BL bit is set) and/or store operations (if the BS bit is set) issued by the DSPCPU32 is compared against the low and high address-range values. The DAC bit in the breakpoint control register determines whether data addresses need to be inside or outside of the range defined by the low and high address-range registers. A successful comparison occurs when either:

- DAC = ‘0’ and low ≤ daddr ≤ high, or
- DAC = ‘1’ and daddr < low or daddr > high.

Note that this comparison works for all addresses regardless of the aperture to which they belong. When the value of the DC bits is ‘2’ or ‘3’, any data value from load operations (if the BL bit is set) and/or store operations (if the BS bit is set) issued by the TM32 CPU is compared against the value in the BDATAVAL register. Only the bits for which the corresponding BDATAMASK register bits are set to ‘1’ will be used in the comparison. The DVC bit in the breakpoint control register determines whether the data value needs to be equal or not equal to the comparison value. A successful comparison occurs when either of the following are true:

- DVC = ‘0’ and (data & BDATAMASK) = (BDATAVAL & BDATAMASK).
- DVC = ‘1’ and (data & BDATAMASK) != (BDATAVAL & BDATAMASK).

Note: Use a nonzero datamask or the result is undefined.

When a successful comparison has taken place, a data breakpoint event is generated, which can be used as a clock input to a timer. After counting the set number of data breakpoint events, the timer will generate an interrupt request.

When the value of the DC bits is ‘3’, a data breakpoint event is generated if and only if a successful comparison occurs on both address and data simultaneously.

Note: Up to two data breakpoint events can occur per clock cycle, due to the dual load/store capability of the CPU and data cache.

1.8 System

This section describes TM32 system issues.

1.8.1 TM32_CTL

The TM32_CTL (MMIO offset 0x10,0030) register allows a boot block or a host processor to set the TM32 operating frequency and to start and (software) reset TM32 (see Section 1.5.1).

 © TriMedia Technologies, Inc.

TM32_CTL is read only for TM32 itself while being read/write for external PI-masters. On reset, TM32_CTL.PLL_EN is set to 0, while TM32_CTL.RATIO is to 0x09 (1.0x). All other bits are undefined after reset.

Table 1-13 TM32_CTL

Bits	Default	Name	Description
31:30	n/a	ACTION	00: no action 01: stop and reset TM32 CPU immediately 10: start TM32 CPU (execution starts at TM32_START_ADR) 11: RESERVED
6	0x1	PLL_EN	1: PLL is generating the TM32 CPU internal clock 0: PLL disabled. Take TM32 CPU internal clock directly from SDRAM_CLK
5:0	0x09	RATIO	TM32 CPU speed to TM32_CLK clock ratio 001001: 1.0[a] 101100: 1.25 100011: 1.3333333 011010: 1.50 101011: 1.6666666 111100: 1.75 010001: 2.0 all else: RESERVED

a. In 1.0 RATIO, PLL_EN should be set to 0, i.e. the internal PLL should NOT be enabled !

1.8.2 TM32_MODID

The read only TM32_MODID register (MMIO offset 0x10,0ffc) provides a unique ID to allow software to identify TM32 and identify the version number.

Table 1-14 TM32_CTL

Bits	Default	Name	Description
31:16	0x2b80	module	designates TM32
15:12	0x0	major rev	major revision 0
11:8	0x0	minor rev	minor revision 0 (metal update)
7:0	0x1	ap size	aperture size = 8k

1.8.3 Endianness

TM32 supports both little-endian and big-endian mode of operation. Endian mode is solely determined by the PCSW BSX bit. Program initiated switching of endian-ness (by writing a new PCSW value) is supported. Endianness only affects how data is loaded/stored - TM32 instructions are coded in an endianness invariant manner.

The 64 bit TrIP highway uses address invariant rules: the 8 lsb's of the bus are associated with the byte with address 'A' (A dividable by 8), the 8 msb's of the bus are associated with address 'A+7'. Byte lane enables are used to implement 8, 16 or 32 bit data transactions, but bytes travel over the lane associated with their memory byte address. This convention holds irrespective of system endianness.

The PI bus has a signal BIGENDIAN, which indicates to all attached blocks which endianness to use. The transfers on the bus always use right hand side (lsb) alignment - a 32 bit load/store value travels over all 32 wires. A 16 bit value travels over the 16 lsb wires, a 8 bit value travels over the 8 lsb wires. For each transfer size, the value travels with the msb on the left and the lsb on the right.

TM32 has been designed to support dynamic endian mode switching, however in normal use software immediately sets it to operate in the same endian mode as the PI-bus, and does not change it thereafter.

1.9 Powerdown

1.9.1 Partial Powerdown

Partial powerdown is activated by any ‘store’ to the POWER_DOWN MMIO register at MMIO offset 0x10,0108.

During partial powerdown, the CCCOUNT register and the TM32 timers (TIMER1,2,3 and SYSTIMER) continue operation.

Any enabled interrupt request, or any incoming PI-bus access will wake TM32 back up, and execution continues where it left off. The PI-bus transaction value/effect will not be affected by the powerdown.

The partial power down mechanism is intended to be activated by TM32 itself and to be used by the idle loop, or idle task, of the real-time kernel. TM32 should be powered down whenever possible, to reduce dissipation to the minimum required for the application.

Table 1-15 Timing of Partial Powerdown

Event	Typical Time (TM32 clock)
powerdown store	6 cycles to activity halt
interrupt assertion	9 cycles to wakeup
PI bus read or write	10 cycles to wakeup

1.9.2 Full Powerdown

Assertion of the external input signal ‘tm32_pwrdwn_req’ causes the TM32 to finish any outstanding bus transactions. It then shuts down all internal clocks and the internal PLL, and asserts ‘tm32_pwrdwn_ack’.

During full powerdown, the CCOUNT register and TM32 timers are not counting.

During full powerdown, the core will not grant access to its registers for incoming PI transactions. Instead, it returns a bogus value on read and acknowledges, but ignores PI-bus writes. PI-bus transactions do not cause wakeup.

Wakeup is accomplished by de-asserting ‘tm32_pwrdwn_req’. This causes the TM32 to re-start its internal PLL, and then re-start execution where it left off. Once started, the core de-asserts ‘tm32_pwrdwn_ack’. At that point the core will again handle incoming PI-bus transactions.

The time from a full powerdown request till powerdown complete depends on ongoing transaction status at the time of the request. If no transactions are ongoing, the powerdown is acknowledged in a few clock cycles. It may however be the case that a complex transaction, that needs to timeout, may be in progress, e.g. due to a speculative load. Extensive simulation of such cases has shown that the acknowledge does occasionally take more than 512 TM32 CPU clock cycles, but never more than 1024 TM32 CPU clocks.

Wakeup, from de-asserting the request till the acknowledge is de-asserted, takes several hundred TM32 CPU clocks, due to the need to re-start and stabilize the CPU PLL.

© TriMedia Technologies, Inc.

Chapter 2 Custom Operations for Multimedia

2.1 Custom Operations Overview

Custom operations in the TM32 CPU (or DSPCPU32) architecture are specialized, high-function operations designed to dramatically improve performance in important multimedia applications. When properly incorporated into application source code, custom operations enable an application to take advantage of the highly parallel TM32 microprocessor implementation. Achieving a similar performance increase through other means—e.g., executing a higher number of traditional microprocessor instructions per cycle—would be prohibitively expensive for TM32's low-cost target applications.

Custom operations are simple to understand and consistent in their definition, but their unusual functions make it difficult for automatic code generation algorithms to use them effectively. Consequently, custom operations are inserted into source code by the programmer. To make this process as painless as possible, custom operation syntax is consistent with the C programming language, and, just as with all other operations generated by the compiler, the scheduler takes care of register allocation, operation packing, and flow analysis.

2.1.1 Custom Operation Motivation

For both general-purpose and embedded microprocessor-based applications, programming in a high-level language is desirable. To effectively support optimizing compilers and a simple programming model, certain microprocessor architecture features are needed, such as a large, linear address space, general-purpose registers, and register-to-register operations that directly support the manipulation of linear address pointers. A common choice in microprocessor architectures is 32-bit linear addresses, 32-bit registers, and 32-bit integer operations. The TM32 is such a microprocessor architecture.

For the data manipulation in many algorithms, however, 32-bit data and operations are wasteful of expensive silicon resources. Important multimedia applications, such as the decompression of MPEG video streams, spend significant amounts of execution time dealing with eight-bit data items. Using 32-bit operations to manipulate small data items makes inefficient use of 32-bit execution hardware in the implementation. If these 32-bit resources could be used instead to operate on four eight-bit data items simultaneously, performance would be improved by a significant factor with only a tiny increase in implementation cost.

Getting the highest execution rate from standard microprocessor resources is one of the motivations behind custom operations in the TM32. A range of custom operations is provided that each processes—simultaneously—four 8-bit or two 16-bit data items. There is little cost difference between a standard 32-bit ALU and one that can process either one pair of 32-bit operands or four pairs of eight-bit operands, but there is a big performance difference for TM32's target applications.

TM32's custom operations go beyond simply making the best use of standard resources. Some custom operations combine several simple operations. These combinations are tailored specifically to the needs of important multimedia applications. Some high-function custom operations eliminate conditional branches, which helps the scheduler make effective use of all five operation slots in each TM32 instruction. Filling up all five slots is especially important in the inner loops of computational intensive multimedia applications.

In short, custom operations help TM32 reach its goals of extremely high multimedia performance at the lowest possible cost.

2.1.2 Introduction to Custom Operations

Table 2-1 and Table 2-2 contain two listings of the custom operations available in the TM32 architecture. Table 2-1 groups the custom operations by type of function while Table 2-2 lists the operations by operand size. For more detailed information about the custom operations, see Appendix A.

Some operations exist in several versions that differ in the treatment of their operands and results, and the mnemonics for these versions make it easy to select the appropriate operation. For example, the sum of products operations all have "fir" in their mnemonics; the prefix and suffix of the mnemonic expresses the treatment of the operands and result. The ifir8ii operation treats both of its operands as signed (ifir8ii) and produces a signed result (ifir8ii). The ifir8iu operation treats its first operand as signed (ifir8iu), the second as unsigned (ifir8iu), and produces a signed result (ifir8iu). The ume8ii operation implements an eight-bit motion-estimation; it treats both operands as signed but produces an unsigned result.

Table 2-1 Key Multimedia Custom Operations Listed by Function Type

Function	Custom Op	Description
DSP absolute value	dspiabs	Clipped signed 32-bit absolute value
	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
Shift	dualasr	dual-16 arithmetic shift right
Clip	dualiclipi	dual-16 clip signed to signed
	dualuclipi	dual-16 clip signed to unsigned
Min,max	quadumax	Unsigned bytewise quad max
	quadumin	Unsigned bytewise quad min
DSP add	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspuquadaddui	Quad clipped add of unsigned/signed bytes
DSP multiply	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords

 © TriMedia Technologies, Inc.

Table 2-1 Key Multimedia Custom Operations Listed by Function Type

Function	Custom Op	Description
DSP subtract	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords
Sum of products	ifir16	Signed sum of products of signed 16-bit halfwords
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	ufir8uu	Unsigned sum of products of unsigned bytes
Merge, pack	mergedual16lsb	Merge dual-16 least-significant bytes
	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords
	packbytes	Pack least-significant bytes
Byte averages	quadavg	Unsigned byte-wise quad average
Byte multiplies	quadumulmsb	Unsigned quad 8-bit multiply most significant
Motion estimation	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences

The operations beginning with “dsp” implement a clipping (sometimes called saturating) function before storing the result(s) in the destination register. Otherwise, their naming follows the rules given above where appropriate. For example, the dspuquadaddui operation implements four 8-bit additions; it treats the first operand of each addition as unsigned, the second operand as signed, and produces an unsigned result for each addition. Each result, which is computed with no loss of precision, is clipped into the representable range of a byte (0..255).

Table 2-2 Key Multimedia Custom Operations Listed by Operand Size

Op. Size	Custom Op	Description
32-bit	dspiabs	Clipped signed 32-bit abs value
	dspiadd	Clipped signed 32-bit add
	dspuadd	Clipped unsigned 32-bit add
	dspimul	Clipped signed 32-bit multiply
	dspumul	Clipped unsigned 32-bit multiply
	dspisub	Clipped signed 32-bit subtract
	dspusub	Clipped unsigned 32-bit subtract

Table 2-2 Key Multimedia Custom Operations Listed by Operand Size

Op. Size	Custom Op	Description
16-bit	mergedual16lsb	Merge dual-16 least-significant bytes
	dualasr	dual-16 arithmetic shift right
	dualiclipi	dual-16 clip signed to signed
	dualuclipi	dual-16 clip signed to unsigned
	dspidualmul	Dual clipped multiply of signed 16-bit halfwords
	dspidualabs	Dual clipped absolute values of signed 16-bit halfwords
	dspidualadd	Dual clipped add of signed 16-bit halfwords
	dspidualsub	Dual clipped subtract of signed 16-bit halfwords
	ifir16	Signed sum of products of signed 16-bit halfwords
	ufir16	Unsigned sum of products of unsigned 16-bit halfwords
	pack16lsb	Pack least-significant 16-bit halfwords
	pack16msb	Pack most-significant 16-bit halfwords
8-bit	quadumax	Unsigned bytewise quad max
	quadumin	Unsigned bytewise quad min
	dspuquadaddui	Quad clipped add of unsigned/signed bytes
	ifir8ii	Signed sum of products of signed bytes
	ifir8iu	Signed sum of products of signed/unsigned bytes
	ufir8uu	Unsigned sum of products of unsigned bytes
	mergelsb	Merge least-significant bytes
	mergemsb	Merge most-significant bytes
	packbytes	Pack least-significant bytes
	quadavg	Unsigned byte-wise quad average
	quadumulmsb	Unsigned quad 8-bit multiply most significant
	ume8ii	Unsigned sum of absolute values of signed 8-bit differences
	ume8uu	Unsigned sum of absolute values of unsigned 8-bit differences

2.1.3 Example Uses of Custom Ops

The next three sections illustrate the advantages of using custom operations. Also, the more complex examples illustrate how custom operations can be integrated into application code by providing listings of C-language program fragments. The examples progress in complexity from simple to intricate; the most interesting examples are taken from actual multimedia codes, such as MPEG decompression.

2.2 Example 1: Byte-Matrix Transposition

The goal of this example is to provide a simple introductory illustration of how custom operations can significantly increase processing speed in small kernels of applications. As in most uses of custom operations,

the power of custom operations in this case comes from their ability to operate on multiple data items in parallel.

Imagine that our task is to transpose a packed, 4-by-4 matrix of bytes in memory; the matrix might, for example, contain 8-bit pixel values. Figure 2-1 illustrates both the organization of the matrix in memory and the task to be performed in standard mathematical notation.

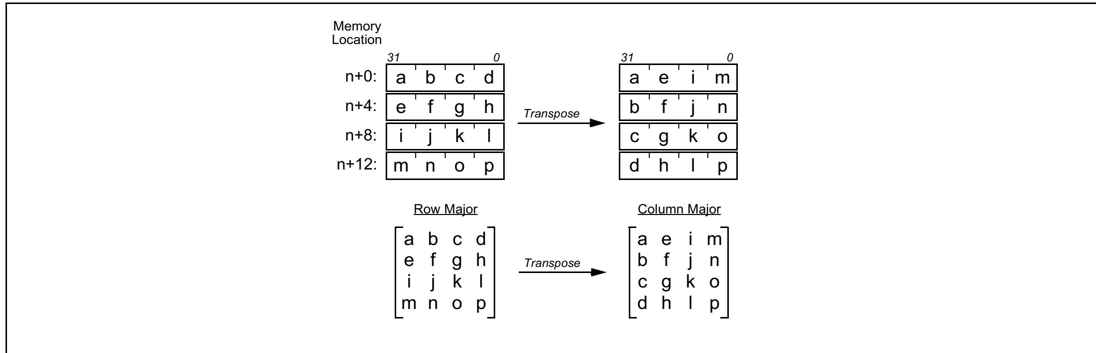


Figure 2-1 Byte-matrix transposition. Top shows byte matrices packed into memory words; bottom shows mathematical matrix representation.

Performing this operation with traditional microprocessor instructions is straight-forward but time consuming. One way to perform the manipulation is to perform 12 load-byte instructions (since only 12 of the 16 bytes need to be repositioned) and 12 store-byte instructions that place the bytes back in memory in their new positions. Another way would be to perform four load-word instructions, reposition the bytes in registers, and then perform four store-word instructions. Unfortunately, repositioning the bytes in registers would require a large number of instructions to properly shift and mask the bytes. Performing the 24 loads and stores makes implicit use of the shifting and masking hardware in the load/store units and thus yields a shorter instruction sequence.

The problem with performing 24 loads and stores is that loads and stores are inherently slow operations because they must access at least the cache and possibly slower layers in the memory hierarchy. Further, performing byte loads and stores when 32-bit word-wide accesses run just as fast wastes the power of the cache/memory interface. We would prefer a fast algorithm that takes full advantage of cache/memory bandwidth while not requiring an inordinate number of byte-manipulation instructions.

The TM32 has instructions that merge and pack bytes and 16-bit halfwords directly and in parallel. Four of these instructions can be applied in this case to speed up the manipulation of bytes that are packed into words.

Figure 2-2 shows the application of these instructions to the byte-matrix transposition problem, and the left side of Figure 2-3 shows a list of the operations needed to implement the matrix transpose. When assembled into actual TM32 instructions, these custom operations would be packed as tightly as dependencies allow, up to five operations per instruction.

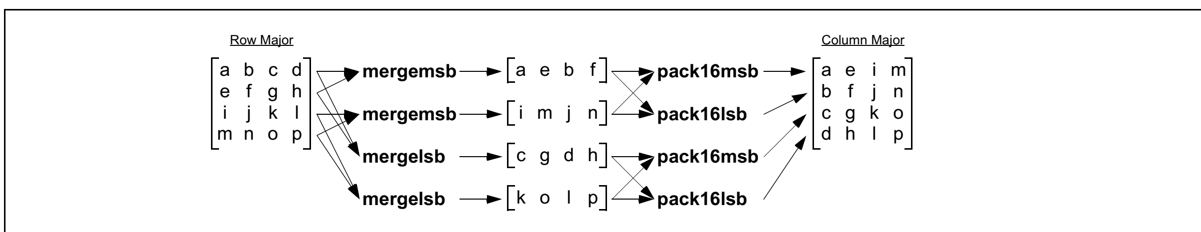


Figure 2-2 Application of merge and pack instructions to the byte-matrix transposition of Figure 2-1.

```
ld32d(0) r100 → r10
ld32d(4) r100 → r11
ld32d(8) r100 → r12
ld32d(12) r100 → r13

mergemsb r10 r11 → r14
mergemsb r12 r13 → r15
mergelsb r10 r11 → r16
mergelsb r12 r13 → r17
pack16msb r14 r15 → r18
pack16lsb r14 r15 → r19
pack16msb r16 r17 → r20
pack16lsb r16 r17 → r21

st32d(0) r101 r18
st32d(4) r101 r19
st32d(8) r101 r20
st32d(12) r101 r21
```

```
char matrix[4][4];
        .
        .
        .
int *m = (int *) matrix;

temp0 = MERGEMSB(m[0], m[1]);
temp1 = MERGEMSB(m[2], m[3]);
temp2 = MERGELSB(m[0], m[1]);
temp3 = MERGELSB(m[2], m[3]);
m[0]  = PACK16MSB(temp0, temp1);
m[1]  = PACK16LSB(temp0, temp1);
m[2]  = PACK16MSB(temp2, temp3);
m[3]  = PACK16LSB(temp2, temp3);
        .
        .
        .
```

Figure 2-3 On the left is a complete list of operations to perform the byte-matrix transposition of Figure 2-1 and Figure 2-2. On the right is an equivalent C-language fragment.

Note that a programmer would not need to program at this level (TM32 assembler). The matrix transpose would be expressed just as efficiently in C-language source code, as shown on the right side of Figure 2-3. The low-level code is shown here for illustration purposes only.

The first sequence of four load-word operations in Figure 2-3 brings the packed words of the input matrix into registers R10, R11, R12, and R13. The next sequence of four merge operations produces intermediate results into registers R14, R15, R16, and R17. The next sequence of four pack operations could then replace the original operands or place the transposed matrix in separate registers if the original matrix operands were needed for further computations (the TM32 optimizing C compiler performs this analysis automatically). In this example, the transpose matrix is placed in registers R18, R19, R20, and R21. The final four store-word operations put the transposed matrix back into memory.

Thus, using the TM32 custom operations, the byte-matrix transposition requires four load-word operations and four store-word operations (the minimum possible) and eight register-to-register data-manipulation operations. The result is 16 operations, or byte-matrix transposition at the rate of one operation per byte.

While the advantage of the custom-operation-based algorithm over the brute-force code that uses 24 load- and store-byte instruction seems to be only eight operations (a 33% reduction), the advantage is actually much greater. First, using custom operations, the number of memory references is reduced from 24 to eight (a factor of three). Since memory references are slower than register-to-register operations (such as the custom operations in this example), the reduction in memory references is significant.

Further, the ability of the TM32 VLIW compilation system to exploit the performance potential of the TM32 microprocessor hardware is enhanced by the custom-operation-based code. This is because it is easier for the compilation system to produce an optimal schedule (arrangement) of the code when the number of memory references is in balance with the number of register-to-register operations. The TM32 CPU (like all high-performance microprocessors) has a limit on the number of memory references that can be processed in a single cycle (two is the current limit). A long sequence of code that contains only memory references can result in empty operation slots in the long TM32 instructions. Empty operation slots waste the performance potential of the TM32 hardware.

As this example has shown, careful use of custom operations has the potential to not only reduce the absolute number of operations needed to perform a computation but can also help the compilation system produce code that fully exploits the performance potential of the TM32 CPU.

2.3 Example 2: MPEG Image Reconstruction

The complete MPEG video decoding algorithm is composed of many different phases, each with computational intensive kernels. One important kernel deals with reconstructing a single image frame given that the forward- and backward-predicted frames and the inverse discrete cosine transform (IDCT) results have already been computed. This kernel provides an excellent opportunity to illustrate of the power of TM32's specialized custom operators.

In the code fragments that follow, the backward-predicted block is assumed to have been computed into an array back[], the forward-predicted block is assumed to have been computed into forward[], and the IDCT results are assumed to have been computed into idct[].

A straightforward coding of the reconstruction algorithm might look as shown in Figure 2-4. This implementation shares many of the undesirable properties of the first example of byte-matrix transposition. The code accesses memory a byte at a time instead of a word at a time, which wastes 75% of the available bandwidth. Also, in light of the many quad-byte-parallel operations introduced in Section 2.1.2 it seems inefficient to spend three separate additions and one shift to process a single eight-bit pixel. Perhaps even more unfortunate for a VLIW processor like TM32 is the branch-intensive code that performs the saturation testing; eliminating these branches could reap a significant performance gain.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 1)
    {
        temp = ((back[i] + forward[i] + 1) >> 1) + idct[i];

        if (temp > 255)
            temp = 255;
        else if (temp < 0)
            temp = 0;

        destination[i] = temp;
    }
}
```

Figure 2-4 Straightforward code for MPEG frame reconstruction.

Since MPEG decoding is the kind of task for which the TM32 was created, there are two custom operations—quadavg and dspuquadaddui—that exactly fit this important MPEG kernel (and other kernels). These custom operations process four pairs of 8-bit pixel values in parallel. In addition, dspuquadaddui performs saturation tests in hardware, which eliminates any need to execute explicit tests and branches.

For readers familiar with the details of MPEG algorithms, the use of eight-bit IDCT values later in this example may be confusing. The standard MPEG implementation calls for nine-bit IDCT values, but extensive analysis has shown that values outside the range [–128..127] occur so rarely that they can be considered unimportant. Pursuant to this observation, the IDCT values are clipped into the eight-bit range [–128..127] with saturating arithmetic before the frame reconstruction code runs. The assumption that this saturation occurs permits some of TM32's custom operations to have clean, simple definitions.

The first step in seeing how custom operations can be of value in this case, is to unroll the loop by a factor of four. The unrolled code is shown in Figure 2-5. This creates code that is parallel with respect to the four

pixel computations. As it is easily seen in the code, the four groups of computations (one group per pixel) do not depend on each other.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    for (i = 0; i < 64; i += 4)
    {
        temp = ((back[i+0] + forward[i+0] + 1) >> 1) + idct[i+0];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+0] = temp;

        temp = ((back[i+1] + forward[i+1] + 1) >> 1) + idct[i+1];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+1] = temp;

        temp = ((back[i+2] + forward[i+2] + 1) >> 1) + idct[i+2];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+2] = temp;

        temp = ((back[i+3] + forward[i+3] + 1) >> 1) + idct[i+3];
        if (temp > 255) temp = 255;
        else if (temp < 0) temp = 0;
        destination[i+3] = temp;
    }
}
```

Figure 2-5 MPEG frame reconstruction code using TM32 custom operations; compare with Figure 2-4.

After some experience is gained with custom operations, it is not necessary to unroll loops to discover situations where custom operations are useful. Often, a good programmer with knowledge of the function of the custom operations can see by simple inspection opportunities to exploit custom operations.

To understand how quadavg and dspuquadaddui can be used in this code, we examine the function of these custom operations.

The quadavg custom operation performs pixel averaging on four pairs of pixels in parallel. Formally, the operation of quadavg is as follows:

```
quadavg rscr1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [abcd], rsrc2 = [wxyz], and rdest = [pqrs] where a, b, c, d, w, x, y, z, p, q, r, and s are all unsigned eight-bit values. Then, quadavg computes the output vector [pqrs] as follows:

```
p = (a + w + 1) >> 1
q = (b + x + 1) >> 1
r = (c + y + 1) >> 1
s = (d + z + 1) >> 1
```

The pixel averaging in Figure 2-5 is evident in the first statement of each of the four groups of statements. The rest of the code—adding idct[i] value and performing the saturation test—can be performed by the dspuquadaddui operation. Formally, its function is as follows:

```
dspuquadaddui rsrc1 rsrc2 -> rdest
```

takes arguments in registers rsrc1 and rsrc2, and it computes a result into register rdest. rsrc1 = [efgh], rsrc2 = [stuv], and rdest = [ijkl] where e, f, g, h, i, j, k, and l are unsigned 8-bit values; s, t, u, and v are signed 8-bit values. Then, dspuquadaddui computes the output vector [ijkl] as follows:

```
i = uclipi(e + s, 255)
j = uclipi(f + t, 255)
k = uclipi(g + u, 255)
l = uclipi(h + v, 255)
```

The uclipi operation is defined in this case as it is for the separate TM32 operation of the same name described in Appendix A. Its definition is as follows:

```
uclipi (m, n)
{
    if (m < 0) return 0;
    else if (m > n) return n;
    else return m;
}
```

To make is easier to see how these operations can subsume all the code in Figure 2-5, Figure 2-6 shows the same code rearranged to group the related functions. Now it should be clear that the quadavg operation can replace the first four lines of the loop assuming that we can get the individual 8-bit elements of the back[] and forward[] arrays positioned correctly into the bytes of a 32-bit word. That, of course, is easy: simply align the byte arrays on word boundaries and access them with word (integer) pointers.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp0, temp1, temp2, temp3;

    for (i = 0; i < 64; i += 4)
    {
        temp0 = ((back[i+0] + forward[i+0] + 1) >> 1);
        temp1 = ((back[i+1] + forward[i+1] + 1) >> 1);
        temp2 = ((back[i+2] + forward[i+2] + 1) >> 1);
        temp3 = ((back[i+3] + forward[i+3] + 1) >> 1);

        temp0 += idct[i+0];
        if (temp0 > 255) temp0 = 255;
        else if (temp0 < 0) temp0 = 0;

        temp1 += idct[i+1];
        if (temp1 > 255) temp1 = 255;
        else if (temp1 < 0) temp1 = 0;

        temp2 += idct[i+2];
        if (temp2 > 255) temp2 = 255;
        else if (temp2 < 0) temp2 = 0;

        temp3 += idct[i+3];
        if (temp3 > 255) temp3 = 255;
        else if (temp3 < 0) temp3 = 0;

        destination[i+0] = temp0;
        destination[i+1] = temp1;
        destination[i+2] = temp2;
        destination[i+3] = temp3;
    }
}
```

Figure 2-6 Re-grouped code of Figure 2-5.

Similarly, it should now be clear that the dspuquadaddui operation can replace the remaining code (except, of course, for storing the result into the destination[] array) assuming, as above, that the 8-bit elements are aligned and packed into 32-bit words.

Figure 2-7 shows the new code. The arrays are now accessed in 32-bit (int-sized) chunks, the loop iteration control has been modified to reflect the 'four-at-a-time' operations, and the quadavg and dspuquadaddui operations have replaced the bulk of the loop code. Finally, Figure 2-8 shows a more compact expression

of the loop code, eliminating the temporary variable. Note that the TM32 C compiler does the optimization by itself.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i, temp;

    int *i_back    = (int *) back;
    int *i_forward = (int *) forward;
    int *i_idct    = (int *) idct;
    int *i_dest    = (int *) destination;

    for (i = 0; i < 16; i += 1)
    {
        temp = QUADAVG(i_back[i], i_forward[i]);

        temp = DSPUQUADADDUI(temp, i_idct[i]);

        i_dest[i] = temp;
    }
}
```

Figure 2-7 Using the custom operation dspquadaddui to speed up the loop of Figure 2-6.

Again, note that the code in Figure 2-7 and Figure 2-8 assumes that the character arrays are 32-bit word aligned and padded if necessary to fill an integral number of 32-bit words.

```
void reconstruct (unsigned char *back,
                  unsigned char *forward,
                           char *idct,
                  unsigned char *destination)
{
    int i;

    int *i_back    = (int *) back;
    int *i_forward = (int *) forward;
    int *i_idct    = (int *) idct;
    int *i_dest    = (int *) destination;

    for (i = 0; i < 16; i += 1)
        i_dest[i] = DSPUQUADADDUI(QUADAVG(i_back[i], i_forward[i]), i_idct[i]);
}
```

Figure 2-8 Final version of the frame-reconstruction code.

The original code required three additions, one shift, two tests, three loads, and one store per pixel. The new code using custom operations requires only two custom operations, three loads, and one store for *four* pixels, which is more than a factor of six improvement. The actual performance improvement can be even greater depending on how well the compiler is able to deal with the branches in the original version of the code, which depends in part on the surrounding code. Reducing the number of branches almost always improves the chances of realizing maximum performance on the TM32 CPU.

The code in Figure 2-8 illustrates several aspects of using custom operations in C-language source code. First, the custom operations require no special declarations or syntax; they appear to be simple function calls. Second, there is no need to explicitly specify register assignments for sources, destinations, and intermediate results; the compiler and scheduler assign registers for custom operations just as they would for built-in language operations such as integer addition. Third, the scheduler packs custom operations into TM32 VLIW instructions as effectively as it packs operations generated by the compiler for native language constructs.

Thus, although the burden of making effective use of custom operations falls on the programmer, that burden consists only of discovering the opportunities for exploiting the operations and then coding them using standard C-language notation. The compiler and scheduler take care of the rest.

2.4 Example 3: Motion-Estimation Kernel

Another part of the MPEG coding algorithm is motion estimation. The purpose of motion estimation is to reduce the cost of storing a frame of video by expressing the contents of the frame in terms of adjacent frames. A given frame is reduced to small blocks, and a subsequent frame is represented by specifying how these small blocks change position and appearance; usually, storing the difference information is cheaper than storing a whole block. For example, in a video sequence where the camera pans across a static scene, some frames can be expressed simply as displaced versions of their predecessor frames. To create a subsequent frame, most blocks are simply displaced relative to the output screen.

The code in this example is for a match-cost calculation, a small kernel of the complete motion-estimation code. As with the previous example, this code provides an excellent example of how to transform source code to make the best use of the TM32's custom operations.

Figure 2-9 shows the original source code for the match-cost loop. Unlike the previous example, the code is not a self-contained function. Somewhere early in the code, the arrays A[][] and B[][] are declared; somewhere between those declarations and the loop of interest, the arrays are filled with data.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 1)
        cost += abs(A[row][col] - B[row][col]);
}
```

Figure 2-9 Match-cost loop for MPEG motion estimation.

2.4.1 A Simple Transformation

First, we will look at the simplest way to use a TM32 custom operation.

We start by noticing that the computation in the loop of Figure 2-9 involves the absolute value of the difference of two unsigned characters (bytes). By now, we are familiar with the fact that TM32 includes a number of operations that process all four bytes in a 32-bit word simultaneously. Since the match-cost calculation is fundamental to the MPEG algorithm, it is not surprising to find a custom operation—ume8uu—that implements this operation exactly.

To understand how ume8uu can be used in this case, we need to transform the code as in the previous example. Though the steps are presented here in detail, a programmer with a even a little experience can often perform these transformations by visual inspection.

To use a custom operation that processes 4 pixel values simultaneously, we first need to create 4 parallel pixel computations. Figure 2-10 shows the loop of Figure 2-9 unrolled by a factor of 4. Unfortunately, the code in the unrolled loop is not parallel because each line depends on the one above it.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost += abs(A[row][col+0] - B[row][col+0]);
        cost += abs(A[row][col+1] - B[row][col+1]);
        cost += abs(A[row][col+2] - B[row][col+2]);
        cost += abs(A[row][col+3] - B[row][col+3]);
```

Figure 2-10 Unrolled, but not parallel, version of the loop from Figure 2-9.

Figure 2-11 shows a more parallel version of the code from Figure 2-10. By simply giving each computation its own cost variable and then summing the costs all at once, each cost computation is completely independent.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
for (row = 0; row < 16; row += 1)
{
    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(A[row][col+0] - B[row][col+0]);
        cost1 = abs(A[row][col+1] - B[row][col+1]);
        cost2 = abs(A[row][col+2] - B[row][col+2]);
        cost3 = abs(A[row][col+3] - B[row][col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
```

Figure 2-11 Parallel version of Figure 2-10.

Excluding the array accesses, the loop body in Figure 2-11 is now recognizable as the function performed by the ume8uu custom operation: the sum of 4 absolute values of 4 differences. To use the ume8uu operation, however, the code must access the arrays with 32-bit word pointers instead of with 8-bit byte pointers.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 1)
    cost += UME8UU(IA[i], IB[i]);
```

Figure 2-12 The loop of Figure 2-14 with the inner loop eliminated.

Figure 2-13 shows the loop recoded to access A[][] and B[][] as one-dimensional instead of two-dimensional arrays. We take advantage of our knowledge of C-language array storage conventions to perform

this code transformation. Recoding to use one-dimensional arrays prepares the code for transformation to 32-bit array accesses.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
unsigned char *CA = A;
unsigned char *CB = B;

for (row = 0; row < 16; row += 1)
{
    int rowoffset = row * 16;

    for (col = 0; col < 16; col += 4)
    {
        cost0 = abs(CA[rowoffset + col+0] - CB[rowoffset + col+0]);
        cost1 = abs(CA[rowoffset + col+1] - CB[rowoffset + col+1]);
        cost2 = abs(CA[rowoffset + col+2] - CB[rowoffset + col+2]);
        cost3 = abs(CA[rowoffset + col+3] - CB[rowoffset + col+3]);

        cost += cost0 + cost1 + cost2 + cost3;
```

Figure 2-13 The loop of Figure 2-11 recoded with one-dimensional array accesses.

(From here on, until the final code is shown, the declarations of the A and B arrays will be omitted from the code fragments for the sake of brevity.)

Figure 2-14 shows the loop of Figure 2-13 recoded to use ume8uu. Once again taking advantage of our knowledge of the C-language array storage conventions, the one-dimensional byte array is now accessed as a one-dimensional 32-bit-word array. The declarations of the pointers IA and IB as pointers to integers is the key, but also notice that the multiplier in the expression for row offset has been scaled from 16 to 4 to account for the fact that there are 4 bytes in a 32-bit word.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (row = 0; row < 16; row += 1)
{
    int rowoffset = row * 4;

    for (col4 = 0; col4 < 4; col4 += 1)
        cost += UME8UU(IA[rowoffset + col4], IB[rowoffset + col4]);
}
```

Figure 2-14 The loop of Figure 2-13 recoded with 32-bit array accesses and the ume8uu custom operation.

Of course, since we are now using one-dimensional arrays to access the pixel data, it is natural to use a single for loop instead of two. Figure 2-12 shows this streamlined version of the code without the inner loop. Since C-language arrays are stored as a linear vector of values, we can simply increase the number of iterations of the outer loop from 16 to 64 to traverse the entire array.

The recoding and use of the ume8uu operation has resulted in a substantial improvement in the performance of the match-cost loop. In the original version, the code executed 1280 operations (including loads, adds, subtracts, and absolute values); in the restructured version, there are only 256 operations—128 loads, 64 ume8uu operations, and 64 additions. This is a factor of five reduction in the number of operations executed. Also, the overhead of the inner loop has been eliminated, further increasing the performance advantage.

2.4.2 More Unrolling

The code transformations of the previous section achieved impressive performance improvements, but given the VLIW nature of the TM32 CPU, more can be done to exploit TM32's parallelism.

The code in Figure 2-12 has a loop containing only 4 operations (excluding loop overhead). Since TM32's branches have a 3-instruction delay and each instruction can contain up to 5 operations, a fully utilized minimum-sized loop can contain 16 operations (20 minus loop overhead).

The TM32 compilation system performs a wide variety of powerful code transformation and scheduling optimizations to ensure that the VLIW capabilities of the CPU are exploited. It is still wise, however, to make program parallelism explicit in source code when possible. Explicit parallelism can only help the compiler produce a fast running program.

To this end, we can unroll the loop of Figure 2-12 some number of times to create explicit parallelism and help the compiler create a fast running loop. In this case, where the number of iterations is a power-of-two, it makes sense to unroll by a factor that is a power-of-two to create clean code.

Figure 2-15 shows the loop unrolled by a factor of eight. The compiler can apply common subexpression elimination and other optimizations to eliminate extraneous operations in the array indexing, but, again, improvements in the source code can only help the compiler produce the best possible code and fastest-running program.

Figure 2-16 shows one way to modify the code for simpler array indexing.

```
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 8)
{
    cost0 = UME8UU(IA[i+0], IB[i+0]);
    cost1 = UME8UU(IA[i+1], IB[i+1]);
    cost2 = UME8UU(IA[i+2], IB[i+2]);
    cost3 = UME8UU(IA[i+3], IB[i+3]);
    cost4 = UME8UU(IA[i+4], IB[i+4]);
    cost5 = UME8UU(IA[i+5], IB[i+5]);
    cost6 = UME8UU(IA[i+6], IB[i+6]);
    cost7 = UME8UU(IA[i+7], IB[i+7]);

    cost += cost0 + cost1 + cost2 +
            cost3 + cost4 + cost5 +
            cost6 + cost7;
}
```

Figure 2-15 Unrolled version of Figure 2-12. This code makes good use of TM32's VLIW capabilities.

```
unsigned char A[16][16];
unsigned char B[16][16];
        .
        .
        .
unsigned int *IA = (unsigned int *) A;
unsigned int *IB = (unsigned int *) B;

for (i = 0; i < 64; i += 8, IA += 8, IB += 8)
{
    cost0 = UME8UU(IA[0], IB[0]);
    cost1 = UME8UU(IA[1], IB[1]);
    cost2 = UME8UU(IA[2], IB[2]);
    cost3 = UME8UU(IA[3], IB[3]);
    cost4 = UME8UU(IA[4], IB[4]);
    cost5 = UME8UU(IA[5], IB[5]);
    cost6 = UME8UU(IA[6], IB[6]);
    cost7 = UME8UU(IA[7], IB[7]);

    cost += cost0 + cost1 + cost2 +
            cost3 + cost4 + cost5 +
            cost6 + cost7;
}
```

Figure 2-16 Code from Figure 2-15 with simplified array index calculations.

 © TriMedia Technologies, Inc.

Chapter 3 Cache Architecture

3.1 Introduction

In order to provide the computational core of the DSPCPU32 with sufficient data, the TM32 employs a data cache, which exploits the temporal and spatial locality inherent in data streams. The performance of the data cache is crucial for the overall system performance. To further reduce the number of stall cycles, the software may issue cache control operations such as prefetches. The control operations may reduce the cache miss cycles considerably at a moderate cost in programming effort.

Only data from the DRAM aperture can be cacheable. All other apertures bypass the cache, as do accesses in the non-cacheable part of the DRAM aperture.

In order to provide the computational core of the DSPCPU32 with sufficient instruction bandwidth, the TM32 employs an instruction cache, which exploits the temporal and spatial locality inherent in instruction streams. By storing instructions in a compressed format in the memory and the instruction cache, both cache efficiency and instruction bandwidth from memory are increased. Instructions are decompressed before being delivered to the DSPCPU32 core.

The instruction cache is the only possible source of decompressed instructions to the DSPCPU32, which means that TM32 cannot execute from uncached memory. The instruction cache will only be able to cache the DRAM memory aperture. As a consequence, the DSPCPU32 can only execute from DRAM and boot code needs to be loaded into DRAM before the DSPCPU32 is started.

3.2 DRAM Aperture

TM32 implements a 32-bit linear address space of bytes. Within that address space, TM32 supports several different apertures for specific purposes. The DRAM aperture describes the part of the address space into which the external SDRAM is mapped. SDRAM must consist of a single, contiguous region of memory, which is the most practical configuration for TM32 systems.

The location and size of the DRAM aperture is defined by two registers, TM32_DRAM_LO and TM32_DRAM_HI (see Section 1.4.1 and Figure 1-5).

In normal operation, the base address registers are assigned once during boot and should not be changed when the DSPCPU32 is running. A memory operation will access SDRAM if its address satisfies:

```
[TM32_DRAM_LO] ≤ address < [TM32_DRAM_HI]
```

Any address outside this range cannot access SDRAM and the configuration of TM32_DRAM_LO and TM32_DRAM_HI needs to guarantee that SDRAM is present for every address within the range.

3.3 Data Cache

The data cache serves only the DSPCPU32 and is controlled by two memory units that execute the load and store operations issued by the DSPCPU32. The following sections describe the data cache and its operation; Table 3-1 summarizes the important characteristics for easy reference.

Table 3-1 . Summary of data cache characteristics

Characteristic	TM32 Implementation
Cache size	16 KB
Cache associativity	8-way set-associative
Block size	64 bytes
Valid bits	One valid bit per 64-byte block
Dirty bits	One dirty bit per 64-byte block
Miss transfer order	Miss transfers begin with the critical word first
Replacement policies	Copyback, allocate on write, hierarchical LRU
Endianness	Either little- or big-endian, determined by PCSW.BSX bit
Ports	The cache is quasi dual ported; two accesses can proceed concurrently if they reference different banks (determined by bits [4:2] of the computed addresses)
Alignment	Access must be naturally aligned (32-bit words on 32-bit boundaries, 16-bit halfwords on 16-bit boundaries); the appropriate number of LSBs of un-naturally aligned addresses are set to zero. For misaligned stores, PCSW.MSE is asserted to generate an exception
Partial word operations	The cache implements 8-bit and 16-bit accesses with the same performance as 32-bit accesses
Operation latency	Three cycles for both load and store operations
Coherency enforcement	Software uses special operations to enforce cache coherency
Cache locking	Up to 1/2 (four out of 8 blocks of each set) of the cache contents can be locked; granularity is 64-byte
Non-cacheable region	One non-cacheable aperture in the DRAM address space is supported.

3.3.1 General Cache Parameters

The TM32 data cache is 16 KB in size with a 64-byte block size. Thus, it contains 256 blocks each with its own address tag. The cache is 8-way set-associative, so there are 32 sets, each containing 8 tags. A single valid bit is associated with a block, so each block and associated address tag is either entirely valid in the cache or invalid. On a cache miss, 64 bytes are read from SDRAM to make the entire block valid.

Each block also contains a dirty bit, which is set whenever a write to the block occurs. Each set contains 10 bits to support the hierarchical LRU replacement policy.

The geometry of the data cache is available to software by reading the MMIO register DC_PARAMS. Figure 3-1 shows the format of the DC_PARAMS register; Table 3-2 lists its field values. The product of block size, associativity, and number of sets gives the total cache size (16 KB in this case).

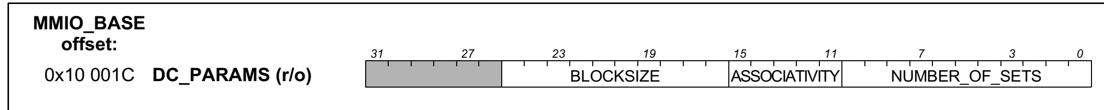


Figure 3-1 Format of the DC_PARAMS register.

Table 3-2 DC_PARAMS field values

Field Name	Value
BLOCK SIZE	64
ASSOCIATIVITY	8
NUMBER_OF_SETS	32

3.3.2 Address Mapping

TM32 data addresses are mapped onto the data cache storage structure as shown in Figure 3-2. A data address is partitioned into four fields as described in Table 3-3.

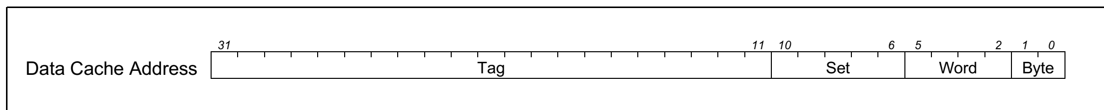


Figure 3-2 Data cache address partitioning.

Table 3-3 Data address field partitioning

Field	Address Bits	Purpose
Byte	1..0	Byte offset within a word for byte or half-word accesses
Word	5..2	Selects one of the words in a set (one of 16 words in the case of TM32)
Set	10..6	Selects one of the sets in the cache (one of 32 in the case of TM32)
Tag	31..11	Compared against address tags of set members

3.3.3 Miss Processing Order

When a miss occurs, the data cache fills the block containing the requested word from the critical word first. The CPU is stalled until the first word is transferred. The block is then filled up while the CPU keeps running.

3.3.4 Replacement Policies, Coherency

The cache implements a copyback replacement policy with one dirty bit per 64-byte block. Thus, when a miss occurs and the block selected for replacement has its dirty bit set, the dirty block must be written to main memory to preserve its modified contents. On TM32, the dirty block is written to memory before the needed block is fetched.

Coherency is not maintained in any way by hardware between the data cache, the instruction cache, and main memory. Special operations are available to implement cache coherency in software. See Section 3.6, "Cache Coherency," for a discussion of coherency issues.

Write misses are handled with an allocate-on-write policy—the write that caused the miss stores its data in the cache after the missing block is fetched into the cache.

The cache implements a hierarchical LRU replacement algorithm to determine which of the eight elements (blocks) in a set is replaced. The algorithm partitions the eight set elements into four groups, each group with two elements. The hierarchical LRU replacement victim is determined by selecting the least-recently used group of two elements and then selecting the least-recently used element in that group. This hierarchical algorithm yields performance close to full LRU but is simpler to implement.

See Section 3.5, "LRU Algorithm," for a full discussion of the LRU algorithm.

3.3.5 Alignment, Partial-Word Transfers, Endian-ness

The cache implements 32-bit word, 16-bit half-word, and 8-bit byte transfers. All transfers, however, must be to addresses that are naturally aligned; that is, 32-bit words must be aligned on 32-bit boundaries, and 16-bit halfwords must be aligned on 16-bit boundaries.

The CPU has the capability to use either big- or little-endian byte order as determined by PCSW.BSX..

3.3.6 Dual Ports

To allow two accesses to proceed in parallel, the data cache is quasi-dual ported. The cache is implemented as eight banks of single-ported memory, but the hardware allows each bank to operate independently. Thus, when the addresses of two simultaneous accesses select two different banks, both accesses can complete simultaneously. Bank selection is determined by the three low-order address bits [4..2] of each address. Thus, the words in a 64-byte cache block are distributed among the eight blocks, which prevents conflicts between two simultaneously issued accesses to adjacent words in a cache block. The TM32 compiling system attempts to avoid bank conflicts as much as possible.

The dual-ported cache can execute the load and store opcodes (ild8d, uld8d, ild16d, uld16d, ld32d, h_st8d, h_st16d, h_st32d, ild8r, uld8r, ild16r, uld16r, ld32r, ild16x, uld16x, ld32x) in either or both of the two ports.

The special opcodes alloc, dcb, dinvalid, pref, rdtag and rdstatus can only be executed in the second port, not in the first port. Whenever any of these special opcodes is issued in the second port, there should not be a concurrent load or store operation in the first. This is a special scheduling constraint.

3.3.7 Cache Locking

The data cache allows the contents of up to one-half of its blocks to be locked. Thus, on TM32, up to 8 KB of the cache can be used as a high-speed local data memory. Only four out of eight blocks in any set can be locked.

A locked block is never chosen as a victim by the replacement algorithm; its contents remain undisturbed until either (1) the block's locked status is changed explicitly by software, or (2) a dinvalid operation is executed that targets the locked block.

Cache locking occurs only for the data in the address range described by the MMIO registers DC_LOCK_ADDR and DC_LOCK_SIZE. The granularity of the address range is one 64-byte cache

 © TriMedia Technologies, Inc.

block. The MMIO register DC_LOCK_CTL contains the cache-locking enable bit DC_LOCK_ENABLE. Figure 3-3 shows the layout of the data-cache lock registers. Locking will occur for an address if locking is enabled and both of the following are true:

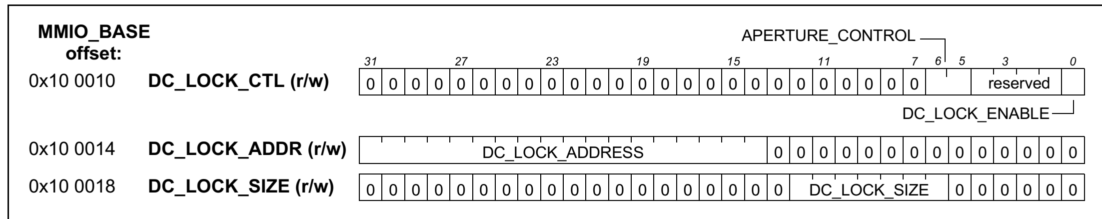


Figure 3-3 Formats of the registers in charge of data-cache locking.

1. The address is greater than or equal to the value in DC_LOCK_ADDR.
2. The address is less than the sum of the values in DC_LOCK_ADDR and DC_LOCK_SIZE.

Programmers (or compilers) must combine all data that needs to be locked into this single linear address range.

Setting DC_LOCK_ENABLE to ‘1’ causes the following sequence of events:

1. All blocks that are in cache locations that will be used for locking are copied back to main memory (if they are dirty) and removed from the cache.
2. All blocks in the lock range are fetched from main memory into the cache. If any block in the lock range was already in the cache, it’s first copied back into main memory (if it’s dirty) and invalidated.
3. The LRU status of any set that contains locked blocks is set to the initialization value.
4. Cache locking is activated so that the locked blocks cannot be victims of the replacement algorithm.

This sequence of events is triggered by writing ‘1’ to DC_LOCK_ENABLE even if the enable is already set to ‘1’. Setting DC_LOCK_ENABLE to ‘0’ causes no action except to allow the previously locked blocks to be replacement victims.

To program a new lock range, the following sequence of operations is used:

1. Disable cache locking by writing ‘0’ to DC_LOCK_ENABLE.
2. Define a new lock range by writing to DC_LOCK_ADDR and DC_LOCK_SIZE.
3. Enable cache locking by writing ‘1’ to DC_LOCK_ENABLE.

Dirty locked blocks can be written back to main memory while locking is enabled by executing copyback operations in software.

Programmer’s note: Software should not execute dinvalid operations on a locked block. If it does, the block will be removed from the cache, creating a ‘hole’ in the lock range (and the data cache) that cannot be reused until locking is deactivated.

Cache locking is disabled by default when TM32 is reset.

The RESERVED field in DC_LOCK_CTL should be ignored on reads and written as all zeroes.

Locking should not be enabled by external PI-master access to the MMIO registers.

3.3.8 Non-cacheable Region

The data cache supports one non-cacheable address region within the DRAM address space aperture. The base address of this region is determined by the value in the TM32_DRAM_CLIMIT MMIO register (see Section 1.4.1 and Figure 1-5). Since uncached memory operations always incur many stall cycles, the non-

cacheable region should be used sparingly. A memory operation is non-cacheable if its target address satisfies:

```
[TM32_DRAM_CLIMIT] <= address < [TM32_DRAM_HI]
```

Thus, the non-cacheable region is at the high end of the DRAM aperture. The format of TM32_DRAM_CLIMIT forces the size of the non-cacheable region to be a multiple of 64 KB.

Programmer's note: When TM32_DRAM_CLIMIT is changed to enlarge the region that is non-cacheable, software must ensure coherency. This is accomplished by explicitly copying back dirty data (using dcb operations) and invalidating (using dinvalid operations) the cache blocks in the previously unlocked region.

3.3.9 Special Data Cache Operations

A program can exercise some control over the operation of the data cache by executing special operations. The special operations can cause the data cache to initiate the copyback or invalidation of a block in the cache. These operations are typically used by software to keep the cache coherent with main memory.

In addition, there are special operations that allow a program to read tag and status information from the data cache.

Special data cache operations are always executed on the memory port associated with issue slot 5.

3.3.9.1 Copyback and invalidate operations

The data cache controller recognizes a copyback and an invalidate operation as shown in Table 3-4.

Table 3-4 Copyback and invalidate operations

Mnemonic	Description
dcb(offset) r*src1*	Data-cache copyback block. Causes the block that contains the target address to be copied back to main memory if the block is valid and dirty.
dinvalid(offset) r*src1*	Data-cache invalidate block. Causes the block that contains the target address to be invalidated. No copyback occurs even if the block is dirty.

The dcb and dinvalid operations both compute a target word address that is the sum of a register and seven-bit offset. The offset can be in the range [–256..252] and must be divisible by four.

dcb operation. The dcb operation computes the target address, and if the block containing the address is found in the data cache, its contents are written back to main memory if the block is both valid and dirty. If the block is not present, not valid, or not dirty, no action results from the dcb operation. If the dcb causes a copyback to occur, the CPU is stalled until the copyback completes. If the block is not in cache, the operation causes no stall cycles. If the block is in cache but not dirty, the operation causes 4 stall cycles. If the block is dirty, the dcb operation causes a writeback and takes at least 19 stall cycles.

The dcb operation clears the dirty bit but leaves a valid copy of the written-back block in the cache.

dinvalid operation. The dinvalid operation computes the target address, and if the block containing the address is found in the data cache, its valid and dirty bits are cleared. No copyback operation will occur even if the block is valid and dirty prior to executing the dinvalid operation. The CPU is stalled for 2 cycles, if the target block is in the cache; otherwise, no stall cycles occur.

A dinvalid or dcb operation updates the LRU information to least recently used in its set.

Programmer's note: Software should not execute dinvalid operations on locked blocks; otherwise, a 'hole' is created that cannot be reused until locking is deactivated.

© TriMedia Technologies, Inc.

0.0.0.1 Data cache tag and status operations

The data cache controller recognizes two DSPCPU32 operations for reading cache status as shown in Table 3-5.

The rdtag and rdstatus operations both compute a target word address that is the sum of a register and scaled seven-bit offset. The offset must be divisible by four and in the range [–256..252].

Table 3-5 . Cache read-status operations

Mnemonic	Description
rdtag(offset) r*src1*	Read data-cache tag. The target address selects a data-cache block directly; the operation returns a 32-bit result containing the 21-bit cache tag and the valid bit.
rdstatus(offset) r*src1*	Read data-cache status. The target address selects a data-cache set directly; the operation returns a 32-bit result containing the set's eight dirty bits and ten LRU bits.

rdtag operation. The target address computed by rdtag selects the data cache block by specifying the cache set and set element directly. Address bits [10..6] specify the cache set (one of 32), and bits [13..11] specify the set element (one of eight). All other target address bits are ignored. This operation causes no CPU stall cycles.

The result of the rdtag operation is a full 32-bit word with the format shown in Figure 3-4.

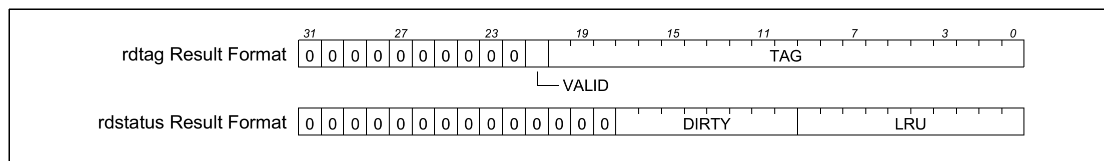


Figure 3-4 Result formats for rdtag and rdstatus operations.

rdstatus operation. The target address computed by rdstatus selects the data cache set by specifying the set number directly. Address bits [10..6] specify the cache set (one of 32); all other target address bits are ignored. This operation causes 1 CPU stall cycle.

The result of the rdstatus operation is a full 32-bit word with the format shown in Figure 3-4. See Section 3.5.4, "LRU Bit Definitions," for a description of the LRU bits.

0.0.0.2 Data cache allocation operation

The data cache controller recognizes allocation operations as shown in Table 3-6. The allocation operations allocate a block and set the status of this block to valid. No data is fetched from main memory. The allocated block is undefined after this operation. The programmer has to fill it with valid data by store operations. Allocation operations to apertures other than cacheable DRAM will be discarded. Allocation of a non-dirty block causes 3 stall cycles. Allocation of a dirty block will cause writeback of this block to the SDRAM and take at least 11 stall cycles.

Table 3-6 . Data cache allocation operations

Mnemonic	Description
allocd(offset) r*src1*	Data-cache allocate block with displacement. Causes the block with address (rsrc1+offset) & (~(cache_block_size - 1)) to be allocated and set valid.
allocr r*src1 rsrc2*	Data-cache allocate block with index. Causes the block with address (rsrc1+rsrc2) & (~(cache_block_size - 1)) to be allocated and set valid.
allocx r*src1 rsrc2*	Data-cache allocate block with scaled index. Causes the block with address (rsrc1 + 4 * rsrc2) & (~(cache_block_size - 1)) to be allocated and set valid.

0.0.0.3 Data cache prefetch operation

The data cache controller recognizes prefetch operations as shown in Table 3-7. The prefetch operations load a full cache block from memory concurrently with other computation. If the prefetched block is already in cache, no data is fetched from main memory. Prefetch operations to other apertures than cacheable DRAM are discarded. This operation is not guaranteed to execute, it will not execute if the cache is already occupied with two cache misses when the operation is issued. The prefetch operations cause 3 stall cycles if there is no copyback of a dirty block. If a dirty block is the target of the prefetch, the dirty block will be written back to SDRAM, and at least 11 stall cycles are taken.

Table 3-7 . Data cache prefetch operations

Mnemonic	Description
prefd(offset) r*src1*	Data-cache prefetch block with displacement. Causes the block with address (rsrc1+offset) & (~(cache_block_size - 1)) to be prefetched
prefr r*src1 rsrc2*	Data-cache prefetch block with index. Causes the block with address (rsrc1+rsrc2) & (~(cache_block_size - 1)) to be prefetched.
pref16x r*src1 rsrc2*	Data-cache prefetch block with scaled 16-bit index. Causes the block with address (rsrc1 + 2 * rsrc2) & (~(cache_block_size - 1)) to be prefetched.
pref32x r*src1 rsrc2*	Data-cache prefetch block with scaled 32-bit index. Causes the block with address (rsrc1 + 4 * rsrc2) & (~(cache_block_size - 1)) to be prefetched.

3.3.10 Memory Operation Ordering

The TM32 memory system implements traditional ordering for memory operations that are issued in different clock cycles. That is, the effects of a memory operation issued in cycle j occur before the effects of a memory operation issued in cycle j+1.

For memory operations issued in the same cycle, however, it is not possible to execute memory operations in a traditional order. So long as the simultaneous memory operations access different addresses (aliasing is not possible in TM32), no problems can occur. If two simultaneous operations do access the same address, however, TM32 behavior is undefined. Specifically, two cases are possible:

1. When multiple values are written to the same address in the same cycle, the resulting value in memory is undefined.
2. When a read and a write occur to the same address in the same clock cycle, the value returned by the read is undefined.

The behavior of simultaneous accesses to the same address is undefined regardless of whether one or both memory operations hit in the cache.

Hidden Memory System Concurrency. Some cache operations may be overlapped with CPU execution. In general, a program cannot determine in what order cache misses will complete nor can a program determine when and in what order copyback operations will complete. A program can, however, enforce the completion of copyback transactions to main memory because copyback and invalidate operations can complete only if pending copyback transactions for the same block have completed. Thus, a program can synchronize to the completion of a copyback operation by dirtying a block, issuing a copyback operation for the block, and then issuing an invalidate operation for the block.

Ordering Of Special Memory Operations. The following are special memory operations:

1. Loads or stores to MMIO aperture.
2. Loads or stores to Aperture1 aperture.
3. Non-cached loads or stores.
4. Any copyback or invalidate operation.

The CPU is stalled until these special memory operations are completed; there is no overlap of CPU execution with these special memory operations. Thus, a programmer can assume that traditional memory operation ordering applies to special memory operations. Note, however, that ordering is undefined for two special memory operations issued in the same cycle.

3.3.11 Operation Latency

Load and store operations have an operation latency of three cycles, regardless of the size of the data transfer.

3.3.12 MMIO Aperture References

Memory operations that reference MMIO aperture are not cached, and the CPU is stalled until the MMIO reference completes. A MMIO register reference occurs when an address is in the range:

`[MMIO_BASE] ≤ address < ([MMIO_BASE] + 0x200000)`

The size of the MMIO aperture is hardwired at 2 MB.

3.3.13 Aperture1 References

Memory operations that reference the Aperture1 aperture are not cached, and the CPU is stalled until the Aperture1 reference completes. An Aperture1 reference occurs when an address is in the range:

`[TM32_APERT1_LO] ≤ address < ([TM32_APERT1_HI]`

See Section 1.4.3 and Figure 1-7 for the definition of TM32_APERT1_LO and TM32_APERT1_HI.

3.3.14 CPU Stall Conditions

The data cache causes the CPU to stall when:

1. Any cache miss occurs.
2. Two simultaneously issued, cacheable memory operations need to access the same cache bank (bank conflict).
3. An access that references an address in the MMIO aperture is issued.
4. An access that references an address in the Aperture1 aperture is issued.
5. A non-trivial copyback or invalidate operation is issued.
6. An access to the non-cacheable region in the DRAM aperture is issued.

3.3.15 Data Cache Initialization

When TM32 is reset, the data cache executes an initialization sequence. The cache asserts the CPU stall signal while it sequentially resets all valid and dirty bits. The cache de-asserts the stall signal after completing the initialization sequence.

3.4 Instruction Cache

The instruction cache stores compressed CPU instructions; instructions are decompressed before being delivered to the CPU. The following sections describe the instruction cache and its operation; Table 3-8 summarizes instruction-cache characteristics.

Table 3-8 . Instruction cache characteristics

Characteristic	TM32 Implementation
Cache size	32 KB
Cache associativity	8-way set-associative
Block size	64 bytes
Valid bits	One valid bit per 64-byte block
Replacement policy	Hierarchical LRU (least-recently used) among the eight blocks in a set
Operation latency	Branch delay is three cycles
Coherency enforcement	Software uses a special operation to enforce cache coherency
Cache locking	Up to 1/2 (four out of eight blocks of each set) of the cache contents can be locked; granularity is 64 bytes

3.4.1 General Cache Parameters

The TM32 instruction cache is 32 KB in size with a 64-byte block size. Thus, the cache contains 512 blocks each with its own address tag. The cache is 8-way set-associative, so there are 64 sets, each containing 8 tags. A single valid bit is associated with a block, so each block and associated address tag is either entirely valid or invalid; on a cache miss, 64 bytes are read from SDRAM to make the entire block valid.

The geometry of the instruction cache is available to software by reading the MMIO register IC_PARAMS. Figure 3-5 shows the format of the IC_PARAMS register; Table 3-9 lists its field values.

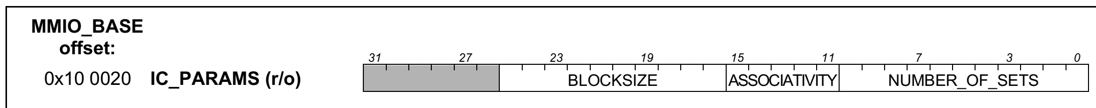


Figure 3-5 Format of the instruction cache parameters register

The product of the block size, associativity, and number of sets gives the total cache size (32 KB in this case).

Table 3-9 . IC_PARAMS field values

Field Name	Value
BLOCKSIZE	64
ASSOCIATIVITY	8
NUMBER_OF_SETS	64

3.4.2 Address Mapping

TM32 instruction addresses are mapped onto the data cache storage structure as shown in Figure 3-7. An instruction address is partitioned into three fields as described in Table 3-10.

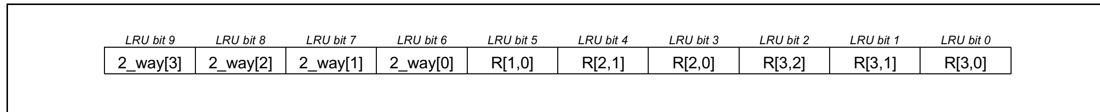


Figure 3-6 LRU bit definitions; 2_way[k] is the two-way LRU bit of pair k = (j div 2) for set element j.

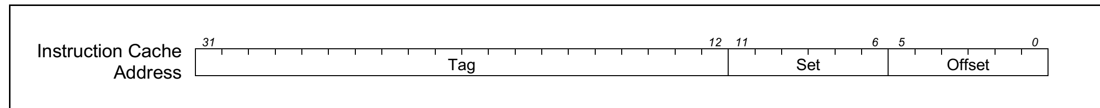


Figure 3-7 Instruction-cache address partitioning.

Table 3-10 Instruction Address Field Partitioning

Field	Address Bits	Purpose
Offset	5..0	Byte offset into a set
Set	11..6	Selects one of the sets in the cache (one of 64 in the case of TM32)
Tag	31..12	Compared against address tags of set members

3.4.3 Miss Processing Order

When a miss occurs, the instruction cache starts filling the requested block from the beginning of the block. The DSPCPU32 is stalled until the entire block is fetched and stored in the cache.

3.4.4 Replacement Policy

The hierarchical LRU replacement policy implemented by the instruction cache is identical to that implemented by the data cache. See Section 3.3.4, “Replacement Policies, Coherency,” for a description of the hierarchical LRU algorithm.

3.4.5 Location of Program Code

All program code must first be loaded into SDRAM. The instruction cache cannot fetch instructions from other memories or devices. In particular, the cache cannot fetch code from on-chip devices or over the PCI bus.

3.4.6 Branch Units

The instruction cache is closely coupled to three branch units. Each unit can accept a branch independently, so three branches can be processed simultaneously in the same cycle.

Branches in TM32 are called ‘delayed branches’ because the effect of a successful (taken) branch is not seen in the flow of control until some number of cycles after the successful branch is executed. The number of cycles of latency is called the branch delay. On TM32, the branch delay is three cycles.

Although three branches can be executed simultaneously, correct operation of the DSPCPU32 requires that only one branch be successful (taken) in any one cycle. DSPCPU32 operation is undefined if more than one concurrent branch operation is successful.

Each branch unit takes four inputs from the DSPCPU32: the branch opcode, a guard bit, a branch condition, and a branch target address. A branch is deemed successful if and only if the opcode is a branch opcode, the guard bit is TRUE (i.e., = 1), and the condition (determined by the opcode) is satisfied.

3.4.7 Coherency: Special iclr Operation

A program can exercise some control over the operation of the instruction cache by executing the special iclr operation. This operation causes the instruction cache to clear the valid bits for all blocks in the cache, including locked blocks. The LRU replacement status of all blocks is reset to its initial value. The CPU is stalled while iclr is executing.

See Section 3.6, “Cache Coherency,” for further discussion of coherency issues.

3.4.8 Reading Tags and Cache Status

The instruction cache supports read access to its tag and status bits, but not through special operations as with the data cache. Since the instruction cache and branch units can execute only resultless operations, access to the instruction-cache tags and status bits is implemented using normal load operations executed by the DSPCPU32 that reference a special region in the MMIO address aperture. The region is 64 KB long and starts at MMIO_BASE. Instruction cache tags and status bits are read-only; store operations to this region have no effect. MMIO operations to this special region are only allowed by the DSPCPU32, not by any other masters of the on-chip data highway, such as external PI initiators.

Programmer’s note: Tag and status information cannot be read by PI access, but only by DSPCPU32 access. Tag and status read cannot be scheduled in the same cycle with or one cycle after an iclr operation.

Reading A Tag And Valid Bit. To read the tag and valid bit for a block in the instruction cache, a program can execute a ld32 operation directed at the instruction-cache region in the MMIO aperture. The top of Figure 3-8 shows the required format for the target address. The most-significant 16 bits must be equal to MMIO_BASE, the least-significant 15 bits select the block (by naming the set and set member), and bit 15 must be set to zero to perform a tag read.

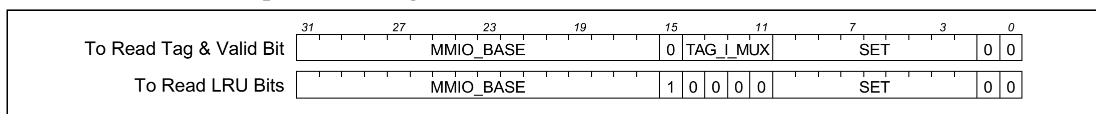


Figure 3-8 Required address format for reading instruction-cache tags and status.

Note that in TM32, valid set numbers range from 0 to 63. Space to encode set numbers 64 to 511 is provided for future extensions.

A ld32 with an address as specified above returns a 32-bit result with the format shown at the top of Figure 3-9. Bit 20 contains the state of the valid bit, and the least-significant 20 bits contain the tag for the block addressed by the ld32.

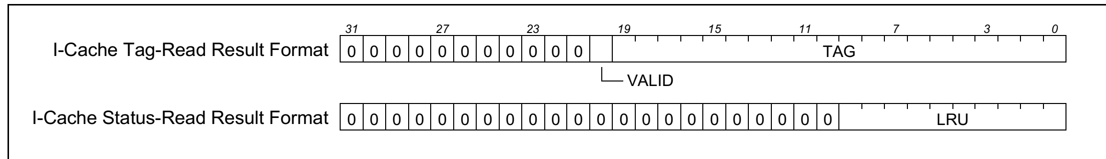


Figure 3-9 Result formats for reads from the instruction-cache region of the MMIO aperture.

Reading The LRU Bits. To read the LRU bits for a set in the instruction cache, a program can execute a ld32 operation as above but using the address format shown at the bottom of Figure 3-8. In this format, bit 15 is set to one to perform the read of the LRU bits, and the tag_i_mux field is set to zeros because it is not needed.

Reading the LRU bits produces a 32-bit result with the format shown at the bottom of Figure 3-9. The least-significant ten bits contain the state of the LRU bits when the ld32 was executed. See Section 3.5.4, "LRU Bit Definitions," for a description of the LRU bits.

Note that the tag_i_mux and set fields in the address formats of Figure 3-8 are larger than necessary for the instruction cache in TM32. These fields will allow future implementations with larger instruction caches to use a compatible mechanism for reading instruction cache information. The tag_i_mux field can accommodate a cache of up to 16-way set-associativity, and the set field can accommodate a cache with up to 512 sets. For TM32, the following constraints of the values of these fields must be observed:

1. $0 \le$ tag_i_mux ≤ 7
2. $0 \le$ set ≤ 63

3.4.9 Cache Locking

Like the data cache, the instruction cache allows up to one-half of its blocks to be locked. A locked block is never chosen as a victim by the replacement algorithm; its contents remain undisturbed until the locked status is changed explicitly by software. Thus, on TM32, up to 16 KB of the cache can be used as a high-speed instruction 'ROM.' Only four out of eight blocks in any set can be locked.

The MMIO registers IC_LOCK_ADDR, IC_LOCK_SIZE, and IC_LOCK_CTL—shown in Figure 3-10—are used to define and enable instruction locking in the same way that the similarly named data-cache locking registers are used. Section 3.3.7, "Cache Locking," describes the details of cache locking; they are not repeated here.

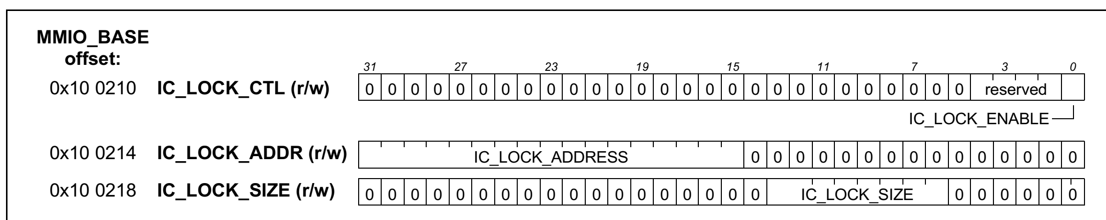


Figure 3-10 Formats of the registers that control instruction-cache locking.

Setting the IC_LOCK_ENABLE bit (in IC_LOCK_CTL) to '1' causes the following sequence of events:

1. The instruction cache invalidates all blocks in the cache.
2. The instruction cache fetches all blocks in the lock range (defined by IC_LOCK_ADDR and

IC_LOCK_SIZE) from main memory into the cache.
3. Cache locking is activated so that the locked blocks cannot be victims of the replacement algorithm.

The only difference between this sequence and the initialization sequence for data-cache locking is that dirty blocks (which cannot exist in the instruction cache) are not written back first.

Programmer's note: Programmers (or compilers) must combine all instructions that need to be locked into the single linear instruction-locking address range.

The special iclr operation also removes locked blocks from the cache. If blocks are locked in the instruction cache, then instruction cache locking should be disabled in software (by writing '0' to IC_LOCK_CTL) before an iclr operation is issued.

Locking should not be enabled by PCI accesses to the MMIO register.

3.4.10 Instruction Cache Initialization and Boot Sequence

When TM32 is reset, the instruction cache executes an initialization and processor boot sequence. While reset is asserted, the instruction cache forces NOP operation to the DSPCPU32, and the program counter is set to the default value reset_vector. When reset is deasserted, the initialization and boot sequence is as follows.

1. The stall signal is asserted to prevent activity in the DSPCPU32 and data cache.
2. The valid bits for all blocks in the instruction cache are reset.
3. At the completion of the block invalidation scan, the stall signal to the DSPCPU32 and data cache are deasserted.
4. The DSPCPU32 can begin normal operation with an instruction fetch from the address start_vector.

The initialization process takes 512 clock cycles. Reset sets start_vector equal to TM32_START_ADR. TM32_START_ADR and boot is described in Section 1.5.1.

3.5 LRU Algorithm

When a cache miss occurs, the block containing the requested data must be brought into the cache to replace an existing cache block. The LRU algorithm is responsible for selecting the replacement victim by selecting the least-recently-used block.

The 8-way set-associative caches implement a hierarchical LRU replacement algorithm as follows. Eight sets are partitioned into four groups of two elements each. To select the LRU element:

- First, the LRU pair is selected out of the four pairs using a four-way LRU algorithm.
- Second, the LRU element of the pair is selected using a two-way LRU algorithm.

3.5.1 Two-Way Algorithm

The two-way LRU requires an administration of one bit per pair of elements. On every cache hit to one of the two blocks, the cache writes once to this bit (just a write, not a read-modify-write). If the even-numbered block is accessed, the LRU bit is set to '1'; if the odd-numbered block is accessed, the LRU bit is set to '0'. On a miss, the cache replaces the LRU element, i.e. if the LRU bit is '0', the even numbered element will be replaced; if the LRU bit is '1', the odd numbered element will be replaced.

3.5.2 Four-Way Algorithm

For administration of the four-way algorithm, the cache maintains an upper-left triangular matrix ‘R’ of 1-bit elements without the diagonal. R contains six bits (in general, n×(n–1)/2 bits for n-way LRU). If set element k is referenced, the cache sets row k to ‘1’ and column k to ‘0’:

```
R[k, 0..n-1] ← 1,
R[0..n-1, k] ← 0
```

The LRU element is the one for which the entire row is ‘0’ (or empty) and the entire column is ‘1’ (or empty):

```
R[k, 0..n-1] = 0 and R[0..n-1, k] = 1
```

For a 4-way set-associative cache, this algorithm requires six bits per set of four cache blocks. On every cache hit, the LRU info is updated by setting three of the six bits to ‘0’ or ‘1’, depending on the set element that was accessed. The bits need only be written, no read-modify-write is necessary. On a miss, the cache reads the six LRU bits to determine the replacement block.

TM32 combines the two-way and four-way algorithms into an 8-way hierarchical LRU algorithm. A total of ten administration bits are required: six to maintain the four-way LRU plus four bits maintain the four two-way LRUs.

The hierarchical algorithm has performance close to full eight-way LRU, but it requires far fewer bits—ten instead of 28 bits—and is much simpler to implement.

To update the LRU bits on a cache hit to element j (with 0 <= j <= 7), the cache applies m = (j div 2) to the four-way LRU administration and (j mod 2) is applied to the two-way administration of pair m. To select a replacement victim, the cache first determines the pair p from the four-way LRU and then retrieves the LRU bit q of pair p. The overall LRU element is the p×2+q.

3.5.3 LRU Initialization

Reset causes the LRU administration bits to initialized to a legal state:

```
R[1,0] ← R[2,0] ← R[3,0] ← 1
R[2,1] ← R[3,1] ← R[3,2] ← 0
2_way[3] ← 2_way[2] ← 2_way[1] ← 2_way[0] ← 0
```

3.5.4 LRU Bit Definitions

The ten LRU bits per set are mapped as shown in Figure 3-11. This is the format of the LRU field as returned by the special operation rdstatus for the data cache and a ld32 from MMIO space (see Section 3.4.8, “Reading Tags and Cache Status”) for the instruction cache.

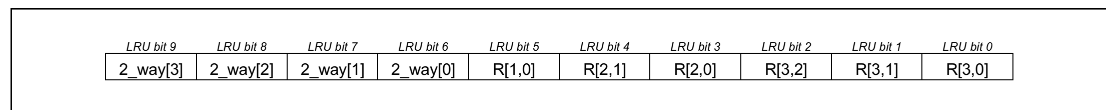


Figure 3-11 LRU bit definitions; 2_way[k] is the two-way LRU bit of pair k = (j div 2) for set element j.

3.5.5 LRU for the Dual-Ported Cache

For the TM32 dual-ported data cache, two memory operations to the same set are possible in a single clock cycle. To support this concurrency, two updates of the LRU bits of a single set must be possible.

The following rules are used by TM32:

1. LRU bits that are changed by exactly one port receive the value according to the algorithm described above.
2. LRU bits that are changed by both ports receive a value as if the algorithm were first applied for the access in port zero and then for the access in port one.

3.6 Cache Coherency

The TM32 hardware does not implement coherency between the caches and main memory. Generalized coherency is the responsibility of software, which can use the special operations dcb, dinvalid, and iclr to enforce cache/memory synchronization.

3.6.1 Example 1: Data-Cache/Input-Unit Coherency

Before the CPU commands the video-in unit to capture a video frame, the CPU must be sure that the data cache contains no blocks that are in the address region that the video-in unit will use to store the input frame. If the video-in unit performs its input function to an address region and the data cache does hold one or more blocks from that region, any of the following may happen:

- A miss in the data cache may cause a dirty block to be copied back to the address region being used by the video-in unit. If the video-in unit already stored data in the block, the write-back will corrupt the frame data.
- The CPU will read stale data from the cache instead of from the block in main memory. Even though the video-in unit stored new video data in the block in main memory, the cache contents will be used instead because it is still valid in the cache.

To prevent erroneous copybacks or the use of stale data, the CPU must use dinvalid operations to invalidate all blocks in the address region that will be used by the VI unit.

3.6.2 Example 2: Data-Cache/Output-Unit Coherency

Before the CPU commands the video-out unit to send a frame of video, the CPU must be sure that all the data for the frame has been written from the data cache to the region of main memory that the video-out unit will output. Explicit action is necessary because the data cache—with its copyback write policy—will hold an exclusive copy of the data until it is either replaced by the LRU algorithm or the CPU explicitly forces it to be copied back to main memory.

Before an output command is issued to the video-out unit, the CPU must execute dcb operations to force coherency between cache contents and main memory.

3.6.3 Example 3: Instruction-Cache/Data-Cache Coherency

If code prepared by a program running on the CPU must be subsequently executed, coherency between the instruction and data caches must be enforced. This is accomplished by a two-step process:

1. Coherency between the data cache and main memory must be enforced since the instruction cache can

 © TriMedia Technologies, Inc.

fetch instructions only from main memory.

2. Coherency between the instruction cache and main memory is enforced by executing an iclr operation.

The CPU will now be able to fetch and execute the new instructions.

3.6.4 Example 4: Instruction-Cache/Input-Unit Coherency

When an input unit is used to load program code into main memory, the iclr operation must be issued before attempting to execute the new code.

3.7 Performance Evaluation Support

The caches implement support for performance evaluation. Several events that occur in the caches can be counted using the TM32 timer/counters, by selecting the source CACHE1 and/or CACHE2, as described in Section 1.6, "Timers." Two different events can be tracked simultaneously by using 2 timers.

The MMIO register MEM_EVENTS determines which events are counted. See Figure 3-12 for the format of MEM_EVENTS. Table 3-11 lists the events that can be tracked and the corresponding values for the MEM_EVENTS fields.

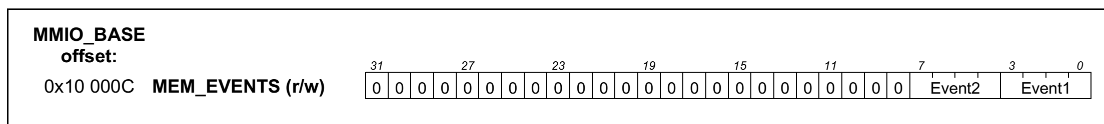


Figure 3-12 Format of the memory_events MMIO register.

Event1 selects the actual source for the TIMER CACHE1 source. Event2 selects the source for TIMER CACHE2.

Table 3-11 . Trackable cache-performance events

Encoding	Event
0	No event counted
1	Instruction-cache misses
2	Instruction-cache stall cycles (including data-cache stall cycles if both instruction-cache and data-cache are stalled simultaneously)
3	Data-cache bank conflicts
4	Data-cache read misses
5	Data-cache write misses
6	Data-cache stall cycles (that are not also instruction-cache stall cycles)
7	Data-cache copyback to SDRAM
8	Copyback buffer full
9	Data-cache write miss with all fetch units occupied
10	Data cache stream miss
11	Prefetch operation started and not discarded

Table 3-11 . Trackable cache-performance events

Encoding	Event
12	Prefetch operation discarded (because it hits in the cache or there is no fetch unit available)
13	Prefetch operation discarded (because it hits in the cache)
14–15	Reserved

3.8 MMIO Register Summary

Table 3-12 lists the MMIO registers that pertain to the operation of TM32's instruction and data caches.

Table 3-12 . MMIO register summary

Name	Description
TM32_DRAM_LO	Sets location of the DRAM aperture
TM32_DRAM_HI	Sets size of the DRAM aperture
TM32_DRAM_CLIMIT	Divides DRAM aperture into cacheable and non-cacheable portions
MEM_EVENTS	Selects which two events will be counted by timer/counters
DC_LOCK_CTL	Data-cache locking enable and aperture control
DC_LOCK_ADDR	Sets low address of the data-cache address lock aperture
DC_LOCK_SIZE	Sets size of the data-cache address lock aperture
DC_PARAMS	Read-only register with data-cache parameter information
IC_PARAMS	Read-only register with instruction-cache parameter information
IC_LOCK_CTL	Instruction-cache locking enable
IC_LOCK_ADDR	Sets low address of the instruction-cache address lock aperture
IC_LOCK_SIZE	Sets size of the instruction-cache address lock aperture

 © TriMedia Technologies, Inc.

Chapter 4 On-Chip Semaphore Assist Device

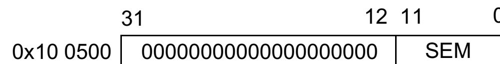
TM32 has a simple MP semaphore-assist device. It is a 32-bit register, accessible through MMIO by either the local TM32 CPU or by any other PI master. The semaphore, SEM, is located at MMIO offset 0x10 0500.

SEM operation is as follows: each master in the system constructs a personal nonzero 12 bit ID (see below). To obtain the global semaphore, a master does the following action:

```
write ID to SEM (use 32 bit store, with ID in 12 LSB)
retrieve SEM    (use 32 bit load, it returns 0x00000nnn)
if (SEM = ID) {
     "performs a short critical section action"
     write 0 to SEM
}
else "try again later, or loop back to write"
```

4.1 SEM Device Specification

SEM is a 32-bit MMIO location. The 12 LSB consist of storage flip-flops with surrounding logic, the 20 MSBs always return a ‘0’ when read.



SEM is RESET to ‘0’ by powerup reset.

When SEM is written to, the storage flip-flops behave as follows:

```
if (cur_content == 0)          new_content = write_value;
else if (write_value == 0)   new_content = 0;
/* ELSE NO ACTION ! */
```

4.2 Constructing a 12-Bit ID

A TM32 processor can construct a personal, nonzero 12-bit ID in a variety of ways. Below are some suggestions.

PCI configspace PERSONALITY entry. Each TM32 receives a 16-bit PERSONALITY value from the EEPROM during boot. This PERSONALITY register is located at offset 0x40 in configuration space. In a MP system, some of the bits of PERSONALITY can be individualized for each CPU involved, giving it a unique 2/3/4-bit ID, as needed given the maximum number of CPUs in the design.

In the case of a host-assisted TM32 boot, the PCI BIOS assigns a unique MMIO_BASE and TM32_DRAM_LO to every TM32. In particular, the 11 MSBs of each MMIO_BASE are unique, since each MMIO aperture is 2 MB in size. These bits can be used as a personality ID. Set bit 11 (MSB) to '1' to guarantee a nonzero ID#.

4.3 Which SEM to Use

Each TM32 in the system adds a SEM device to the mix. The intended use is to treat one of these SEM devices as THE master semaphore in the system. Many methods can be used to determine which SEM is master SEM. Some examples below:

Each DSPCPU32 can use PCI configuration space accesses to determine which other TM32s are present in the system. Then, the TM32 with the lowest PERSONALITY number, or the lowest MMIO_base is chosen as the TM32 containing the master semaphore.

4.4 Usage Notes

To avoid contention on the master SEM device, it should only be used for inter-processor semaphores. Processes running on a single CPU can use regular memory to implement synchronization primitives.

The critical section associated with SEM should be kept as short as possible. Preferably, SEM should only be used as the basis to make multiple memory-resident simple semaphores. In this case, the non-cacheable DRAM area of each TM32 can be used to implement the semaphore data structures efficiently.

As described here, SEM does not guarantee starvation-free access to critical resources. Claiming of SEM is purely stochastic. This should work fine as long as SEM is not overloaded. Utmost care should be taken in SEM access frequency and duration of the basic critical sections to keep the load conditions reasonable.

Appendix A Instruction Set

A.1 Alphabetic Operation List

The following table lists the complete operation set of TM32. Note that this is not an instruction list; a TM32 instruction contains from one to five of these operations.

A
alloc 3
allocd 4
allocr 5
allocx 6
asl 7
asli 8
asr 9
asri 10

B
bitand 11
bitandinv 12
bitinv 13
bitor 14
bitxor 15
borrow 16

C
carry 17
curcycles 18
cycles 19

D
dcb 20
dinvalid 21
dspiabs 22
dspiadd 23
dspidualabs 24
dspidualadd 25
dspidualmul 26
dspidualsub 27
dspimul 28
dspisub 29
dspuadd 30
dspumul 31
dspuquadaddui 32
dspusub 33
dualasr 34
dualiclipi 35
dualuclipi 36

F
fabsval 37
fabsvalflags 38
fadd 39
faddflags 40
fdiv 41
fdivflags 42
feql 43
feqlflags 44
fgeq 45
fgeqflags 46
fgtr 47
fgtrflags 48
fleq 49
fleqflags 50
fles 51
flesflags 52
fmul 53
fmulflags 54
fneq 55
fneqflags 56
fsign 57
fsignflags 58
fsqrt 59
fsqrtflags 60
fsub 61
fsubflags 62
funshift1 63
funshift2 64
funshift3 65

H
h_dspiabs 66
h_dspidualabs 67
h_iabs 68
h_st16d 69
h_st32d 70
h_st8d 71
hicycles 72

I
iabs 73
iadd 74
iaddi 75
iavgonep 76
ibytesel 77
iclipi 78
iclr 79
ident 80
ieql 81
ieqli 82
ifir16 83
ifir8ii 84
ifir8ui 85
ifixieee 86
ifixieeeflags 87
ifixrz 88
ifixrzflags 89
iflip 90
ifloat 91
ifloatflags 92
ifloatrz 93
ifloatrzflags 94
igeq 95
igeqi 96
igtr 97
igtri 98
iimm 99
ijmpf 100
ijmpi 101
ijmpt 102
ild16 103
ild16d 104
ild16r 105
ild16x 106
ild8 107
ild8d 108
ild8r 109
ileq 110
ileqi 111
iles 112
ilesi 113
imax 114
imin 115
imul 116
imulm 117
ineg 118
ineq 119
ineqi 120
inonzero 121
isub 122
isubi 123
izero 124

J
jmpf 125
jmpi 126
jmpt 127

L
ld32 128
ld32d 129
ld32r 130
ld32x 131
lsl 132
lsli 133
lsr 134
lsri 135

M
mergedual16lsb 136
mergelsb 137
mergemsb 138

N
nop 139

P
pack16lsb 140
pack16msb 141
packbytes 142
pref 143
pref16x 144
pref32x 145
prefd 146
prefr 147

Q
quadavg 148
quadumax 149
quadumin 150
quadumulmsb 151

R
rdstatus 152
rdtag 153
readdpc 154
readpcsw 155
readspc 156
rol 157
roli 158

S
sex16 159
sex8 160
st16 161
st16d 162
st32 163
st32d 164
st8 165
st8d 166

U
ubytesel 167
uclipi 168
uclipu 169
ueql 170
ueqli 171
ufir16 172
ufir8uu 173
ufixieee 174
ufixieeeflags 175
ufixrz 176
ufixrzflags 177
ufloat 178
ufloatflags 179
ufloatrz 180
ufloatrzflags 181
ugeq 182
ugeqi 183
ugtr 184
ugtri 185
uimm 186
uld16 187
uld16d 188
uld16r 189
uld16x 190
uld8 191
uld8d 192
uld8r 193
uleq 194
uleqi 195
ules 196
ulesi 197
ume8ii 198
ume8uu 199
umin 200
umul 201
umulm 202
uneq 203
uneqi 204

W
writedpc 205
writepcsw 206
writespc 207

Z
zex16 208
zex8 209

A.2 Operation List By Function

Load/Store Operations
- alloc 3
- allocd 4
- allocr 5
- allocx 6
- h_st16d 69
- h_st32d 70
- h_st8d 71
- ild16 103
- ild16d 104
- ild16r 105
- ild16x 106
- ild8 107
- ild8d 108
- ild8r 109
- ld32 128
- ld32d 129
- ld32r 130
- ld32x 131
- pref 143
- pref16x 144
- pref32x 145
- prefd 146
- prefr 147
- st16 161
- st16d 162
- st32 163
- st32d 164
- st8 165
- st8d 166
- uld16 187
- uld16d 188
- uld16r 189
- uld16x 190
- uld8 191
- uld8d 192
- uld8r 193

Shift Operations
- asl 7
- asli 8
- asr 9
- asri 10
- funshift1 63
- funshift2 64
- funshift3 65
- lsl 132
- lsli 133
- lsr 134
- lsri 135
- rol 157
- roli 158

Logical Operations
- bitand 11
- bitandinv 12
- bitinv 13
- bitor 14
- bitxor 15

DSP Operations
- dspiabs 22
- dspiadd 23
- dspidualabs 24
- dspidualadd 25
- dspidualmul 26
- dspidualsub 27
- dspimul 28
- dspisub 29
- dspuadd 30
- dspumul 31
- dspuquadaddui 32
- dspusub 33
- dualasr 34
- dualiclipi 35
- dualuclipi 36
- h_dspiabs 66
- h_dspidualabs 67
- iclipi 78
- ifir16 83
- ifir8ii 84
- ifir8ui 85
- iflip 90
- imax 114
- imin 115
- quadavg 148
- quadumax 149
- quadumin 150
- quadumulmsb 151
- uclipi 168
- uclipu 169
- ufir16 172
- ufir8uu 173
- ume8ii 198
- ume8uu 199
- umin 200

Floating-Point Arithmetic
- fabsval 37
- fabsvalflags 38
- fadd 39
- faddflags 40
- fdiv 41
- fdivflags 42
- fmul 53
- fmulflags 54
- fsign 57
- fsignflags 58
- fsqrt 59
- fsqrtflags 60
- fsub 61
- fsubflags 62

Floating-Point Conversion
- ifixieee 86
- ifixieeeflags 87
- ifixrz 88
- ifixrzflags 89
- ifloat 91
- ifloatflags 92
- ifloatrz 93
- ifloatrzflags 94
- ufixieee 174
- ufixieeeflags 175
- ufixrz 176
- ufixrzflags 177
- ufloat 178
- ufloatflags 179
- ufloatrz 180
- ufloatrzflags 181

Floating-Point Relationals
- feql 43
- feqlflags 44
- fgeq 45
- fgeqflags 46
- fgtr 47
- fgtrflags 48
- fleq 49
- fleqflags 50
- fles 51
- flesflags 52
- fneq 55
- fneqflags 56

Integer Arithmetic
- borrow 16
- carry 17
- h_iabs 68
- iabs 73
- iadd 74
- iaddi 75
- iavgonep 76
- ident 80
- imul 116
- imulm 117
- ineg 118
- inonzero 121
- isub 122
- isubi 123
- izero 124
- umul 201
- umulm 202

Immediate Operations
- iimm 99
- uimm 186

Sign/Zero Extend Ops
- sex16 159
- sex8 160
- zex16 208
- zex8 209

Integer Relationals
- ieql 81
- ieqli 82
- igeq 95
- igeqi 96
- igtr 97
- igtri 98
- ileq 110
- ileqi 111
- iles 112
- ilesi 113
- ineq 119
- ineqi 120
- ueql 170
- ueqli 171
- ugeq 182
- ugeqi 183
- ugtr 184
- ugtri 185
- uleq 194
- uleqi 195
- ules 196
- ulesi 197
- uneq 203
- uneqi 204

Control-Flow Operations
- ijmpf 100
- ijmpi 101
- ijmpt 102
- jmpf 125
- jmpi 126
- jmpt 127

Special-Register Ops
- cycles 19
- curcycles 18
- hicycles 72
- nop 139
- readdpc 154
- readpcsw 155
- readspc 156
- writedpc 205
- writepcsw 206
- writespc 207

Cache Operations
- dcb 20
- dinvalid 21
- iclr 79
- rdstatus 152
- rdtag 153

Pack/Merge/Select Ops
- ibytesel 77
- mergedual16lsb 136
- mergelsb 137
- mergemsb 138
- pack16lsb 140
- pack16msb 141
- packbytes 142
- ubytesel 167

alloc

Allocate a cache block pseudo-op for allocd(0)

SYNTAX

```
[ IF rguard ] alloc(d) rsrc1
```

FUNCTION

if r*guard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + 0) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	213
Number of operands	1
Modifier	-
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocr allocx

DESCRIPTION

The alloc operation is a pseudo operation transformed by the scheduler into an allocd(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The alloc operation allocate a cache block with the address computed from **[(rsrc1 + 0) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the 'cache architecture' section for details on the cache block size.

The alloc operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the alloc operation. If the LSB of rguard is 1, alloc operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`alloc r10`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 alloc r10`	since guard is false, alloc operation is not executed
r10 = 0xac0f, r11 = 1, cache_block_size = 0x40	`IF r11 alloc r10`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

allocd
Allocate a cache block with displacement

SYNTAX
```
[ IF rguard ] allocd(d) rsrc1
```

FUNCTION
if *rguard* **then {**
cache_block_mask = ~(cache_block_size -1)]
allocate adata cache block with [(rsrc1 + d) & cache_block_mask] address
}

ATTRIBUTES

Function unit	dmemspec
Operation code	213
Number of operands	1
Modifier	7 bits
Modifier range	-255..252 by 4
Latency	-
Issue slots	5

SEE ALSO
allocr allocx

DESCRIPTION
The allocd operation allocate a cache block with the address computed from **[(rsrc1 + d) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the 'cache architecture' section for details on the cache block size.

The `allocd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocd operation. If the LSB of r*guard* is 1, allocd operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`allocd(0x32) r10`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 allocd(0x32) r10`	since guard is false, allocd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 allocd(0x4) r10`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

 © TriMedia Technologies, Inc.

Allocate a cache block with index

allocr

SYNTAX

```
[ IF rguard ] allocr rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size -1)]
    allocate adata cache block with [(rsrc1 + rsrc2) & cache_block_mask] address
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	214
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocx

DESCRIPTION

The allocr operation allocate a cache block with the address computed from **[(rsrc1 + rscr2) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the 'cache architecture' section for details on the cache block size.

The `allocr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocr operation. If the LSB of r*guard* is 1, allocr operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0x32 cache_block_size = 0x40	`allocr r10 r12`	Allocates a cache block for the address space from 0xabc0 to 0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, r12=0x32, cache_block_size = 0x40	`IF r11 allocr r10 r12`	since guard is false, allocr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x4, cache_block_size = 0x40	`IF r11 allocr r10 r12`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

allocx

Allocate a cache block with scaled index

SYNTAX

`[ IF rguard ] allocx rsrc1 rsrc2`

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size -1)]
    allocate adata cache blockwith [(rsrc1 + 4 x rsrc2) & cache_block_mask] address
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	215
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

allocd allocr

DESCRIPTION

The allocx operation allocate a cache block with the address computed from **[(rsrc1 + 4 x rscr2) & cache_block_mask]** and sets the status of this cache block as valid. No data is fetched from main memory for this operation. The allocated cache block data is undefined after this operation. It is the responsibility of the programmer to update the allocated cache block by store operations.

Refer to the 'cache architecture' section for details on the cache block size.

The `allocx` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the allocx operation. If the LSB of r*guard* is 1, allocx operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xc cache_block_size = 0x40	`allocx r10 r12`	Allocates a cache block for the address space from 0xabc0 to 0x0xabff without fetching the data from main memory; The data in this address space is undefined.
r10 = 0xabcd, r11 = 0, r12=0xc, cache_block_size = 0x40	`IF r11 allocx r10 r12`	since guard is false, allocx operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x4, cache_block_size = 0x40	`IF r11 allocx r10 r12`	Allocates a cache block for the address space from 0xac00 to 0xac3f without fetching the data from main memory; the data in this address space is undefined.

© TriMedia Technologies, Inc.

Arithmetic shift left

asl

SYNTAX

```
[ IF rguard ] asl rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    n ← rsrc2<4:0>
    rdest<31:n> ← rsrc1<31–n:0>
    rdest<n–1:0> ← 0
    if rsrc2<31:5> != 0 {
        rdest <- 0
    }
}
```

ATTRIBUTES

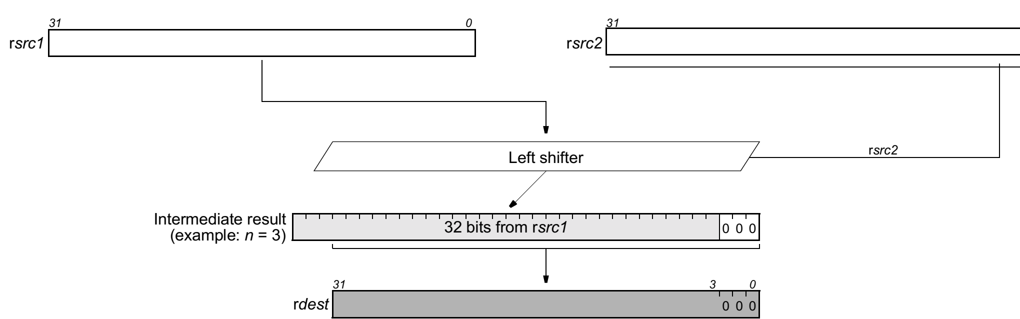
Function unit	shifter
Operation code	19
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asli asr asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asl` operation takes two arguments, r*src1* and r*src2*. R*src2* specify an unsigned shift amount, and r*dest* is set to r*src1* arithmetically shifted left by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `asl` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`asl r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 asl r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 asl r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`asl r70 r40 → r120`	r120 ← 0xfffffff0
r80 = 0xe, r50 = 0xfffffffe	`asl r80 r50 → r125`	r125 ← 0x00000000 (shift by more than 32)
r30 = 0x7008000f, r60 = 0x20	`asl r30 r60 → r111`	r111 ← 0x00000000
r30 = 0x8008000f, r45 = 0x80000000	`asl r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x23	`asl r30 r45 → r100`	r100 ← 0x00000000

asli   Arithmetic shift left immediate

SYNTAX

```
[ IF rguard ] asli(n) rsrc1 → rdest
```

FUNCTION

if *rguard* **then {**
    r*dest*<31:*n*> ← r*src1*<31–*n*:0>
    r*dest*<*n*–1:0> ← 0
}

ATTRIBUTES

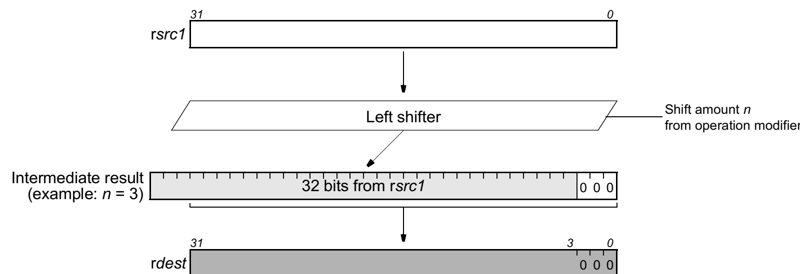
Function unit	shifter
Operation code	11
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asr asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* arithmetically shifted left by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `asli` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`asli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 asli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 asli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`asli(2) r70 → r120`	r120 ← 0xfffffff0
r80 = 0xe	`asli(30) r80 → r125`	r125 ← 0x80000000

© TriMedia Technologies, Inc.

Arithmetic shift right

asr

SYNTAX

```
[ IF rguard ] asr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    n ← rsrc2<4:0>
    rdest<31:31–n> ← rsrc1<31>
    rdest<30–n:0> ← rsrc1<30:n>
    if rsrc2<31:5> != 0 {
        rdest <- rsrc1<31>
    }
}
```

ATTRIBUTES

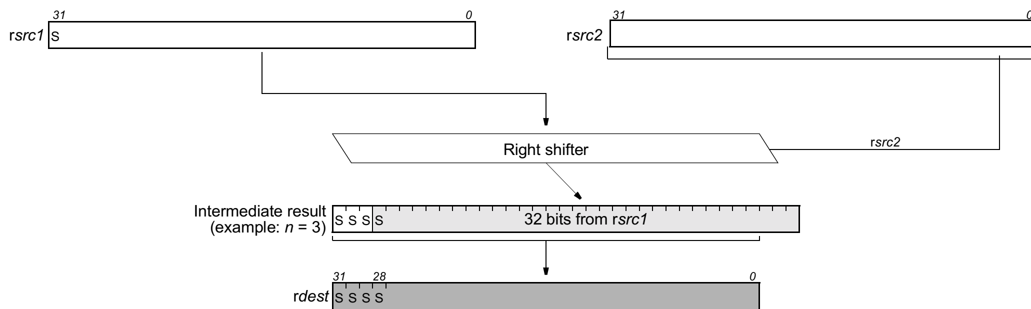
Function unit	shifter
Operation code	18
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asri lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asr` operation takes two arguments, r*src1* and r*src2*. R*src2* specifies an unsigned shift amount, and r*src1* is arithmetically shifted right by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The `asr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f, r20 = 1	`asr r30 r20 → r50`	r50 ← 0x38040007
r30 = 0x7008000f, r42 = 2	`asr r30 r42 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f, r44 = 4	`IF r10 asr r30 r44 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f, r44 = 4	`IF r20 asr r30 r44 → r80`	r80 ← 0x07008000
r40 = 0x80030007, r44 = 4	`asr r40 r44 → r90`	r90 ← 0xf8003000
r30 = 0x7008000f, r45 = 0x1f	`asr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x1f	`asr r30 r45 → r100`	r100 ← 0xffffffff
r30 = 0x7008000f, r45 = 0x20	`asr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x20	`asr r30 r45 → r100`	r100 ← 0xffffffff
r30 = 0x8008000f, r45 = 0x23	`asr r30 r45 → r100`	r100 ← 0xffffffff

asri
Arithmetic shift right by immediate amount

SYNTAX

`[ IF rguard ] asri(n) rsrc1 → rdest`

FUNCTION

if *rguard* **then {**
  rdest<31:31–*n*> ← *rsrc1*<31>
  rdest<30–*n*:0> ← *rsrc1*<31:*n*>
}

ATTRIBUTES

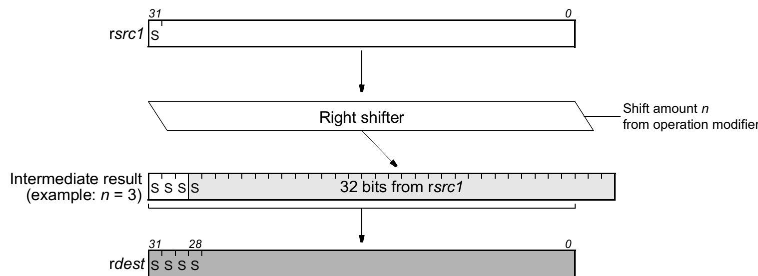
Function unit	shifter
Operation code	10
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr lsl lsli lsr lsri rol roli

DESCRIPTION

As shown below, the `asri` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* arithmetically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. The MSB (sign bit) of r*src1* is replicated as needed to fill vacated bits from the left.



The `asri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	`asri(1) r30 → r50`	r50 ← 0x38040007
r30 = 0x7008000f	`asri(2) r30 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	`IF r10 asri(4) r30 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f	`IF r20 asri(4) r30 → r80`	r80 ← 0x07008000
r40 = 0x80030007	`asri(4) r40 → r90`	r90 ← 0xf8003000
r30 = 0x7008000f	`asri(31) r30 → r100`	r100 ← 0x00000000
r40 = 0x80030007	`asri(31) r40 → r110`	r110 ← 0xffffffff

 © TriMedia Technologies, Inc.

Bitwise logical AND

bitand

SYNTAX

`[ IF rguard ] bitand rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
 r*dest* ← r*src1* & r*src2*

ATTRIBUTES

Function unit	alu
Operation code	16
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitor bitxor bitandinv

DESCRIPTION

The `bitand` operation computes the bitwise, logical AND of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitand` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitand r30 r40 → r90`	r90 ← 0xf3100000
r10 = 0, r50 = 0x88888888	`IF r10 bitand r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitand r30 r50 → r100`	r100 ← 0x80008888
r60 = 0x11119999, r50 = 0x88888888	`bitand r60 r50 → r110`	r110 ← 0x00008888
r70 = 0x55555555, r30 = 0xf310ffff	`bitand r70 r30 → r120`	r120 ← 0x51105555

bitandinv

Bitwise logical AND NOT

SYNTAX

```
[ IF rguard ] bitandinv rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* & ~r*src2*

ATTRIBUTES

Attribute	Value
Function unit	alu
Operation code	49
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitor bitxor

DESCRIPTION

The `bitandinv` operation computes the bitwise, logical AND of the first argument, r*src1*, with the 1's complement of the second argument, r*src2*. The result is stored in the destination register, r*dest*.

The `bitandinv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitandinv r30 r40 → r90`	r90 ← 0x0000ffff
r10 = 0, r50 = 0x88888888	`IF r10 bitandinv r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitandinv r30 r50 → r100`	r100 ← 0x73107777
r60 = 0x11119999, r50 = 0x88888888	`bitandinv r60 r50 → r110`	r110 ← 0x11111111
r70 = 0x55555555, r30 = 0xf310ffff	`bitandinv r70 r30 → r120`	r120 ← 0x04450000

© TriMedia Technologies, Inc.

Bitwise logical NOT

bitinv

SYNTAX

```
[ IF rguard ] bitinv rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ~r*src1*

ATTRIBUTES

Function unit	alu
Operation code	50
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitor bitxor

DESCRIPTION

The `bitinv` operation computes the bitwise, logical NOT of the argument r*src1* and writes the result into r*dest*.

The `bitinv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff	`bitinv r30 → r60`	r60 ← 0x0cef0000
r10 = 0, r40 = 0xffff0000	`IF r10 bitinv r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xffff0000	`IF r20 bitinv r40 → r100`	r100 ← 0x0000ffff
r50 = 0x88888888	`bitinv r50 → r110`	r110 ← 0x77777777

bitor

Bitwise logical OR

SYNTAX

```
[ IF rguard ] bitor rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← *rsrc1* | *rsrc2*

ATTRIBUTES

Function unit	alu
Operation code	17
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

bitand bitandinv bitinv bitxor

DESCRIPTION

The `bitor` operation computes the bitwise, logical OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitor r30 r40 → r90`	r90 ← 0xffffffff
r10 = 0, r50 = 0x88888888	`IF r10 bitor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitor r30 r50 → r100`	r100 ← 0xfb98ffff
r60 = 0x11119999, r50 = 0x88888888	`bitor r60 r50 → r110`	r110 ← 0x99999999
r70 = 0x55555555, r30 = 0xf310ffff	`bitor r70 r30 → r120`	r120 ← 0xf755ffff

 © TriMedia Technologies, Inc.

Bitwise logical exclusive-OR

bitxor

SYNTAX

```
[ IF rguard ] bitxor rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* ⊕ r*src2*

ATTRIBUTES

Function unit	alu
Operation code	48
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`bitand` `bitandinv` `bitinv` `bitor`

DESCRIPTION

The `bitxor` operation computes the bitwise, logical exclusive-OR of the first and second arguments, r*src1* and r*src2*. The result is stored in the destination register, r*dest*.

The `bitxor` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf310ffff, r40 = 0xffff0000	`bitxor r30 r40 → r90`	r90 ← 0x0cefffff
r10 = 0, r50 = 0x88888888	`IF r10 bitxor r30 r50 → r80`	no change, since guard is false
r20 = 1, r30 = 0xf310ffff, r50 = 0x88888888	`IF r20 bitxor r30 r50 → r100`	r100 ← 0x7b987777
r60 = 0x11119999, r50 = 0x88888888	`bitxor r60 r50 → r110`	r110 ← 0x99991111
r70 = 0x55555555, r30 = 0xf310ffff	`bitxor r70 r30 → r120`	r120 ← 0xa645aaaa

borrow

Compute borrow bit from unsigned subtract

pseudo-op for ugtr

SYNTAX

```
[ IF rguard ] borrow rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* < r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`ugtr` `carry`

DESCRIPTION

The `borrow` operation is a pseudo operation transformed by the scheduler into an `ugtr` with reversed arguments. (Note: pseudo operations cannot be used in assembly source files.)

The `borrow` operation computes the unsigned difference of the first and second arguments, r*src1*–r*src2*. If the difference generates a borrow (if r*src2* > r*src1*), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `borrow` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 2, r30 = 0xfffffffc	`borrow r70 r30 → r80`	r80 ← 1
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 borrow r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 borrow r70 r30 → r100`	r100 ← 1
r60 = 4, r30 = 0xfffffffc	`borrow r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`borrow r30 r30 → r120`	r120 ← 0

© TriMedia Technologies, Inc.

Compute carry bit from unsigned add

carry

SYNTAX

```
[ IF rguard ] carry rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if (rsrc1+rsrc2) < 2^32 then
        rdest ← 0
    else
        rdest ← 1
}
```

ATTRIBUTES

Function unit	alu
Operation code	45
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

borrow

DESCRIPTION

The `carry` operation computes the unsigned sum of the first and second arguments, r*src1*+r*src2*. If the sum generates a carry (if the sum is greater than 2^{32}-1), 1 is stored in the destination register, r*dest*; otherwise, r*dest* is set to 0.

The `carry` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 2, r30 = 0xfffffffc	`carry r70 r30 → r80`	r80 ← 0
r10 = 0, r70 = 2, r30 = 0xfffffffc	`IF r10 carry r70 r30 → r90`	no change, since guard is false
r20 = 1, r70 = 2, r30 = 0xfffffffc	`IF r20 carry r70 r30 → r100`	r100 ← 0
r60 = 4, r30 = 0xfffffffc	`carry r60 r30 → r110`	r110 ← 1
r30 = 0xfffffffc	`carry r30 r30 → r120`	r120 ← 1

curcycles

Read current clock cycle counter, least-significant word

SYNTAX

`[ IF rguard ] curcycles → rdest`

FUNCTION

if *rguard* **then**
rdest ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	162
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`cycles` `hicycles` `writepcsw`

DESCRIPTION

Refer to Section 1.1.5, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The `curcycles` operation copies the current low 32 bits of the master Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*.. The master CCCOUNT increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `curcycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`curcycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 curcycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 curcycles → r100`	r100 ← 0x12345678

© TriMedia Technologies, Inc.

Read clock cycle counter, least-significant word

cycles

SYNTAX

```
[ IF rguard ] cycles → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← CCCOUNT<31:0>

ATTRIBUTES

Function unit	fcomp
Operation code	154
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

hicycles curcycles writepcsw

DESCRIPTION

Refer to Section 1.1.5, “CCCOUNT—Clock Cycle Counter” for a description of the CCCOUNT operation. The `cycles` operation copies the low 32 bits of the slave register of Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if `cycles` and `hicycles` are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The `cycles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	`cycles → r60`	r30 ← 0x12345678
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	`IF r10 cycles → r70`	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	`IF r20 cycles → r100`	r100 ← 0x12345678

dcb — Data cache copy back

SYNTAX

```
[ IF rguard ] dcb(d) rsrc1
```

ATTRIBUTES

Function unit	dmemspec
Operation code	205
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

FUNCTION

```
if rguard then {
    addr ← rsrc1 + d
    if dcache_valid_addr(addr) && dcache_dirty_addr(addr) then {
        dcache_copyback_addr(addr)
        dcache_reset_dirty_addr(addr)
    }
}
```

SEE ALSO

`dinvalid`

DESCRIPTION

The `dcb` operation causes a block in the data cache to be copied back to main memory if the block is marked dirty and valid, and the block's dirty bit is reset. The target block of `dcb` is the block in the data cache that contains the byte addressed by r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

A valid copy of the target block remains in the cache. Stall cycles are taken as necessary to complete the copy-back operation. If the target block is not dirty or if the block is not in the cache, `dcb` has no effect and no stall cycles are taken.

`dcb` has no effect on blocks that are in the non-cacheable SDRAM aperture. `dcb` does not change the replacement status of data-cache blocks.

`dcb` ensures coherency between caches and main memory by discarding all pending prefetch operations and by causing all non-empty copyback buffers to be emptied to main memory.

The `dcb` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls if the operation is carried out or not.If the LSB of r*guard* is 1, the operation is carried out; otherwise,it is not carried out.

EXAMPLES

Initial Values	Operation	Result
	`dcb(0) r30`	
r10 = 0	`IF r10 dcb(4) r40`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 dcb(8) r50`	

 © TriMedia Technologies, Inc.

Invalidate data cache block

dinvalid

SYNTAX

```
[ IF rguard ] dinvalid(d) rsrc1
```

FUNCTION

if r*guard* **then {**
addr ← r*src1* + *d*
if dcache_valid_addr(addr) **then {**
dcache_reset_valid_addr(addr)
dcache_reset_dirty_addr(addr)
}
}

ATTRIBUTES

Function unit	dmemspec
Operation code	206
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

dcb

DESCRIPTION

The `dinvalid` operation resets the valid and dirty bit of a block in the data cache. Regardless of the block's dirty bit, the block is not written back to main memory. The target block of `dinvalid` is the block in the data cache that contains the byte addressed by r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

Stall cycles are taken as necessary to complete the invalidate operation. If the target block is not in the cache, `dinvalid` has no effect and no stall cycles are taken.

`dinvalid` has no effect on blocks that are in the non-cacheable SDRAM aperture. `dinvalid` does clear the valid bits of locked blocks. `dinvalid` does not change the replacement status of data-cache blocks.

`dinvalid` ensures coherency between caches and main memory by discarding all pending prefetch operations and by causing all non-empty copyback buffers to be emptied to main memory.

The `dinvalid` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls if the operation is carried out or not. If the LSB of r*guard* is 1, the operation is carried out; otherwise, it is not carried out.

EXAMPLES

Initial Values	Operation	Result
	`dinvalid(0) r30`	
r10 = 0	`IF r10 dinvalid(4) r40`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 dinvalid(8) r50`	

dspiabs

Clipped signed absolute value

pseudo-op for h_dspiabs

SYNTAX

```
[ IF rguard ] dspiabs rsrc1 → rdest
```

FUNCTION

if *rguard* **then {**
  if r*src1* >= 0 **then**
    r*dest* ← r*src1*
  else if rsrc1 = 0x80000000 **then**
    r*dest* ← 0x7fffffff
  else
    r*dest* ← –r*src1*
}

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	1
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspiabs h_dspidualabs dspiadd dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

The dspiabs operation is a pseudo operation transformed by the scheduler into an h_dspiabs with a constant first argument zero and second argument equal to the dspiabs argument. (Note: pseudo operations cannot be used in assembly source files.)

The dspiabs operation computes the absolute value of rsrc1, clips the result into the range [2^{31}–1..0] (or [0x7fffffff..0]), and stores the clipped value into r*dest*. All values are signed integers.

The dspiabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	dspiabs r30 → r60	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	IF r10 dspiabs r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80000001	IF r20 dspiabs r40 → r100	r100 ← 0x7fffffff
r50 = 0x80000000	dspiabs r50 → r80	r80 ← 0x7fffffff
r90 = 0x7fffffff	dspiabs r90 → r110	r110 ← 0x7fffffff

© TriMedia Technologies, Inc.

Clipped signed add

dspiadd

SYNTAX

```
[ IF rguard ] dspiadd rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
temp ← sign_ext32to64(*rsrc1*) + sign_ext32to64(*rsrc2*)
if temp < 0xffffffff80000000 **then**
rdest ← 0x80000000
else if temp > 0x000000007fffffff **then**
rdest ← 0x7fffffff
else
rdest ← temp
}

ATTRIBUTES

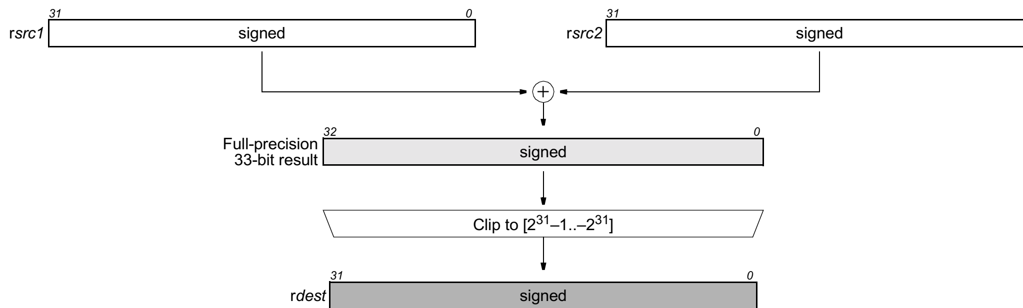
Function unit	dspalu
Operation code	66
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspimul dspisub
dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspiadd operation computes the sum *rsrc1*+*rsrc2*, clips the result into the 32-bit signed range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into *rdest*. All values are signed integers.



The dspiadd operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspiadd r30 r40 → r60	r60 ← 0x12ff
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspiadd r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspiadd r30 r40 → r100	r100 ← 0x12ff
r50 = 0x7fffffff, r90 = 1	dspiadd r50 r90 → r110	r110 ← 0x7fffffff
r70 = 0x80000000, r80 = 0xffffffff	dspiadd r70 r80 → r120	r120 ← 0x80000000

dspidualabs

Dual clipped absolute value of signed 16-bit halfwords

pseudo-op for h_dspidualabs

SYNTAX

`[ IF rguard ] dspidualabs rsrc1 → rdest`

FUNCTION

if r*guard* **then {**
  temp1 ← sign_ext16to32(r*src1*<15:0>)
  temp2 ← sign_ext16to32(r*src1*<31:16>)
  if temp1 = 0xffff8000 **then** temp1 ← 0x7fff
  if temp2 = 0xffff8000 **then** temp2 ← 0x7fff
  if temp1 < 0 **then** temp1 ← –temp1
  if temp2 < 0 **then** temp2 ← –temp2
  r*dest*<31:16> ← temp2<15:0>
  r*dest*<15:0> ← temp1<15:0>
}

ATTRIBUTES

Function unit	dspalu
Operation code	72
Number of operands	1
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspidualabs dspiabs dspidualadd dspidualmul dspidualsub

DESCRIPTION

The `dspidualabs` operation is a pseudo operation transformed by the scheduler into an h_`dspidualabs` with a constant zero as first argument and the `dspidualabs` argument as second argument. (Note: pseudo operations cannot be used in assembly source files.)

The `dspidualabs` operation performs two 16-bit clipped, signed absolute value computations separately on the high and low 16-bit halfwords of r*src1*. Both absolute values are clipped into the range [0x0..0x7fff] and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.

The `dspidualabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0032	`dspidualabs r30 → r60`	r60 ← 0x00010032
r10 = 0, r40 = 0x80008001	`IF r10 dspidualabs r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0x80008001	`IF r20 dspidualabs r40 → r100`	r100 ← 0x7fff7fff
r50 = 0x0032ffff	`dspidualabs r50 → r80`	r80 ← 0x00320001
r90 = 0x7fffffff	`dspidualabs r90 → r110`	r110 ← 0x7fff0001

© TriMedia Technologies, Inc.

Dual clipped add of signed 16-bit halfwords

dspidualadd

SYNTAX

```
[ IF rguard ] dspidualadd rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
temp1 ← sign_ext16to32(r*src1*<15:0>) + sign_ext16to32(r*src2*<15:0>)
temp2 ← sign_ext16to32(r*src1*<31:16>) + sign_ext16to32(r*src2*<31:16>)
if temp1 < 0xffff8000 **then** temp1 ← 0x8000
if temp2 < 0xffff8000 **then** temp2 ← 0x8000
if temp1 > 0x7fff **then** temp1 ← 0x7fff
if temp2 > 0x7fff **then** temp2 ← 0x7fff
r*dest*<31:16> ← temp2<15:0>
r*dest*<15:0> ← temp1<15:0>
}

ATTRIBUTES

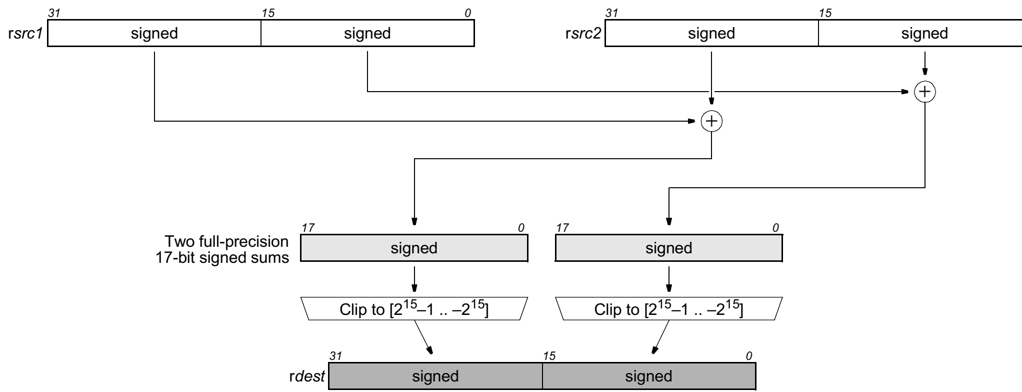
Function unit	dspalu
Operation code	70
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspidualmul dspidualsub dspiabs

DESCRIPTION

As shown below, the `dspidualadd` operation computes two 16-bit clipped, signed sums separately on the two pairs of high and low 16-bit halfwords of r*src1* and r*src2*. Both sums are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers.



The `dspidualadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12340032, r40 = 0x00010002	`dspidualadd r30 r40 → r60`	r60 ← 0x12350034
r10 = 0, r30 = 0x12340032, r40 = 0x00010002	`IF r10 dspidualadd r30 r40 → r70`	no change, since guard is false
r20 = 1, r30 = 0x12340032, r40 = 0x00010002	`IF r20 dspidualadd r30 r40 → r100`	r100 ← 0x12350034
r50 = 0x80000001, r80 = 0xffff7fff	`dspidualadd r50 r80 → r90`	r90 ← 0x80007fff
r110 = 0x00017fff, r120 = 0x7fff7fff	`dspidualadd r110 r120 → r125`	r125 ← 0x7fff7fff

dspidualmul

Dual clipped multiply of signed 16-bit halfwords

SYNTAX

```
[ IF rguard ] dspidualmul rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
temp1 ← sign_ext16to32(*rsrc1*<15:0>) × sign_ext16to32(*rsrc2*<15:0>)
temp2 ← sign_ext16to32(*rsrc1*<31:16>) × sign_ext16to32(*rsrc2*<31:16>)
if temp1 < 0xffff8000 **then** temp1 ← 0x8000
if temp2 < 0xffff8000 **then** temp2 ← 0x8000
if temp1 > 0x7fff **then** temp1 ← 0x7fff
if temp2 > 0x7fff **then** temp2 ← 0x7fff
rdest<31:16> ← temp2<15:0>
rdest<15:0> ← temp1<15:0>
}

ATTRIBUTES

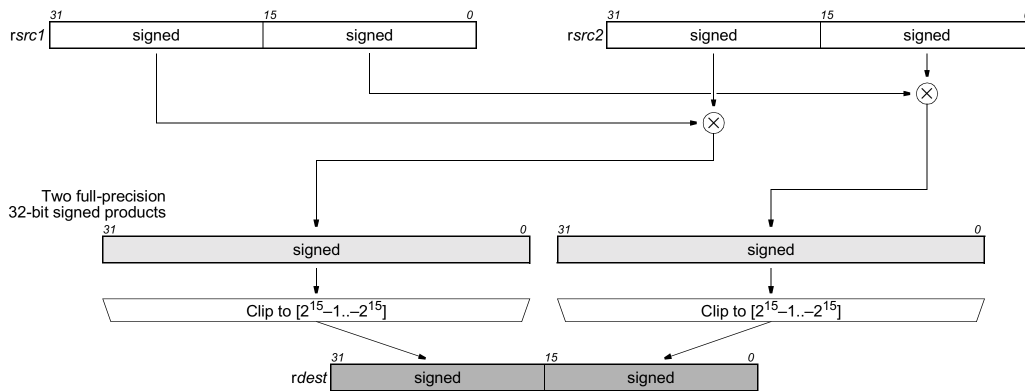
Function unit	dspmul
Operation code	95
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

dspidualabs dspidualadd dspidualsub dspiabs

DESCRIPTION

As shown below, the `dspidualmul` operation computes two 16-bit clipped, signed products separately on the two pairs of high and low 16-bit halfwords of *rsrc1* and *rsrc2*. Both products are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of *rdest*. All values are signed 16-bit integers.



The `dspidualmul` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0020010, r40 = 0x00030020	`dspidualmul r30 r40 → r60`	r60 ← 0x00060200
r10 = 0, r30 = 0x0020010, r40 = 0x00030020	`IF r10 dspidualmul r30 r40 → r70`	no change, since guard is false
r20 = 1, r30 = 0x0020010, r40 = 0x00030020	`IF r20 dspidualmul r30 r40 → r100`	r100 ← 0x00060200
r50 = 0x80000002, r80 = 0x00024000	`dspidualmul r50 r80 → r90`	r90 ← 0x80007fff
r110 = 0x08000003, r120 = 0x00108001	`dspidualmul r110 r120 → r125`	r125 ← 0x7fff8000

© TriMedia Technologies, Inc.

Dual clipped subtract of signed 16-bit halfwords

dspidualsub

SYNTAX

```
[ IF rguard ] dspidualsub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp1 ← sign_ext16to32(rsrc1<15:0>) – sign_ext16to32(rsrc2<15:0>)
    temp2 ← sign_ext16to32(rsrc1<31:16>) – sign_ext16to32(rsrc2<31:16>)
    if temp1 < 0xffff8000 then temp1 ← 0x8000
    if temp2 < 0xffff8000 then temp2 ← 0x8000
    if temp1 > 0x7fff then temp1 ← 0x7fff
    if temp2 > 0x7fff then temp2 ← 0x7fff
    rdest<31:16> ← temp2<15:0>
    rdest<15:0> ← temp1<15:0>
}
```

ATTRIBUTES

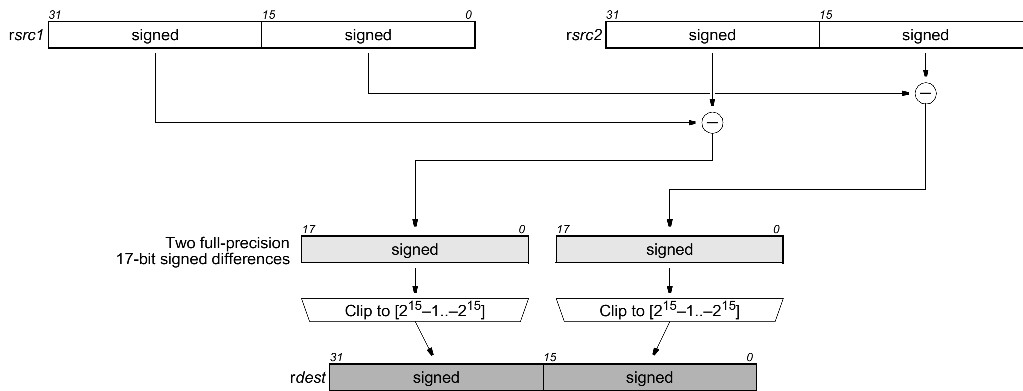
Function unit	dspalu
Operation code	71
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspidualadd dspidualmul dspiabs

DESCRIPTION

As shown below, the dspidualsub operation computes two 16-bit clipped, signed differences separately on the two pairs of high and low 16-bit halfwords of *rsrc1* and *rsrc2*. Both differences are clipped into the range [2^{15}–1..–2^{15}] (or [0x7fff..0x8000]) and written into the corresponding halfwords of *rdest*. All values are signed 16-bit integers.



The dspidualsub operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12340032, r40 = 0x00010002	dspidualsub r30 r40 → r60	r60 ← 0x12330030
r10 = 0, r30 = 0x12340032, r40 = 0x00010002	IF r10 dspidualsub r30 r40 → r70	no change, since guard is false
r20 = 1, r30 = 0x12340032, r40 = 0x00010002	IF r20 dspidualsub r30 r40 → r100	r100 ← 0x12330030
r50 = 0x80000001, r80 = 0x00018001	dspidualsub r50 r80 → r90	r90 ← 0x80007fff
r110 = 0x00018001, r120 = 0x80010002	dspidualsub r110 r120 → r125	r125 ← 0x7fff8000

dspimul

Clipped signed multiply

SYNTAX

```
[ IF rguard ] dspimul rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	ifmul
Operation code	141
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

FUNCTION

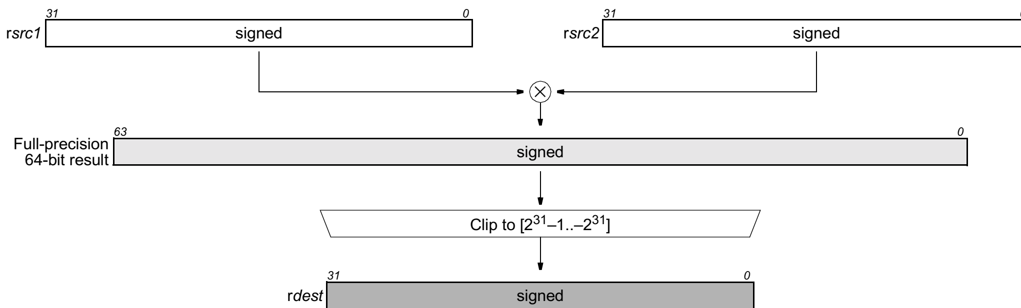
```
if rguard then {
    temp ← sign_ext32to64(rsrc1) × sign_ext32to64(rsrc2)
    if temp < 0xffffffff80000000 then
        rdest ← 0x80000000
    else if temp > 0x000000007fffffff then
        rdest ← 0x7fffffff
    else
        rdest ← temp<31:0>
}
```

SEE ALSO

dspiabs dspiadd dspisub dspuadd dspumul dspusub

DESCRIPTION

As shown below, the `dspimul` operation computes the product *rsrc1*×*rsrc2*, clips the result into the 32-bit range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into *rdest*. All values are signed integers.



The `dspimul` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x10, r40 = 0x20	`dspimul r30 r40 → r60`	r60 ← 0x200
r10 = 0, r30 = 0x10, r40 = 0x20	`IF r10 dspimul r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x10, r40 = 0x20	`IF r20 dspimul r30 r40 → r100`	r100 ← 0x200
r50 = 0x40000000, r90 = 2	`dspimul r50 r90 → r110`	r110 ← 0x7fffffff
r80 = 0xffffffff	`dspimul r80 r80 → r120`	r120 ← 0x1
r70 = 0x80000000, r90 = 2	`dspimul r70 r90 → r120`	r120 ← 0x80000000

© TriMedia Technologies, Inc.

Clipped signed subtract

dspisub

SYNTAX

```
[ IF rguard ] dspisub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← sign_ext32to64(rsrc1) – sign_ext32to64(rsrc2)
    if temp < 0xffffffff80000000 then
        rdest ← 0x80000000
    else if temp > 0x000000007fffffff then
        rdest ← 0x7fffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

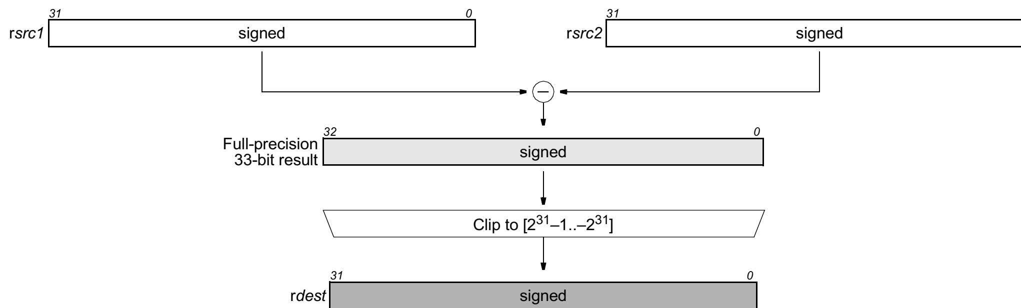
Function unit	dspalu
Operation code	68
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul dspuadd dspumul dspusub

DESCRIPTION

As shown below, the dspisub operation computes the difference *rsrc1*–*rsrc2*, clips the result into the 32-bit range [2^{31}–1..–2^{31}] (or [0x7fffffff..0x80000000]), and stores the clipped value into *rdest*. All values are signed integers.



The dspisub operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	`dspisub r30 r40 → r60`	r60 ← 0x1101
r10 = 0, r30 = 0x1200, r40 = 0xff	`IF r10 dspisub r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	`IF r20 dspisub r30 r40 → r100`	r100 ← 0x1101
r50 = 0x7fffffff, r90 = 0xffffffff	`dspisub r50 r90 → r110`	r110 ← 0x7fffffff
r70 = 0x80000000, r80 = 1	`dspisub r70 r80 → r120`	r120 ← 0x80000000

dspuadd

Clipped unsigned add

SYNTAX

```
[ IF rguard ] dspuadd rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) + zero_ext32to64(rsrc2)
    if (unsigned)temp > 0x00000000ffffffff then
        rdest ← 0xffffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

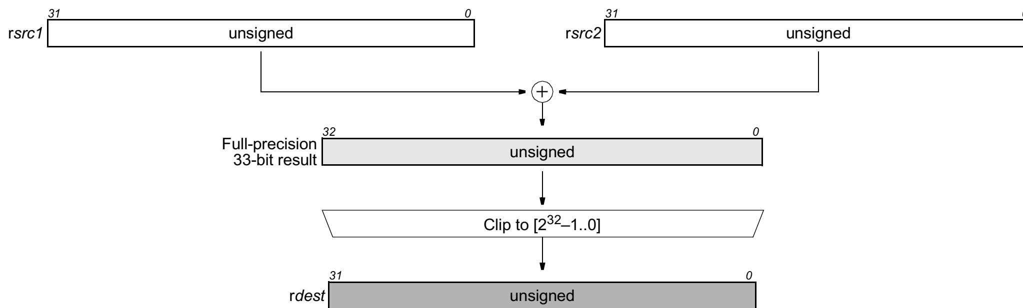
Function unit	dspalu
Operation code	67
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul dspisub dspumul dspusub

DESCRIPTION

As shown below, the `dspuadd` operation computes unsigned sum r*src1*+r*src2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into r*dest*.



The `dspuadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	`dspuadd r30 r40 → r60`	r60 ← 0x12ff
r10 = 0, r30 = 0x1200, r40 = 0xff	`IF r10 dspuadd r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	`IF r20 dspuadd r30 r40 → r100`	r100 ← 0x12ff
r50 = 0xffffffff, r90 = 1	`dspuadd r50 r90 → r110`	r110 ← 0xffffffff
r70 = 0x80000001, r80 = 0x7fffffff	`dspuadd r70 r80 → r120`	r120 ← 0xffffffff

Clipped unsigned multiply

dspumul

SYNTAX

```
[ IF rguard ] dspumul rsrc1 rsrc2 → rdest
```

OPERATION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) × zero_ext32to64(rsrc2)
    if (unsigned)temp > 0x00000000ffffffff then
        rdest ← 0xffffffff
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

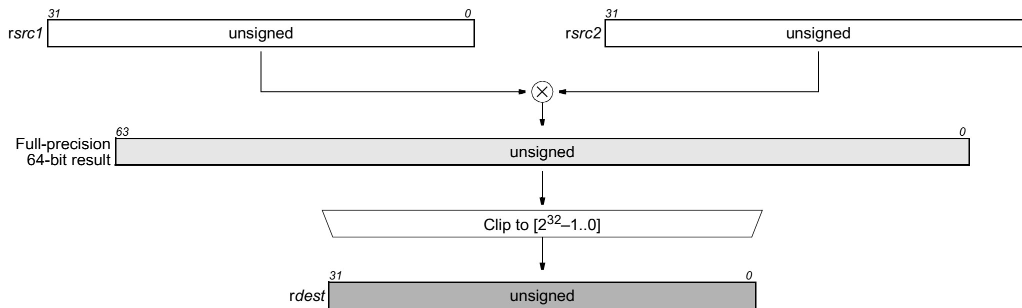
Function unit	ifmul
Operation code	142
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

dspiabs dspiadd dspisub
dspuadd dspumul dspusub

DESCRIPTION

As shown below, the `dspumul` operation computes unsigned product *rsrc1*×*rsrc2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into *rdest*.



The `dspumul` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x10, r40 = 0x20	`dspumul r30 r40 → r60`	r60 ← 0x200
r10 = 0, r30 = 0x10, r40 = 0x20	`IF r10 dspumul r30 r40 → r80`	no change, since guard is false
r20 = 1, r30 = 0x10, r40 = 0x20	`IF r20 dspumul r30 r40 → r100`	r100 ← 0x200
r50 = 0x40000000, r90 = 2	`dspumul r50 r90 → r110`	r110 ← 0x80000000
r80 = 0xffffffff	`dspumul r80 r80 → r120`	r120 ← 0xffffffff
r70 = 0x80000000, r90 = 2	`dspumul r70 r90 → r120`	r120 ← 0xffffffff

dspuquadaddui

Quad clipped add of unsigned/signed bytes

SYNTAX

```
[ IF rguard ] dspuquadaddui rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dspalu
Operation code	78
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

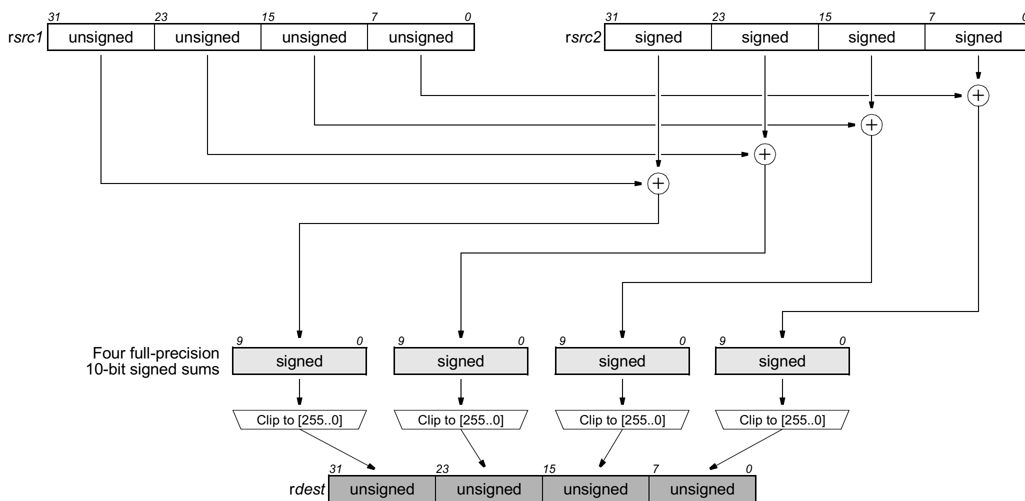
dspidualadd

FUNCTION

```
if rguard then {
    for (i ← 0, m ← 31, n ← 24; i < 4; i ← i + 1, m ← m – 8, n ← n – 8) {
        temp ← zero_ext8to32(rsrc1<m:n>) + sign_ext8to32(rsrc2<m:n>)
        if temp < 0 then
            rdest<m:n> ← 0
        else if temp > 0xff then
            rdest<m:n> ← 0xff
        else rdest<m:n> ← temp<7:0>
    }
}
```

DESCRIPTION

As shown below, the `dspuquadaddui` operation computes four separate sums of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. The bytes in r*src1* are considered unsigned values; the bytes in r*src2* are considered signed. The four sums are clipped into the unsigned range [255..0] (or [0xff..0]); thus, the final byte sums are unsigned. All computations are performed without loss of precision.



The `dspuquadaddui` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x02010001, r40 = 0xffffff01	`dspuquadaddui r30 r40 → r50`	r50 ← 0x01000002
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	`IF r10 dspuquadaddui r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	`IF r20 dspuquadaddui r60 r70 → r90`	r90 ← 0xff38c800

 © TriMedia Technologies, Inc.

Clipped unsigned subtract

dspusub

SYNTAX

```
[ IF rguard ] dspusub rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← zero_ext32to64(rsrc1) – zero_ext32to64(rsrc2)
    if (signed)temp < 0 then
        rdest ← 0
    else
        rdest ← temp<31:0>
}
```

ATTRIBUTES

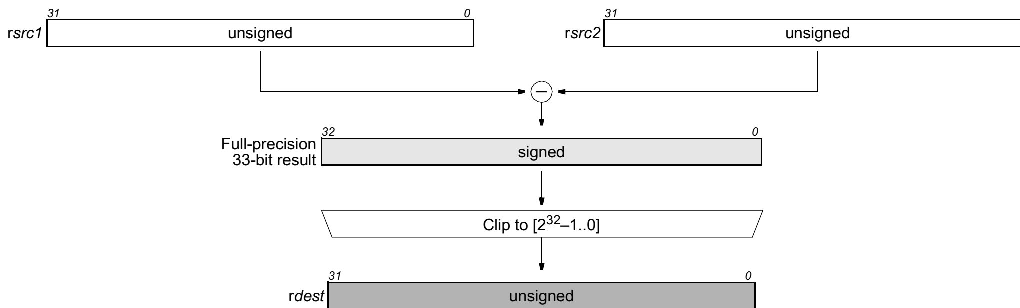
Function unit	dspalu
Operation code	69
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspiabs dspiadd dspimul
dspisub dspuadd dspumul

DESCRIPTION

As shown below, the dspusub operation computes unsigned difference *rsrc1*–*rsrc2*, clips the result into the unsigned range [2^{32}–1..0] (or [0xffffffff..0]), and stores the clipped value into *rdest*.



The dspusub operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x1200, r40 = 0xff	dspusub r30 r40 → r60	r60 ← 0x1101
r10 = 0, r30 = 0x1200, r40 = 0xff	IF r10 dspusub r30 r40 → r80	no change, since guard is false
r20 = 1, r30 = 0x1200, r40 = 0xff	IF r20 dspusub r30 r40 → r100	r100 ← 0x1101
r50 = 0, r90 = 1	dspusub r50 r90 → r110	r110 ← 0
r70 = 0x80000001, r80 = 0xffffffff	dspusub r70 r80 → r120	r120 ← 0

dualasr

Dual-16 arithmetic shift right

SYNTAX

```
[ IF rguard ] dualasr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    n <- rsrc2<3:0>
    rdest<31:31-n> <- rsrc1<31>
    rdest<30-n:16> <- rsrc1<30:16+n>
    rdest<15:15-n> <- rsrc1<15>
    rdest<14-n:0> <- rsrc1<14:n>
    if rsrc2<31:4> != 0 {
        rdest<31:16> <- rsrc1<31>
        rdest<15:0> <- rsrc1<15>
    }
}
```

ATTRIBUTES

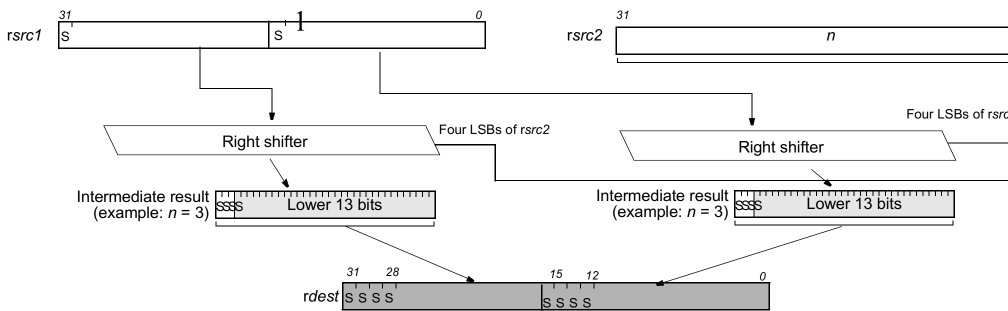
Function unit	shifter
Operation code	102
Number of operands	2
Modifier	No
Modifier range	-
Latency	1
Issue slots	1,2

SEE ALSO

asl asli asri lsl lsli lsr lsri rol roli

DESCRIPTION

The argument rsrc1 contains two 16-bit signed integers, rsrc1<31:16> and rsrc1<15:0>. Rsrc2 specifies an unsigned shift amount, and the two 16-bit integers shifted right by this amount. The sign bits rsrc1<31> and rsrc1<15> are replicated as needed within each 16-bit value from the left. If the rsrc2<31:4> value is not zero, then take this as a shift by 16 or more, i.e. extend the sign bit into either result.



The dualasr operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x70087008, r40 = 0x1	`dualasr r30 r40 -> r50`	r50 <- 0x38043804
r30 = 0x70087008, r40 = 0x2	`dualasr r30 r40 -> r50`	r50 <- 0x1c021c02
r10 = 0, r30 = 0x70087008, r40 = 0x2	`IF r10 dualasr r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x70084008, r40 = 0x4	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0x07000400
r10 = 1, r30 = 0x800c800c, r40 = 0x4	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xf800f800
r10 = 1, r30 = 0x700c700c, r40 = 0xf	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0x00000000
r10 = 1, r30 = 0x700c800c, r40 = 0xf	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0x0000ffff
r10 = 1, r30 = 0x800c700c, r40 = 0xf	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xffff0000
r10 = 1, r30 = 0x800c700c, r40 = 0x10000000	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xffff0000
r10 = 1, r30 = 0x800c700c, r40 = 0x10	`IF r10 dualasr r30 r40 -> r50`	r50 <- 0xffff0000

 © TriMedia Technologies, Inc.

Dual-16 clip signed to signed

dualiclipi

SYNTAX

```
[ IF rguard ] dualiclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
rdest<31:16> <- min(max(rscrc1<31:16>, -rsrc2<15:0>-1), rsrc2<15:0>)
rdest<15:0> <- min(max(rscrc1<15:0>, -rsrc2<15:0>-1), rsrc2<15:0>)
}

ATTRIBUTES

Function unit	dspalu
Operation code	82
Number of operands	2
Modifier	No
Modifier range	-
Latency	2
Issue slots	1,3

SEE ALSO

iclipi uclipi dualuclipi imin imax quadumax quadumin

DESCRIPTION

The argument rsrc1 contains two signed16-bit integers, rsrc1<31:16> and rsrc1<15:0>. Each integer value is clipped into the signed integer range (-rsrc2 -1) to rsrc2. The value in rsrc2 contains an unsigned integer and must have the value between 0 and 0x7fff inclusive.

The dualiclipi operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00800080, r40 = 0x7f	`dualiclipi r30 r40 -> r50`	r50 <- 0x007f007f
r30 = 0x7ffff7ffff, r40 = 0x7ffe	`dualiclipi r30 r40 -> r50`	r50 <- 0x7ffe7ffe
r10 = 0, r30 = 0x7ffff7ffff, r40 = 0x7ffe	`IF r10 dualiclipi r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x12345678, r40 = 0xabc	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0x0abc0abc
r10 = 1, r30 = 0x80008000, r40 = 0x03ff	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0xfc00fc00
r10 = 1, r30 = 0x800003fe, r40 = 0x03ff	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0xfc0003fe
r10 = 1, r30 = 0x000f03fe, r40 = 0x03ff	`IF r10 dualiclipi r30 r40 -> r50`	r50 <- 0x000f03fe

dualuclipi

Dual-16 clip signed to unsigned

SYNTAX

```
[ IF rguard ] dualuclipi rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    rdest<31:16> <- min(max(rscrc1<31:16>, 0), rsrc2<15:0>)
    rdest<15:0> <- min(max(rscrc1<15:0>, 0), rsrc2<15:0>)
}
```

ATTRIBUTES

Attribute	Value
Function unit	dspalu
Operation code	83
Number of operands	2
Modifier	No
Modifier range	-
Latency	2
Issue slots	1,3

SEE ALSO

iclipi uclipi dualiclipi imin imax quadumax quadumin

DESCRIPTION

The argument rsrc1 contains two 16-bit signed integers, rsrc1<31:16> and rsrc1<15:0>. Each integer value is clipped into the unsigned integer range 0 to rsrc2. The value in rsrc2 contains an unsigned integer and must have the value between 0 and 0xffff inclusive.

The dualuclipi operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00800080, r40 = 0x7f	`dualuclipi r30 r40 -> r50`	r50 <- 0x007f007f
r30 = 0x7ffff7ffff, r40 = 0x7ffe	`dualuclipi r30 r40 -> r50`	r50 <- 0x7ffe7ffe
r10 = 0, r30 = 0x7ffff7ffff, r40 = 0x7ffe	`IF r10 dualuclipi r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x12345678, r40 = 0xabc	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x0abc0abc
r10 = 1, r30 = 0x80008000, r40 = 0x03ff	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x00000000
r10 = 1, r30 = 0x800003fe, r40 = 0x03ff	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x000003fe
r10 = 1, r30 = 0x000f03fe, r40 = 0x03ff	`IF r10 dualuclipi r30 r40 -> r50`	r50 <- 0x000f03fe

© TriMedia Technologies, Inc.

Floating-point absolute value

fabsval

SYNTAX

```
[ IF rguard ] fabsval rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if (float)r*src1* < 0 **then**
    r*dest* ← –(float)r*src1*
  else
    r*dest* ← (float)r*src1*
}

ATTRIBUTES

Function unit	falu
Operation code	115
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

iabs dspiabs dspidualabs fabsvalflags readpcsw writepcsw

DESCRIPTION

The `fabsval` operation computes the absolute value of the argument r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. If an argument is denormalized, zero is substituted for the argument before computing the absolute value, and the IFZ flag in the PCSW is set. If `fabsval` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fabsvalflags` operation computes the exception flags that would result from an individual `fabsval`.

The `fabsval` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsval r30 → r90`	r90 ← 0x40400000 (3.0)
r35 = 0xbf800000 (-1.0)	`fabsval r35 → r95`	r95 ← 0x3f800000 (1.0)
r40 = 0x00400000 (5.877471754e-39)	`fabsval r40 → r100`	r100 ← 0x0 (+0.0), IFZ set
r45 = 0xffffffff (QNaN)	`fabsval r45 → r105`	r105 ← 0xffffffff (QNaN)
r50 = 0xffbfffff (SNaN)	`fabsval r50 → r110`	r110 ← 0xffffffff (QNaN), INV set
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsval r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsval r55 → r120`	r120 ← 0x7f7fffff (3.402823466e+38)

fabsvalflags

IEEE status flags from floating-point absolute value

SYNTAX

```
[ IF rguard ] fabsvalflags rsrc1 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← ieee_flags(abs_val((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	116
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

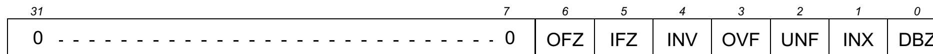
SEE ALSO

`fabsval` `faddflags` `readpcsw`

DESCRIPTION

The `fabsvalflags` operation computes the IEEE exceptions that would result from computing the absolute value of r*src1* and writes a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If r*src1* is denormalized, the IFZ bit in the result is set.

The `fabsvalflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fabsvalflags r30 → r90`	r90 ← 0x0
r35 = 0xbf800000 (-1.0)	`fabsvalflags r35 → r95`	r95 ← 0x0
r40 = 0x00400000 (5.877471754e-39)	`fabsvalflags r40 → r100`	r100 ← 0x20 (IFZ)
r45 = 0xffffffff (QNaN)	`fabsvalflags r45 → r105`	r105 ← 0x0
r50 = 0xffbfffff (SNaN)	`fabsvalflags r50 → r110`	r110 ← 0x10 (INV)
r10 = 0, r55 = 0xff7fffff (–3.402823466e+38)	`IF r10 fabsvalflags r55 → r115`	no change, since guard is false
r20 = 1, r55 = 0xff7fffff (–3.402823466e+38)	`IF r20 fabsvalflags r55 → r120`	r120 ← 0x0

© TriMedia Technologies, Inc.

Floating-point add

fadd

SYNTAX

```
[ IF rguard ] fadd rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* + (float)r*src2*

ATTRIBUTES

Function unit	falu
Operation code	22
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

faddflags iadd dspiadd dspidualadd readpcsw writepcsw

DESCRIPTION

The fadd operation computes the sum r*src1*+r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the sum, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If fadd causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The faddflags operation computes the exception flags that would result from an individual fadd.

The fadd operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	fadd r60 r30 → r90	r90 ← 0xc0000000 (–2.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	fadd r40 r60 → r95	r95 ← 0x00000000 (0.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	IF r10 fadd r40 r80 → r100	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	IF r20 fadd r40 r80 → r110	r110 ← 0x40400000 (3.0), INX flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	fadd r40 r81 → r111	r111 ← 0x40400000 (3.0), IFZ flag set
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x80800000 (–1.175494351e-38)	fadd r82 r83 → r112	r112 ← 0x00000000 (0.0), OFZ, UNF, INX flags set
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	fadd r84 r85 → r113	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38)	fadd r70 r70 → r120	r120 ← 0x7f800000 (+INF), OVF, INX flags set
r80 = 0x00800000 (1.763241526e–38)	fadd r80 r80 → r125	r125 ← 0x01000000 (2.350988702e–38)

faddflags IEEE status flags from floating-point add

SYNTAX

```
[ IF rguard ] faddflags rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* + (float)r*src2*)

ATTRIBUTES

Function unit	falu
Operation code	112
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

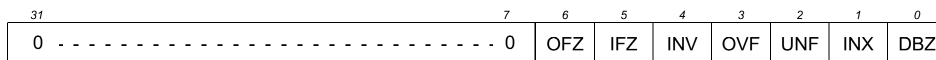
SEE ALSO

fadd fsubflags readpcsw

DESCRIPTION

The faddflags operation computes the IEEE exceptions that would result from computing the sum r*src1*+r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the sum, and the IFZ bit in the result is set. If the sum would be denormalized, the OFZ bit in the result is set.

The faddflags operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r10 = 0x7f7fffff (3.402823466e+38), r20 = 0x3f800000 (1.0)	`faddflags r10 r20 → r60`	r60 ← 0x2 (INX)
r30 = 0, r10 = 0x7f7fffff (3.402823466e+38)	`IF r30 faddflags r10 r10 → r50`	no change, since guard is false
r40 = 1, r10 = 0x7f7fffff (3.402823466e+38)	`IF r40 faddflags r10 r10 → r70`	r70 ← 0xa (OVF INX)
r80 = 0x00a00000 (1.469367939e–38), r81 = 0x80800000 (–1.17549435e–38)	`faddflags r80 r81 → r100`	r100 ← 0x46 (OFZ UNF INX)
r95 = 0x7f800000 (+INF), r96 = 0xff800000 (–INF)	`faddflags r95 r96 → r105`	r105 ← 0x10 (INV)
r98 = 0x40400000 (3.0), r99 = 0x00400000 (5.877471754e–39)	`faddflags r98 r99 → r111`	r111 ← 0x20 (IFZ)

© TriMedia Technologies, Inc.

Floating-point divide

fdiv

SYNTAX

`[ IF rguard ] fdiv rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* / (float)r*src2*

ATTRIBUTES

Function unit	ftough
Operation code	108
Number of operands	2
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

`fdivflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fdiv` operation computes the quotient r*src1*÷r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the quotient, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fdiv` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fdivflags` operation computes the exception flags that would result from an individual `fdiv`.

The `fdiv` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fdiv r60 r30 → r90`	r90 ← 0xc0400000 (–3.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fdiv r40 r60 → r95`	r95 ← 0xbf800000 (–1.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fdiv r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fdiv r40 r80 → r110`	r110 ← 0x7f400000 (2.552117754e38)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fdiv r40 r81 → r111`	r111 ← 0x7f800000 (+INF), IFZ, DBZ flags set
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x80800000 (–1.175494351e–38)	`fdiv r82 r83 → r112`	r112 ← 0xbfc00000 (-1.5)
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fdiv r84 r85 → r113`	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38)	`fdiv r70 r70 → r120`	r120 ← 0x3f800000 (1.0)
r80 = 0x00800000 (1.763241526e–38)	`fdiv r80 r80 → r125`	r125 ← 0x3f800000 (1.0)
r75 = 0x40400000 (3.0), r76 = 0x0 (0.0)	`fdiv r75 r76 → r126`	r126 ← 0x7f800000 (+INF), DBZ flag set

fdivflags IEEE status flags from floating-point divide

SYNTAX

```
[ IF rguard ] fdivflags rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	ftough
Operation code	109
Number of operands	2
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

fdiv faddflags readpcsw

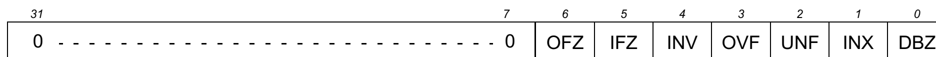
FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* / (float)r*src2*)

DESCRIPTION

The `fdivflags` operation computes the IEEE exceptions that would result from computing the quotient r*src1*÷r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the quotient, and the IFZ bit in the result is set. If the quotient would be denormalized, the OFZ bit in the result is set.

The `fdivflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x7f7fffff (3.402823466e+38), r40 = 0x3f800000 (1.0)	`fdivflags r30 r40 → r100`	r100 ← 0
r10 = 0, r50 = 0x7f7fffff (3.402823466e+38) r60 = 0x3e000000 (0.125)	`IF r10 fdivflags r50 r60 → r110`	no change, since guard is false
r20 = 1, r50 = 0x7f7fffff (3.402823466e+38) r60 = 0x3e000000 (0.125)	`IF r20 fdivflags r50 r60 → r111`	r111 ← 0xa (OVF INX)
r70 = 0x40400000 (3.0), r80 = 0x00400000 (5.877471754e–39)	`fdivflags r70 r80 → r112`	r112 ← 0x21 (IFZ DBZ)
r85 = 0x7f800000 (+INF), r86 = 0xff800000 (–INF)	`fdivflags r85 r86 → r113`	r113 ← 0x10 (INV)

© TriMedia Technologies, Inc.

Floating-point compare equal

feql

SYNTAX

```
[ IF rguard ] feql rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (float)r*src1* = (float)r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	fcomp
Operation code	148
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ieql feqlflags fneq
readpcsw writepcsw

DESCRIPTION

The feql operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If feql causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The feqlflags operation computes the exception flags that would result from an individual feql.

The feql operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	feql r30 r40 → r80	r80 ← 0
r30 = 0x40400000 (3.0)	feql r30 r30 → r90	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r10 feql r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r20 feql r60 r30 → r110	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	feql r30 r60 → r120	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	feql r30 r61 → r121	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	feql r50 r55 → r125	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	feql r60 r65 → r126	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	feql r50 r50 → r127	r127 ← 1

feqlflags

IEEE status flags from floating-point compare equal

SYNTAX

```
[ IF rguard ] feqlflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
 r*dest* ← ieee_flags((float)r*src1* = (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	149
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

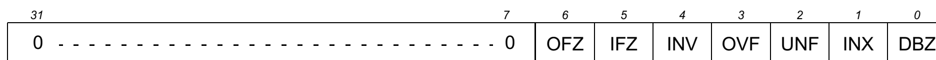
SEE ALSO

feql ieql fgtrflags readpcsw

DESCRIPTION

The feqlflags operation computes the IEEE exceptions that would result from computing the comparison r*src1*=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The feqlflags operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	feqlflags r30 r40 → r80	r80 ← 0
r30 = 0x40400000 (3.0)	feqlflags r30 r30 → r90	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r10 feqlflags r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r20 feqlflags r60 r30 → r110	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	feqlflags r30 r60 → r120	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	feqlflags r30 r61 → r121	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	feqlflags r50 r55 → r125	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	feqlflags r60 r65 → r126	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	feqlflags r50 r50 → r127	r127 ← 0

© TriMedia Technologies, Inc.

Floating-point compare greater or equal

fgeq

SYNTAX

```
[ IF rguard ] fgeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if (float)rsrc1 >= (float)rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	146
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

igeq fgeqflags fgtr readpcsw writepcsw

DESCRIPTION

The `fgeq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fgeq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fgeqflags` operation computes the exception flags that would result from an individual `fgeq`.

The `fgeq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgeq r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fgeq r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgeq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgeq r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgeq r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgeq r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgeq r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgeq r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fgeq r50 r50 → r127`	r127 ← 1

fgeqflags

IEEE status flags from floating-point compare greater or equal

SYNTAX

```
[ IF rguard ] fgeqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
r*dest* ← ieee_flags((float)r*src1* >= (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	147
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

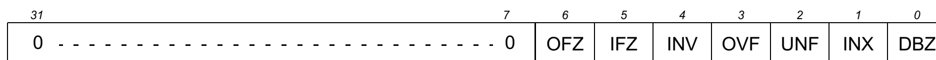
SEE ALSO

fgeq igeq fgtrflags readpcsw

DESCRIPTION

The `fgeqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*>=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fgeqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgeqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fgeqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgeqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgeqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgeqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgeqflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgeqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgeqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fgeqflags r50 r50 → r127`	r127 ← 0

 © TriMedia Technologies, Inc.

Floating-point compare greater

fgtr

SYNTAX

```
[ IF rguard ] fgtr rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if (float)rsrc1 > (float)rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	144
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

igtr fgtrflags fgeq readpcsw writepcsw

DESCRIPTION

The fgtr operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If fgtr causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The fgtrflags operation computes the exception flags that would result from an individual fgtr.

The fgtr operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	fgtr r30 r40 → r80	r80 ← 1
r30 = 0x40400000 (3.0)	fgtr r30 r30 → r90	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r10 fgtr r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r20 fgtr r60 r30 → r110	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	fgtr r30 r60 → r120	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	fgtr r30 r61 → r121	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	fgtr r50 r55 → r125	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	fgtr r60 r65 → r126	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	fgtr r50 r50 → r127	r127 ← 0

fgtrflags

IEEE status flags from floating-point compare greater

SYNTAX

`[ IF rguard ] fgtrflags rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float)r*src1* > (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	145
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

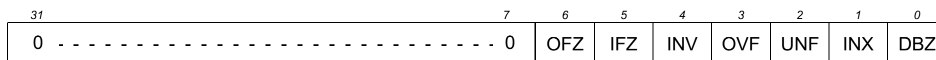
SEE ALSO

fgtr igtr fgeqflags readpcsw

DESCRIPTION

The `fgtrflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*>r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fgtrflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fgtrflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fgtrflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fgtrflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fgtrflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fgtrflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fgtrflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fgtrflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fgtrflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fgtrflags r50 r50 → r127`	r127 ← 0

© TriMedia Technologies, Inc.

Floating-point compare less-than or equal

pseudo-op for fgeq

fleq

SYNTAX

```
[ IF rguard ] fleq rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (float)r*src1* <= (float)r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	fcomp
Operation code	146
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

`ileq` `fgeq` `fleqflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fleq` operation is a pseudo operation transformed by the scheduler into an `fgeq` with the arguments exchanged (`fleq`'s r*src1* is `fgeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fleq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fleq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fleqflags` operation computes the exception flags that would result from an individual `fleq`.

The `fleq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fleq r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fleq r30 r30 → r90`	r90 ← 1
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fleq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fleq r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fleq r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fleq r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fleq r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fleq r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`fleq r50 r50 → r127`	r127 ← 1

fleqflags

IEEE status flags from floating-point compare less-than or equal

pseudo-op for fgeqflags

SYNTAX

```
[ IF rguard ] fleqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
r*dest* ← ieee_flags((float)r*src1* <= (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	147
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

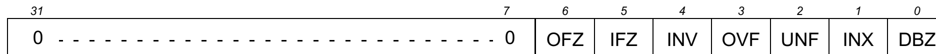
fleq ileq fgeqflags readpcsw

DESCRIPTION

The `fleqflags` operation is a pseudo operation transformed by the scheduler into an `fgeqflags` with the arguments exchanged (`fleqflags`'s r*src1* is `fgeqflags`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fleqflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*<=r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `fleqflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fleqflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fleqflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fleqflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fleqflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fleqflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fleqflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fleqflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fleqflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`fleqflags r50 r50 → r127`	r127 ← 0

© TriMedia Technologies, Inc.

Floating-point compare less-than

fles

pseudo-op for fgtr

SYNTAX

```
[ IF rguard ] fles rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if (float)rsrc1 < (float)rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	144
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

iles fgtr flesflags readpcsw writepcsw

DESCRIPTION

The `fles` operation is a pseudo operation transformed by the scheduler into an `fgtr` with the arguments exchanged (`fles`'s r*src1* is `fgtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `fles` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fles` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `flesflags` operation computes the exception flags that would result from an individual `fles`.

The `fles` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fles r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`fles r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fles r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fles r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fles r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fles r30 r61 → r121`	r121 ← 0, INV flag set
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fles r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fles r60 r65 → r126`	r126 ← 0, IFZ flag set
r50 = 0x7f800000 (+INF)	`fles r50 r50 → r127`	r127 ← 0

flesflags

IEEE status flags from floating-point compare less-than

pseudo-op for fgtrflags

SYNTAX

```
[ IF rguard ] flesflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
r*dest* ← ieee_flags((float)r*src1* < (float)r*src2*)

ATTRIBUTES

Function unit	fcomp
Operation code	145
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

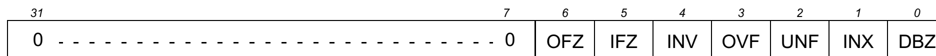
`fles` `iles` `fleqflags` `readpcsw`

DESCRIPTION

The `flesflags` operation is a pseudo operation transformed by the scheduler into an `fgtrflags` with the arguments exchanged (`flesflags`'s r*src1* is `fgtrflags`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `flesflags` operation computes the IEEE exceptions that would result from computing the comparison r*src1*<r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The `flesflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`flesflags r30 r40 → r80`	r80 ← 0
r30 = 0x40400000 (3.0)	`flesflags r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 flesflags r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 flesflags r60 r30 → r110`	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`flesflags r30 r60 → r120`	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`flesflags r30 r61 → r121`	r121 ← 0x10 (INV)
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`flesflags r50 r55 → r125`	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`flesflags r60 r65 → r126`	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	`flesflags r50 r50 → r127`	r127 ← 0

© TriMedia Technologies, Inc.

Floating-point multiply

fmul

SYNTAX

```
[ IF rguard ] fmul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← (float)r*src1* × (float)r*src2*

ATTRIBUTES

Function unit	ifmul
Operation code	28
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

`imul` `umul` `dspimul` `dspidualmul` `fmulflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fmul` operation computes the product r*src1*×r*src2* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the product, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fmul` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fmulflags` operation computes the exception flags that would result from an individual `fmul`.

The `fmul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fmul r60 r30 → r90`	r90 ← 0xc0400000 (-3.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fmul r40 r60 → r95`	r95 ← 0xc1100000 (-9.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fmul r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fmul r40 r80 → r105`	r105 ← 0x1400000 (3.52648305e-38)
r41 = 0x3f000000 (0.5), r80 = 0x00800000 (1.17549435e–38)	`fmul r41 r80 → r110`	r110 ← 0x0, OFZ, UNF, INX flags set
r42 = 0x7f800000 (+INF), r43 = 0x0 (0.0)	`fmul r42 r43 → r106`	r106 ← 0xffffffff (QNaN), INV flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fmul r40 r81 → r111`	r111 ← 0, IFZ flag set
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x8080000 (–1.175494351e–38)	`fmul r82 r83 → r112`	r112 ← 0, UNF, INX flag set
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fmul r84 r85 → r113`	r113 ← 0xff800000 (-INF)
r70 = 0x7f7fffff (3.402823466e+38)	`fmul r70 r70 → r120`	r120 ← 0x7f800000, OVF, INX flags set
r80 = 0x00800000 (1.763241526e–38)	`fmul r80 r80 → r125`	r125 ← 0, UNF, INX flag set

fmulflags

IEEE status flags from floating-point multiply

SYNTAX

```
[ IF rguard ] fmulflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← ieee_flags((float)*rsrc1* × (float)*rsrc2*)

ATTRIBUTES

Function unit	ifmul
Operation code	143
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

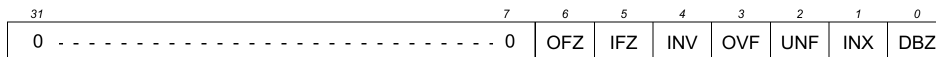
SEE ALSO

`fmul faddflags readpcsw`

DESCRIPTION

The `fmulflags` operation computes the IEEE exceptions that would result from computing the product r*src1*×r*src2* and stores a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the product, and the IFZ bit in the result is set. If the product would be denormalized, the OFZ bit in the result is set.

The `fmulflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fmulflags r60 r30 → r90`	r90 ← 0
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fmulflags r40 r60 → r95`	r95 ← 0
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r10 fmulflags r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e–38)	`IF r20 fmulflags r40 r80 → r105`	r105 ← 0
r41 = 0x3f000000 (0.5), r80 = 0x00800000 (1.17549435e–38)	`fmulflags r41 r80 → r110`	r110 ← 0x46 (OFZ UNF INX)
r42 = 0x7f800000 (+INF), r43 = 0x0 (0.0)	`fmulflags r42 r43 → r106`	r106 ← 0x10 (INV)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fmulflags r40 r81 → r111`	r111 ← 0x20 (IFZ)
r82 = 0x00c00000 (1.763241526e–38), r83 = 0x8080000 (–1.175494351e–38)	`fmulflags r82 r83 → r112`	r112 ← 0x06 (UNF INX)
r84 = 0x7f800000 (+INF), r85 = 0xff800000 (–INF)	`fmulflags r84 r85 → r113`	r113 ← 0
r70 = 0x7f7fffff (3.402823466e+38)	`fmulflags r70 r70 → r120`	r120 ← 0x0a (OVF INX)
r80 = 0x00800000 (1.763241526e–38)	`fmulflags r80 r80 → r125`	r125 ← 0x06 (UNF INX)

 © TriMedia Technologies, Inc.

Floating-point compare not equal

fneq

SYNTAX

```
[ IF rguard ] fneq rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (float)r*src1* != (float)r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	fcomp
Operation code	150
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

ineq feql fneqflags readpcsw writepcsw

DESCRIPTION

The `fneq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as IEEE single-precision floating-point values; the result is an integer. If an argument is denormalized, zero is substituted for the argument before computing the comparison, and the IFZ flag in the PCSW is set. If `fneq` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fneqflags` operation computes the exception flags that would result from an individual `fneq`.

The `fneq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	`fneq r30 r40 → r80`	r80 ← 1
r30 = 0x40400000 (3.0)	`fneq r30 r30 → r90`	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r10 fneq r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	`IF r20 fneq r60 r30 → r110`	r110 ← 1
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	`fneq r30 r60 → r120`	r120 ← 1
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	`fneq r30 r61 → r121`	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	`fneq r50 r55 → r125`	r125 ← 1
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	`fneq r60 r65 → r126`	r126 ← 1, IFZ flag set
r50 = 0x7f800000 (+INF)	`fneq r50 r50 → r127`	r127 ← 0

fneqflags

IEEE status flags from floating-point compare not equal

SYNTAX

```
[ IF rguard ] fneqflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
    rdest ← ieee_flags((float)*rsrc1* != (float)*rsrc2*)

ATTRIBUTES

Function unit	fcomp
Operation code	151
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

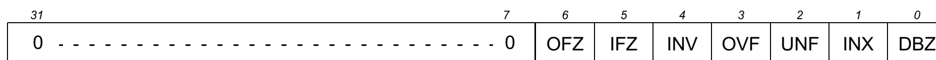
SEE ALSO

fneq ineq fleqflags readpcsw

DESCRIPTION

The fneqflags operation computes the IEEE exceptions that would result from computing the comparison *rsrc1*!=*rsrc2* and stores a bit vector representing the exception flags into *rdest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in *rdest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If an argument is denormalized, zero is substituted before computing the comparison, and the IFZ bit in the result is set.

The fneqflags operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0), r40 = 0 (0.0)	fneqflags r30 r40 → r80	r80 ← 0
r30 = 0x40400000 (3.0)	fneqflags r30 r30 → r90	r90 ← 0
r10 = 0, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r10 fneqflags r60 r30 → r100	no change, since guard is false
r20 = 1, r60 = 0x3f800000 (1.0), r30 = 0x40400000 (3.0)	IF r20 fneqflags r60 r30 → r110	r110 ← 0
r30 = 0x40400000 (3.0), r60 = 0x3f800000 (1.0)	fneqflags r30 r60 → r120	r120 ← 0
r30 = 0x40400000 (3.0), r61 = 0xffffffff (QNaN)	fneqflags r30 r61 → r121	r121 ← 0
r50 = 0x7f800000 (+INF) r55 = 0xff800000 (-INF)	fneqflags r50 r55 → r125	r125 ← 0
r60 = 0x3f800000 (1.0), r65 = 0x00400000 (5.877471754e-39)	fneqflags r60 r65 → r126	r126 ← 0x20 (IFZ)
r50 = 0x7f800000 (+INF)	fneqflags r50 r50 → r127	r127 ← 0

© TriMedia Technologies, Inc.

Sign of floating-point value

fsign

SYNTAX

```
[ IF rguard ] fsign rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (float)rsrc1 = 0.0 then
        rdest ← 0
    else if (float)rsrc1 < 0.0 then
        rdest ← 0xffffffff
    else
        rdest ← 1
}
```

ATTRIBUTES

Function unit	fcomp
Operation code	152
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

fsignflags readpcsw writepcsw

DESCRIPTION

The `fsign` operation sets the destination register, r*dest*, to either 0, 1, or –1 depending on the sign of the argument in r*src1*. rdest is set to 0 if r*src1* is equal to zero, to 1 if r*src1* is positive, or to –1 if r*src1* is negative. The argument is treated as an IEEE single-precision floating-point value; the result is an integer. If the argument is denormalized, zero is substituted before computing the comparison, and the IFZ flag in the PCSW is set; thus, the result of `fsign` for a denormalized argument is 0. If `fsign` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsignflags` operation computes the exception flags that would result from an individual `fsign`.

The `fsign` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fsign r30 → r100`	r100 ← 1
r40 = 0xbf800000 (-1.0)	`fsign r40 → r105`	r105 ← 0xffffffff (-1)
r50 = 0x80800000 (-1.175494351e-38)	`fsign r50 → r110`	r110 ← 0xffffffff (-1)
r60 = 0x80400000 (-5.877471754e-39)	`fsign r60 → r115`	r115 ← 0, IFZ flag set
r10 = 0, r70 = 0xffffffff (QNaN)	`IF r10 fsign r70 → r116`	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	`IF r20 fsign r70 → r117`	r117 ← 0, INV flag set
r80 = 0xff800000 (-INF)	`fsign r80 → r120`	r120 ← 0xffffffff (-1)

fsignflags

IEEE status flags from floating-point sign

SYNTAX

```
[ IF rguard ] fsignflags rsrc1 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← ieee_flags(sign((float)*rsrc1*))

ATTRIBUTES

Function unit	fcomp
Operation code	153
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

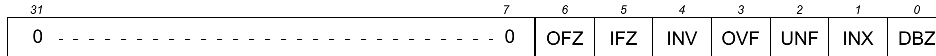
SEE ALSO

fsign readpcsw

DESCRIPTION

The `fsignflags` operation computes the IEEE exceptions that would result from computing the sign of *rsrc1* and stores a bit vector representing the exception flags into *rdest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in *rdest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. If the argument is denormalized, zero is substituted before computing the sign, and the IFZ bit in the result is set.

The `fsignflags` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`fsignflags r30 → r100`	r100 ← 0
r40 = 0xbf800000 (-1.0)	`fsignflags r40 → r105`	r105 ← 0
r50 = 0x80800000 (-1.175494351e-38)	`fsignflags r50 → r110`	r110 ← 0
r60 = 0x80400000 (-5.877471754e-39)	`fsignflags r60 → r115`	r115 ← 0x20 (IFZ)
r10 = 0, r70 = 0xffffffff (QNaN)	`IF r10 fsignflags r70 → r116`	no change, since guard is false
r20 = 1, r70 = 0xffffffff (QNaN)	`IF r20 fsignflags r70 → r117`	r117 ← 0x10 (INV)
r80 = 0xff800000 (-INF)	`fsignflags r80 → r120`	r120 ← 0

 © TriMedia Technologies, Inc.

Floating-point square root

fsqrt

SYNTAX

```
[ IF rguard ] fsqrt rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← square_root(r*src1*)

ATTRIBUTES

Attribute	Value
Function unit	ftough
Operation code	110
Number of operands	1
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

SEE ALSO

`fsqrtflags` `readpcsw` `writepcsw`

DESCRIPTION

The `fsqrt` operation computes the squareroot of r*src1* and stores the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the squareroot, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If `fsqrt` causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `fsqrtflags` operation computes the exception flags that would result from an individual `fsqrt`.

The `fsqrt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0)	`fsqrt r60 → r90`	r90 ← 0xffffffff (QNaN), INV flag set
r40 = 0x40400000 (3.0)	`fsqrt r40 → r95`	r95 ← 0x3fddb3d7 (1.732051), INX flag set
r10 = 0, r40 = 0x40400000 (3.0)	`IF r10 fsqrt r40 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0)	`IF r20 fsqrt r40 → r110`	r110 ← 0x3fddb3d7 (1.732051), INX flag set
r82 = 0x00c00000 (1.763241526e–38)	`fsqrt r82 → r112`	r112 ← 0x201cc471 (1.32787105e-19), INX flag set
r84 = 0x7f800000 (+INF)	`fsqrt r84 → r113`	r113 ← 0x7f800000 (+INF)
r70 = 0x7f7fffff (3.402823466e+38)	`fsqrt r70 → r120`	r120 ← 0x5f7fffff (1.8446743e19), INX flag set
r80 = 0x00400000 (5.877471754e-39)	`fsqrt r80 → r125`	r125 ← 0, IFZ flag set

fsqrtflags

IEEE status flags from floating-point square root

SYNTAX

```
[ IF rguard ] fsqrtflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags(square_root((float)r*src1*))

ATTRIBUTES

Function unit	ftough
Operation code	111
Number of operands	1
Modifier	No
Modifier range	—
Latency	17
Recovery	16
Issue slots	2

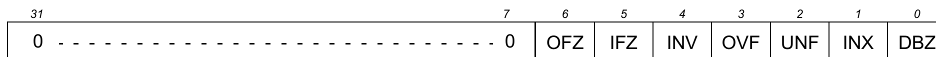
SEE ALSO

fsqrt readpcsw

DESCRIPTION

The `fsqrtflags` operation computes the IEEE exceptions that would result from computing the squareroot of r*src1* and stores a bit vector representing the exception flags into r*dest*. The argument value is in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If the argument is denormalized, zero is substituted before computing the squareroot, and the IFZ bit in the result is set. If the result is denormalized, and the OFZ flag in the PCSW is set.

The `fsqrtflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0)	`fsqrtflags r60 → r90`	r90 ← 0x10 (INV)
r40 = 0x40400000 (3.0)	`fsqrtflags r40 → r95`	r95 ← 0x2 (INX)
r10 = 0, r40 = 0x40400000 (3.0)	`IF r10 fsqrtflags r40 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0)	`IF r20 fsqrtflags r40 → r110`	r110 ← 0x2 (INX)
r82 = 0x00c00000 (1.763241526e–38)	`fsqrtflags r82 → r112`	r112 ← 0x2 (INX)
r84 = 0x7f800000 (+INF)	`fsqrtflags r84 → r113`	r113 ← 0
r70 = 0x7f7fffff (3.402823466e+38)	`fsqrtflags r70 → r120`	r120 ← 0x2 (INX)
r80 = 0x00400000 (5.877471754e-39)	`fsqrtflags r80 → r125`	r125 ← 0x20 (IFZ)

 © TriMedia Technologies, Inc.

Floating-point subtract

fsub

SYNTAX

```
[ IF rguard ] fsub rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest* ← (float)r*src1* – (float)r*src2*

ATTRIBUTES

Function unit	falu
Operation code	113
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

fsubflags isub dspisub dspidualsub readpcsw writepcsw

DESCRIPTION

The fsub operation computes the difference r*src1*–r*src2* and writes the result into r*dest*. All values are in IEEE single-precision floating-point format. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted for the argument before computing the difference, and the IFZ flag in the PCSW is set. If the result is denormalized, the result is set to zero instead, and the OFZ flag in the PCSW is set. If fsub causes an IEEE exception, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The fsubflags operation computes the exception flags that would result from an individual fsub.

The fsub operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	fsub r60 r30 → r90	r90 ← 0xc0800000 (-4.0)
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	fsub r40 r60 → r95	r95 ← 0x40c00000 (6.0)
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	IF r10 fsub r40 r80 → r100	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	IF r20 fsub r40 r80 → r110	r110 ← 0x40400000 (3.0), INX flag set
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	fsub r40 r81 → r111	r111 ← 0x40400000 (3.0), IFZ flag set
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x0080000 (1.175494351e-38)	fsub r82 r83 → r112	r112 ← 0x0, OFZ flag set
r84 = 0x7f800000 (+INF), r85 = 0x7f800000 (+INF)	fsub r84 r85 → r113	r113 ← 0xffffffff (QNaN), INV flag set
r70 = 0x7f7fffff (3.402823466e+38) r86 = 0xff7fffff (-3.402823466e+38)	fsub r70 r86 → r120	r120 ← 0x7f800000 (+INF), OVF, INX flag set
r87 = 0xffffffff (QNaN)) r30 = 0x3f800000 (1.0	fsub r87 r30 → r125	r125 ← 0xffffffff (QNaN)
r87 = 0xffbfffff (SNaN)) r30 = 0x3f800000 (1.0	fsub r87 r30 → r125	r125 ← 0xffffffff (QNaN), INV flag set
r83 = 0x0080001 (1.175494421e-38), r89 = 0x0080000 (1.175494351e-38)	fsub r83 r89 → r126	r126 ← 0x0, UNF flag set

fsubflags

IEEE status flags from floating-point subtract

SYNTAX

```
[ IF rguard ] fsubflags rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← ieee_flags((float)*rsrc1* – (float)*rsrc2*)

ATTRIBUTES

Function unit	falu
Operation code	114
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

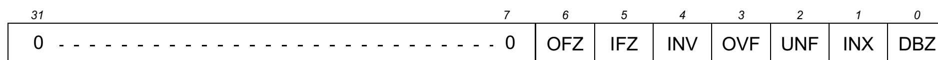
SEE ALSO

fsub faddflags readpcsw

DESCRIPTION

The `fsubflags` operation computes the IEEE exceptions that would result from computing the difference r*src1*–r*src2* and writes a bit vector representing the exception flags into r*dest*. The argument values are in IEEE single-precision floating-point format; the result is an integer bit vector. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the difference, and the IFZ bit in the result is set. If the difference would be denormalized, the OFZ bit in the result is set.

The `fsubflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r60 = 0xc0400000 (–3.0), r30 = 0x3f800000 (1.0)	`fsubflags r60 r30 → r90`	r90 ← 0
r40 = 0x40400000 (3.0), r60 = 0xc0400000 (–3.0)	`fsubflags r40 r60 → r95`	r95 ← 0
r10 = 0, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r10 fsubflags r40 r80 → r100`	no change, since guard is false
r20 = 1, r40 = 0x40400000 (3.0), r80 = 0x00800000 (1.17549435e-38)	`IF r20 fsubflags r40 r80 → r110`	r110 ← 0x2 (INX)
r40 = 0x40400000 (3.0), r81 = 0x00400000 (5.877471754e–39)	`fsubflags r40 r81 → r111`	r111 ← 0x20 (IFZ)
r82 = 0x00c00000 (1.763241526e-38), r83 = 0x0080000 (1.175494351e-38)	`fsubflags r82 r83 → r112`	r112 ← 0x40 (OFZ)
r84 = 0x7f800000 (+INF), r85 = 0x7f800000 (+INF)	`fsubflags r84 r85 → r113`	r113 ← 0x10 (INV)
r70 = 0x7f7fffff (3.402823466e+38) r86 = 0xff7fffff (-3.402823466e+38)	`fsubflags r70 r86 → r120`	r120 ← 0xA (OVF,INX)
r87 = 0xffffffff (QNaN)) r30 = 0x3f800000 (1.0	`fsubflags r87 r30 → r125`	r125 ← 0x0
r87 = 0xffbfffff (SNaN)) r30 = 0x3f800000 (1.0	`fsubflags r87 r30 → r125`	r125 ← 0x10 (INV)
r83 = 0x0080001 (1.175494421e-38), r89 = 0x0080000 (1.175494351e-38)	`fsubflags r83 r89 → r126`	r126 ← 0x4 (UNF)

 © TriMedia Technologies, Inc.

Funnel-shift 1byte

funshift1

SYNTAX

```
[ IF rguard ] funshift1 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest*<31:8> ← r*src1*<23:0>
r*dest*<7:0> ← r*src2*<31:24>

ATTRIBUTES

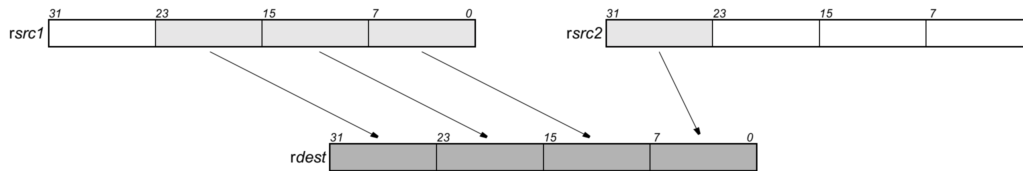
Function unit	shifter
Operation code	99
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift2 funshift3 rol

DESCRIPTION

As shown below, the `funshift1` operation effectively shifts left by one byte the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift1` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift1 r30 r40 → r50`	r50 ← 0xbbccdd11
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift1 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift1 r40 r30 → r70`	r70 ← 0x223344aa

funshift2

Funnel-shift 2 bytes

SYNTAX

```
[ IF rguard ] funshift2 rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest<31:16> ← *rsrc1*<15:0>
rdest<15:0> ← *rsrc2*<31:16>

ATTRIBUTES

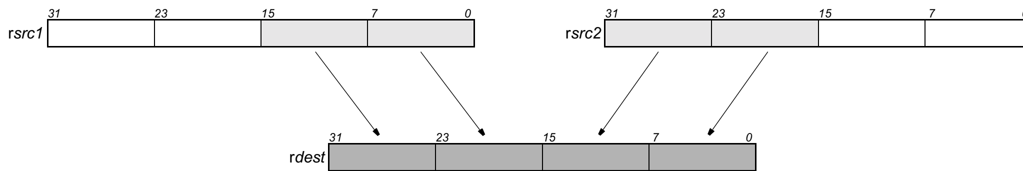
Function unit	shifter
Operation code	100
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift1 funshift3 rol

DESCRIPTION

As shown below, the `funshift2` operation effectively shifts left by two bytes the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift2` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift2 r30 r40 → r50`	r50 ← 0xccdd1122
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift2 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift2 r40 r30 → r70`	r70 ← 0x3344aabb

© TriMedia Technologies, Inc.

Funnel-shift 3 bytes

funshift3

SYNTAX

```
[ IF rguard ] funshift3 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
    r*dest*<31:24> ← r*src1*<7:0>
    r*dest*<23:0> ← r*src2*<31:8>

ATTRIBUTES

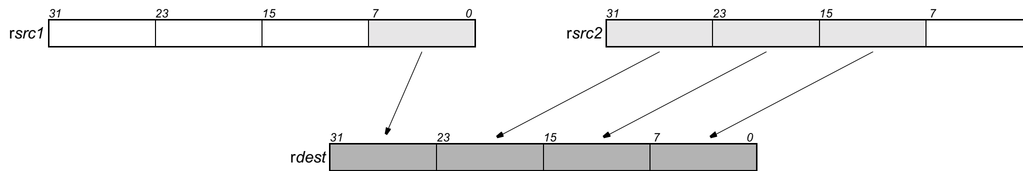
Function unit	shifter
Operation code	101
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

`funshift1` `funshift2` `rol`

DESCRIPTION

As shown below, the `funshift3` operation effectively shifts left by three bytes the 64-bit concatenation of r*src1* and r*src2* and writes the most-significant 32 bits of the shifted result to r*dest*.



The `funshift3` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift3 r30 r40 → r50`	r50 ← 0xdd112233
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift3 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift3 r40 r30 → r70`	r70 ← 0x44aabbcc

h_dspiabs

Clipped signed absolute value

SYNTAX

```
[ IF rguard ] h_dspiabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc2 >= 0 then
        rdest ← rsrc2
    else if rsrc2 = 0x80000000 then
        rdest ← 0x7fffffff
    else
        rdest ← –rsrc2
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	65
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

h_dspiabs dspidualabs dspiadd dspimul dspisub dspuadd dspumul dspusub

DESCRIPTION

The `h_dspiabs` operation computes the absolute value of rsrc2, clips the result into the range [0x0..0x7fffffff], and stores the clipped value into r*dest*. All values are signed integers. This operation requires a zero as first argument. The programmer is advised to use the unary pseudo operation `dspiabs` instead.

The `h_dspiabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`h_dspiabs r0 r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0x80000001	`IF r10 h_dspiabs r0 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0x80000001	`IF r20 h_dspiabs r0 r40 → r100`	r100 ← 0x7fffffff
r50 = 0x80000000	`h_dspiabs r0 r50 → r80`	r80 ← 0x7fffffff
r90 = 0x7fffffff	`h_dspiabs r0 r90 → r110`	r110 ← 0x7fffffff

 © TriMedia Technologies, Inc.

Dual clipped absolute value of signed 16-bit halfwords

h_dspidualabs

SYNTAX

```
[ IF rguard ] h_dspidualabs r0 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp1 ← sign_ext16to32(rsrc2<15:0>)
    temp2 ← sign_ext16to32(rsrc2<31:16>)
    if temp1 = 0xffff8000 then temp1 ← 0x7fff
    if temp2 = 0xffff8000 then temp2 ← 0x7fff
    if temp1 < 0 then temp1 ← –temp1
    if temp2 < 0 then temp2 ← –temp2
    rdest<31:16> ← temp2<15:0>
    rdest<15:0> ← temp1<15:0>
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	72
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

dspidualabs dspiabs
dspidualadd dspidualmul
dspidualsub dspiabs

DESCRIPTION

The h_dspidualabs operation performs two 16-bit clipped, signed absolute value computations separately on the high and low 16-bit halfwords of rsrc2. Both absolute values are clipped into the range [0x0..0x7fff] and written into the corresponding halfwords of r*dest*. All values are signed 16-bit integers. This operation requires a zero as first argument. The programmer is advised to use the dspidualabs pseudo operation instead.

The h_dspidualabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0032	h_dspidualabs r0 r30 → r60	r60 ← 0x00010032
r10 = 0, r40 = 0x80008001	IF r10 h_dspidualabs r0 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0x80008001	IF r20 h_dspidualabs r0 r40 → r100	r100 ← 0x7fff7fff
r50 = 0x0032ffff	h_dspidualabs r0 r50 → r80	r80 ← 0x00320001
r90 = 0x7fffffff	h_dspidualabs r0 r90 → r110	r110 ← 0x7fff0001

h_iabs

Hardware absolute value

SYNTAX

```
[ IF rguard ] h_iabs r0 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
  if rsrc2 < 0 **then**
    rdest ← –rsrc2
  else
    rdest ← rsrc2
}

ATTRIBUTES

Function unit	alu
Operation code	44
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`iabs` `fabsval`

DESCRIPTION

The `h_iabs` operation computes the absolute value of rsrc2 and stores the result into r*dest*. The argument is a signed integer; the result is an unsigned integer. This operation requires a zero as first argument. The programmer is advised to use the `iabs` pseudo operation instead.

The `h_iabs` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`h_iabs r0 r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 h_iabs r0 r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 h_iabs r0 r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`h_iabs r0 r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`h_iabs r0 r60 → r110`	r110 ← 0x80000000
r20 = 1	`h_iabs r0 r20 → r120`	r120 ← 1

© TriMedia Technologies, Inc.

Hardware 16-bit store with displacement

h_st16d

SYNTAX

```
[ IF rguard ] h_st16d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    mem[rsrc2 + d + (1 ⊕ bs)] ← rsrc1<7:0>
    mem[rsrc2 + d + (0 ⊕ bs)] ← rsrc1<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	n/a
Issue slots	4, 5

SEE ALSO

`st16` `st16d` `st8` `st8d` `st32` `st32d` `readpcsw` `ijmpf`

DESCRIPTION

The `h_st16d` operation stores the least-significant 16-bit halfword of r*src1* into the memory locations pointed to by the address in r*src2* + *d*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `h_st16d` is misaligned (the memory address computed by r*src2* + *d* is not a multiple of 2), the result of `h_st16d` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The `h_st16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st16d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfe, r80 = 0x44332211	`h_st16d(2) r80 r10`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd05, r70 = 0xaabbccdd	`IF r50 h_st16d(-4) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd06, r70 = 0xaabbccdd	`IF r60 h_st16d(-4) r70 r30`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

h_st32d

Hardware 32-bit store with displacement

SYNTAX

```
[ IF rguard ] h_st32d(d) rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 3
  else
    bs ← 0
  mem[r*src2* + *d* + (3 ⊕ bs)] ← r*src1*<7:0>
  mem[r*src2* + *d* + (2 ⊕ bs)] ← r*src1*<15:8>
  mem[r*src2* + *d* + (1 ⊕ bs)] ← r*src1*<24:16>
  mem[r*src2* + *d* + (0 ⊕ bs)] ← r*src1*<31:24>
}

ATTRIBUTES

Attribute	Value
Function unit	dmem
Operation code	31
Number of operands	2
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	n/a
Issue slots	4, 5

SEE ALSO

st32 st32d st16 st16d st8 st8d readpcsw ijmpf

DESCRIPTION

The h_st32d operation stores all 32 bits of r*src1* into the memory locations pointed to by the address in r*src2* + *d*. The *d* value is an opcode modifier, must be in the range –256 and 252 inclusive, and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If h_st32d is misaligned (the memory address computed by r*src2* + *d* is not a multiple of 4), the result of h_st32d is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The h_st32d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st32d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r80 = 0x44332211	`h_st32d(4) r80 r10`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd0b, r70 = 0xaabbccdd	`IF r50 h_st32d(-8) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd0c, r70 = 0xaabbccdd	`IF r60 h_st32d(-8) r70 r30`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

 © TriMedia Technologies, Inc.

Hardware 8-bit store with displacement

h_st8d

SYNTAX

`[ IF rguard ] h_st8d(d) rsrc1 rsrc2`

FUNCTION

if r*guard* **then**
mem[r*src2* + *d*] ← r*src1*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	7 bits
Modifier range	–64..63
Latency	n/a
Issue slots	4, 5

SEE ALSO

st8 st8d st16 st16d st32 st32d

DESCRIPTION

The h_st8d operation stores the least-significant 8-bit byte of r*src1* into the memory location pointed to by the address formed from the sum r*src2* + *d*. The value of the opcode modifier *d* must be in the range -64 and 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The h_st8d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, h_st8d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`h_st8d(3) r80 r10`	[0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 h_st8d(-4) r70 r20`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 h_st8d(-4) r70 r30`	[0xcfe] ← 0xdd

hicycles

Read clock cycle counter, most-significant word

SYNTAX

```
[ IF rguard ] hicycles → rdest
```

FUNCTION

if *rguard* **then**
  rdest ← CCCOUNT<63:32>

ATTRIBUTES

Function unit	fcomp
Operation code	155
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

cycles curcycles writepcsw

DESCRIPTION

Refer to Section 1.1.5, "CCCOUNT—Clock Cycle Counter" for a description of the CCCOUNT operation. The hicycles operation copies the high 32 bits of the slave register Clock Cycle Counter (CCCOUNT) to the destination register, r*dest*. The contents of the master counter are transferred to the slave CCCOUNT register only on a successful interruptible jump and on processor reset. Thus, if cycles and hicycles are executed without intervening interruptible jumps, the operation pair is guaranteed to be a coherent sample of the master clock-cycle counter. The master counter increments on all cycles (processor-stall and non-stall) if PCSW.CS = 1; otherwise, the counter increments only on non-stall cycles.

The hicycles operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
CCCOUNT_HR = 0xabcdefff12345678	hicycles → r60	r60 ← 0xabcdefff
r10 = 0, CCCOUNT_HR = 0xabcdefff12345678	IF r10 hicycles → r70	no change, since guard is false
r20 = 1, CCCOUNT_HR = 0xabcdefff12345678	IF r20 hicycles → r100	r100 ← 0xabcdefff

 © TriMedia Technologies, Inc.

Absolute value

pseudo-op for h_iabs

iabs

SYNTAX

```
[ IF rguard ] iabs rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 < 0 then
        rdest ← –rsrc1
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	alu
Operation code	44
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

h_iabs dspiabs dspidualabs fabsval

DESCRIPTION

The iabs operation is a pseudo operation transformed by the scheduler into an h_iabs with zero as the first argument and a second argument equal to the iabs argument. (Note: pseudo operations cannot be used in assembly source files.)

The iabs operation computes the absolute value of r*src1* and stores the result into r*dest*. The argument is a signed integer; the result is an unsigned integer.

The iabs operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	iabs r30 → r60	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	IF r10 iabs r40 → r80	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	IF r20 iabs r40 → r90	r90 ← 0xc
r50 = 0x80000001	iabs r50 → r100	r100 ← 0x7fffffff
r60 = 0x80000000	iabs r60 → r110	r110 ← 0x80000000
r20 = 1	iabs r20 → r120	r120 ← 1

iadd

Signed add

SYNTAX

`[ IF rguard ] iadd rsrc1 rsrc2 → rdest`

FUNCTION

if *rguard* **then**
  r*dest* ← r*src1* + r*src2*

ATTRIBUTES

Function unit	alu
Operation code	12
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iaddi carry dspiadd dspidualadd fadd

DESCRIPTION

The `iadd` operation computes the sum r*src1*+r*src2* and stores the result into r*dest*. The operands can be either both signed or unsigned integers. No overflow or underflow detection is performed.

The `iadd` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	`iadd r60 r60 → r80`	r80 ← 0x200
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 iadd r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	`IF r20 iadd r60 r30 → r90`	r90 ← 0x1011
r70 = 0xffffff00, r40 = 0xffffff9c	`iadd r70 r40 → r100`	r100 ← 0xfffffe9c

© TriMedia Technologies, Inc.

Add with immediate

iaddi

SYNTAX

```
[ IF rguard ] iaddi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* + *n*

ATTRIBUTES

Function unit	alu
Operation code	5
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iadd carry

DESCRIPTION

The iaddi operation sums a single argument in r*src1* and an immediate modifier *n* and stores the result in r*dest*. The value of *n* must be between 0 and 127, inclusive.

The iaddi operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf11	iaddi(127) r30 → r70	r70 ← 0xf90
r10 = 0, r40 = 0xffffff9c	IF r10 iaddi(1) r40 → r80	no change, since guard is false
r20 = 1, r40 = 0xffffff9c	IF r20 iaddi(1) r40 → r90	r90 ← 0xffffff9d
r50 = 0x1000	iaddi(15) r50 → r120	r120 ← 0x100f
r60 = 0xfffffff0	iaddi(2) r60 → r110	r110 ← 0xfffffff2
r60 = 0xfffffff0	iaddi(17) r60 → r120	r120 ← 1

iavgonep

Signed average

SYNTAX

```
[ IF rguard ] iavgonep rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← (sign_ext32to64(*rsrc1*) + sign_ext32to64(*rsrc2*) + 1) >> 1;

ATTRIBUTES

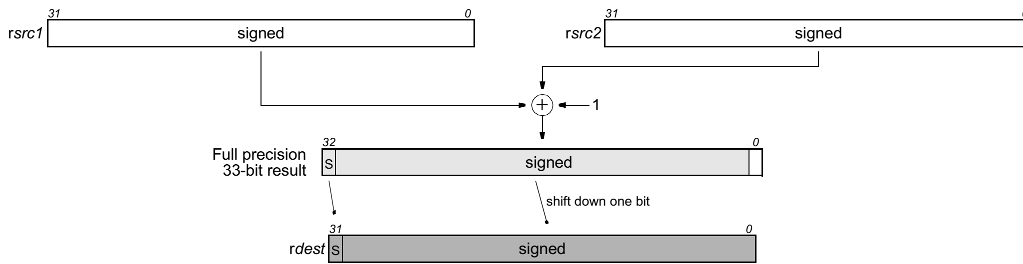
Function unit	dspalu
Operation code	25
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

quadavg iadd

DESCRIPTION

As shown below, the `iavgonep` operation returns the average of the two arguments. This operation computes the sum r*src1*+r*src2*+1, shifts the sum right by 1 bit, and stores the result into r*dest*. The operands are signed integers.



The `iavgonep` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10, r70 = 0x20	`iavgonep r60 r70 → r80`	r80 ← 0x18
r10 = 0, r60 = 0x10, r30 = 0x20	`IF r10 iavgonep r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x9, r30 = 0x20	`IF r20 iavgonep r60 r30 → r90`	r90 ← 0x15
r70 = 0xfffffff7, r40 = 0x2	`iavgonep r70 r40 → r100`	r100 ← 0xfffffffd
r70 = 0xfffffff7, r40 = 0x3	`iavgonep r70 r40 → r100`	r100 ← 0xfffffffd

 © TriMedia Technologies, Inc.

Signed select byte

ibytesel

SYNTAX

```
[ IF rguard ] ibytesel rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	alu
Operation code	56
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

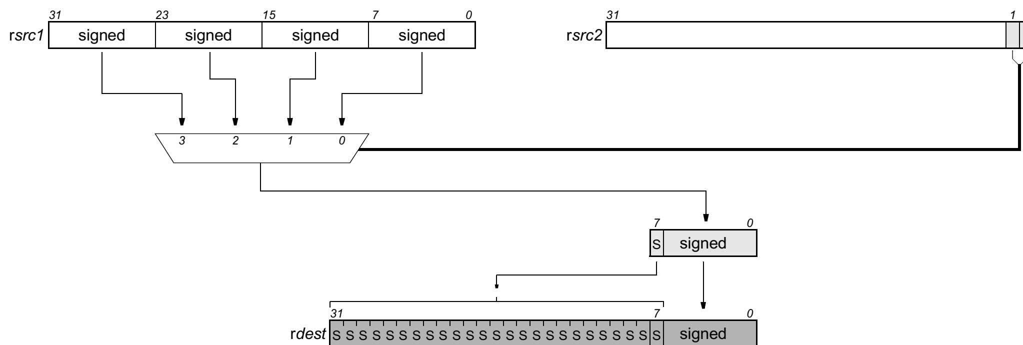
ubytesel sex8 packbytes

FUNCTION

if r*guard* **then {**
  if r*src2* = 0 **then**
    r*dest* ← sign_ext8to32(r*src1*<7:0>)
  else if r*src2* = 1 **then**
    r*dest* ← sign_ext8to32(r*src1*<15:8>)
  else if r*src2* = 2 **then**
    r*dest* ← sign_ext8to32(r*src1*<23:16>)
  else if r*src2* = 3 **then**
    r*dest* ← sign_ext8to32(r*src1*<31:24>)
}

DESCRIPTION

As shown below, the `ibytesel` operation selects one byte from the argument, r*src1*, sign-extends the byte to 32 bits, and stores the result in r*dest*. The value of r*src2* determines which byte is selected, with r*src2*=0 selecting the LSB of r*src1* and r*src2*=3 selecting the MSB of r*src1*. If rsrc2 is not between 0 and 3 inclusive, the result of `ibytesel` is undefined.



The `ibytesel` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x44332211, r40 = 1	`ibytesel r30 r40 → r50`	r50 ← 0x00000022
r10 = 0, r60 = 0xddccbbaa, r70 = 2	`IF r10 ibytesel r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xddccbbaa, r70 = 2	`IF r20 ibytesel r60 r70 → r90`	r90 ← 0xffffffcc
r100 = 0xffffff7f, r110 = 0	`ibytesel r100 r110 → r120`	r120 ← 0x0000007f

iclipi

Clip signed to signed

SYNTAX

```
[ IF rguard ] iclipi rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← min(max(r*src1*, –r*src2*–1), r*src2*)

ATTRIBUTES

Function unit	dspalu
Operation code	74
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

`uclipi` `uclipu` `imin` `imax`

DESCRIPTION

The `iclipi` operation returns the value of r*src1* clipped into the unsigned integer range (–r*src2*–1) to r*src2*, inclusive. The argument r*src1* is considered a signed integer; rsrc2 is considered an unsigned integer and must have a value between 0 and 0x7fffffff inclusive.

The `iclipi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`iclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 iclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 iclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`iclipi r100 r110 → r120`	r120 ← 0xffc00000

© TriMedia Technologies, Inc.

Invalidate all instruction cache blocks

iclr

SYNTAX

```
[ IF rguard ] iclr
```

FUNCTION

```
if rguard then {
    block ← 0
    for all blocks in instruction cache {
        icache_reset_valid_block(block)
        block ← block + 1
    }
}
```

ATTRIBUTES

Function unit	branch
Operation code	184
Number of operands	0
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	2, 3, 4

SEE ALSO

`dcb` `dinvalid`

DESCRIPTION

The `iclr` operation resets the valid bits of all blocks in the instruction cache.

`iclr` does clear the valid bits of locked blocks. `iclr` does not change the replacement status of instruction-cache blocks.

`iclr` ensures coherency between caches and main memory by discarding all pending prefetch operations.

The side effect time behavior of `iclr` is such that if instruction *i* performs an iclr, instructions *i, i+1, i+2* will be included in the discard from the instruction cache, but *i+3* will be retained.

The `iclr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`iclr`	
r10 = 0	`IF r10 iclr`	no change and no stall cycles, since guard is false
r20 = 1	`IF r20 iclr`	

ident

Identity
pseudo-op for iadd

SYNTAX
`[ IF rguard ] ident rsrc1 → rdest`

FUNCTION
if r*guard* **then**
r*dest* ← r*src1*

ATTRIBUTES

Function unit	alu
Operation code	12
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO
`iadd`

DESCRIPTION
The `ident` operation is a pseudo operation transformed by the scheduler into an `iadd` with r0 (always contains 0) as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

The `ident` operation copies the argument r*src1* to r*dest*. It is used by the instruction scheduler to implement register to register copying.

The `ident` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x100	`ident r30 → r40`	r40 ← 0x100
r10 = 0, r50 = 0x12345678	`IF r10 ident r50 → r60`	no change, since guard is false
r20 = 1, r50 = 0x12345678	`IF r20 ident r50 → r70`	r70 ← 0x12345678

© TriMedia Technologies, Inc.

Signed compare equal

ieql

SYNTAX

```
[ IF rguard ] ieql rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	37
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ueql ieqli ineq

DESCRIPTION

The ieql operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The ieql operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	ieql r30 r40 → r80	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	IF r10 ieql r60 r30 → r50	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	IF r20 ieql r50 r60 → r90	r90 ← 1
r70 = 0x80000000, r40 = 4	ieql r70 r40 → r100	r100 ← 0
r70 = 0x80000000	ieql r70 r70 → r110	r110 ← 1

ieqli

Signed compare equal with immediate

SYNTAX

```
[ IF rguard ] ieqli(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	4
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`ieql` `igeqi` `ueqli` `ineqi`

DESCRIPTION

The `ieqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ieqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ieqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ieqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ieqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ieqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ieqli(63) r40 → r100`	r100 ← 0
r60 = 0xffffffc0	`ieqli(-64) r60 → r120`	r120 ← 1

 © TriMedia Technologies, Inc.

Sum of products of signed 16-bit halfwords

ifir16

SYNTAX

```
[ IF rguard ] ifir16 rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← sign_ext16to32(r*src1*<31:16>) × sign_ext16to32(r*src2*<31:16>) +
    sign_ext16to32(r*src1*<15:0>) × sign_ext16to32(r*src2*<15:0>)

ATTRIBUTES

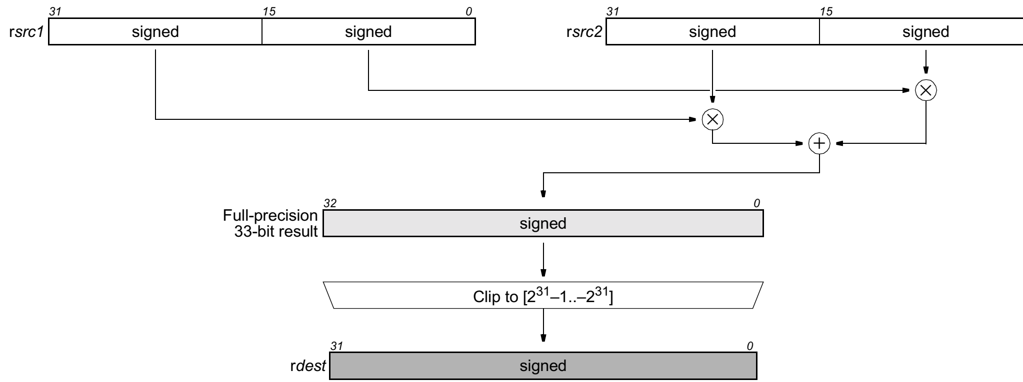
Function unit	dspmul
Operation code	93
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ii ifir8ui ufir8uu ifir16

DESCRIPTION

As shown below, the `ifir16` operation computes two separate products of the two pairs of corresponding 16-bit halfwords of r*src1* and r*src2*; the two products are summed, and the result is written to r*dest*. All values are considered signed; thus, the intermediate products and the final sum of products are signed. All intermediate computations are performed without loss of precision; the final sum of products is clipped into the range [0x80000000..0x7fffffff] before being written into r*dest*.



The `ifir16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00020003, r40 = 0x00010002	`ifir16 r30 r40 → r50`	r50 ← 0x8
r10 = 0, r60 = 0xff9c0064, r70 = 0x0064ff9c	`IF r10 ifir16 r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xff9c0064, r70 = 0x0064ff9c	`IF r20 ifir16 r60 r70 → r90`	r90 ← 0xffffb1e0
r30 = 0x00020003, r70 = 0x0064ff9c	`ifir16 r30 r70 → r100`	r100 ← 0xffffff9c

ifir8ii

Signed sum of products of signed bytes

SYNTAX

```
[ IF rguard ] ifir8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← sign_ext8to32(*rsrc1*<31:24>) × sign_ext8to32(*rsrc2*<31:24>) +
sign_ext8to32(*rsrc1*<23:16>) × sign_ext8to32(*rsrc2*<23:16>) +
sign_ext8to32(*rsrc1*<15:8>) × sign_ext8to32(*rsrc2*<15:8>) +
sign_ext8to32(*rsrc1*<7:0>) × sign_ext8to32(*rsrc2*<7:0>)

ATTRIBUTES

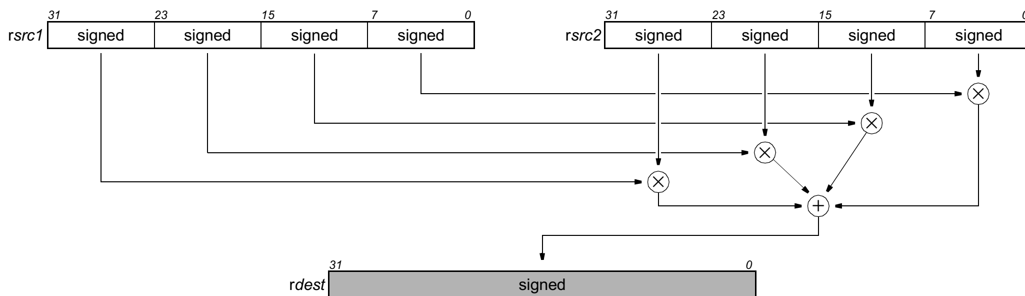
Function unit	dspmul
Operation code	92
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ui ufir8uu ifir16 ufir16

DESCRIPTION

As shown below, the `ifir8ii` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. All values are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ii` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ii r70 r30 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ii r70 r30 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ii r80 r40 → r110`	r110 ← 0xffff63c0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ii r50 r60 → r120`	r120 ← 0x200

© TriMedia Technologies, Inc.

Signed sum of products of unsigned/signed bytes

ifir8ui

SYNTAX

```
[ IF rguard ] ifir8ui rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext8to32(r*src1*<31:24>) × sign_ext8to32(r*src2*<31:24>) +
zero_ext8to32(r*src1*<23:16>) × sign_ext8to32(r*src2*<23:16>) +
zero_ext8to32(r*src1*<15:8>) × sign_ext8to32(r*src2*<15:8>) +
zero_ext8to32(r*src1*<7:0>) × sign_ext8to32(r*src2*<7:0>)

ATTRIBUTES

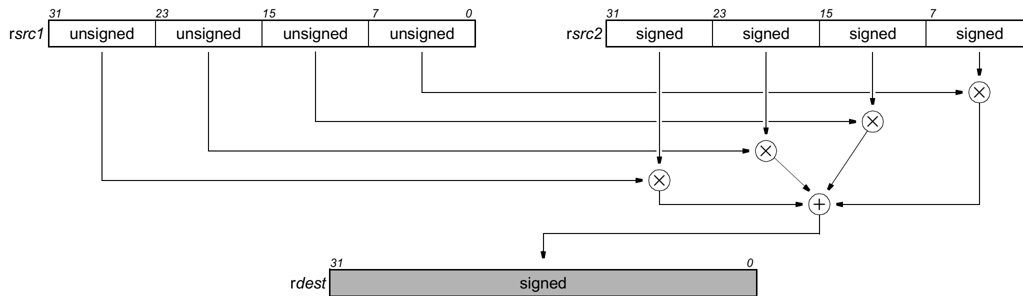
Function unit	dspmul
Operation code	91
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ii ufir8uu ifir16 ufir16

DESCRIPTION

As shown below, the `ifir8ui` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. The bytes from r*src1* are considered unsigned, but the bytes from r*src2* are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ui` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ui r30 r70 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ui r30 r70 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ui r40 r80 → r110`	r110 ← 0x2bc0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ui r60 r50 → r120`	r120 ← 0xfffe0200

ifixieee

Convert floating-point to integer using PCSW rounding mode

SYNTAX

```
[ IF rguard ] ifixieee rsrc1 → rdest
```

FUNCTION

if *rguard* **then {**
  r*dest* ← (long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	121
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ufixieee ifixrz ufixrz

DESCRIPTION

The `ifixieee` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixieee` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifixieeeflags` operation computes the exception flags that would result from an individual `ifixieee`.

The `ifixieee` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixieee r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixieee r35 → r102`	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieee r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieee r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixieee r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixieee r50 → r115`	r115 ← -2, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixieee r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixieee r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixieee r80 → r122`	r122 ← 0, INV flag set

 © TriMedia Technologies, Inc.

IEEE status flags from convert floating-point to integer using PCSW rounding mode

ifixieeeflags

SYNTAX

`[ IF rguard ] ifixieeeflags rsrc1 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	122
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

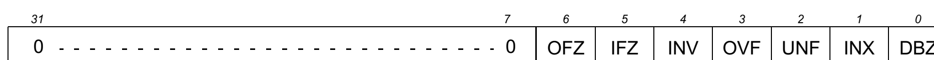
SEE ALSO

`ifixieee` `ufixieeeflags` `ifixrzflags` `ufixrzflags`

DESCRIPTION

The `ifixieeeflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to a signed integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ifixieeeflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixieeeflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ifixieeeflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixieeeflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixieeeflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ifixieeeflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ifixieeeflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x00400000 (5.877471754e-39)	`ifixieeeflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ifixieeeflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ifixieeeflags r80 → r122`	r122 ← 0x10 (INV)

ifixrz

Convert floating-point to integer with round toward zero

SYNTAX

`[ IF rguard ] ifixrz rsrc1 → rdest`

FUNCTION

```
if rguard then {
    rdest ← (long) ((float)rsrc1)
}
```

ATTRIBUTES

Function unit	falu
Operation code	21
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`ifixieee` `ufixieee` `ufixrz`

DESCRIPTION

The `ifixrz` operation converts the single-precision IEEE floating-point value in r*src1* to a signed integer and writes the result into r*dest*. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding for ANSI C. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ifixrz` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifixrzflags` operation computes the exception flags that would result from an individual `ifixrz`.

The `ifixrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixrz r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ifixrz r35 → r102`	r102 ← 2, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixrz r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixrz r40 → r110`	r110 ← 0x80000000 (-2^{31}), INV flag set
r45 = 0x7f800000 (+INF))	`ifixrz r45 → r112`	r112 ← 0x7fffffff (2^{31}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ifixrz r50 → r115`	r115 ← -1, INX flag set
r60 = 0x00400000 (5.877471754e-39)	`ifixrz r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ifixrz r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ifixrz r80 → r122`	r122 ← 0, INV flag set

 © TriMedia Technologies, Inc.

IEEE status flags from convert floating-point to integer with round toward zero

SYNTAX

```
[ IF rguard ] ifixrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	129
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

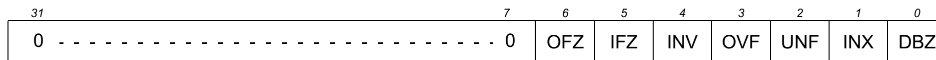
SEE ALSO

ifixrz ufixrzflags
ifixieeeflags
ufixieeeflags

DESCRIPTION

The `ifixrzflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to a signed integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. If r*src1* is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ifixrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ifixrzflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ifixrzflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ifixrzflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ifixrzflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ifixrzflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ifixrzflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x00400000 (5.877471754e-39)	`ifixrzflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ifixrzflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ifixrzflags r80 → r122`	r122 ← 0x10 (INV)

iflip

If non-zero negate

SYNTAX

```
[ IF rguard ] iflip rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* = 0 **then**
    r*dest* ← r*src2*
  else
    r*dest* ← –r*src2*
}

ATTRIBUTES

Function unit	dspalu
Operation code	77
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

`inonzero` `izero`

DESCRIPTION

The `iflip` operation copies r*src2* to r*dest* if r*src1* = 0; otherwise (if r*src1* != 0), r*dest* is set to the two's-complement of r*src2*. All values are signed integers.

The `iflip` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0, r40 = 1	`iflip r30 r40 → r50`	r50 ← 0x1
r10 = 0, r60 = 0xffff0000, r70 = 0xabc	`IF r10 iflip r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xffff0000, r70 = 0xabc	`IF r20 iflip r60 r70 → r90`	r90 ← 0xfffff544
r30 = 0, r100 = 0xffffff9c	`iflip r30 r100 → r110`	r110 ← 0xffffff9c
r40 = 1, r110 = 0xffffffff	`iflip r40 r110 → r120`	r120 ← 0x1

© TriMedia Technologies, Inc.

Convert signed integer to floating-point

ifloat

SYNTAX

```
[ IF rguard ] ifloat rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	20
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`ufloat` `ifloatrz` `ufloatrz` `ifixieee` `ifloatflags`

DESCRIPTION

The `ifloat` operation converts the signed integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If `ifloat` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifloatflags` operation computes the exception flags that would result from an individual `ifloat`.

The `ifloat` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloat r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (-1)	`ifloat r40 → r105`	r105 ← 0xbf800000 (-1.0)
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloat r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloat r50 → r115`	r115 ← 0xc0400000 (–3.0)
r60 = 0x7fffffff (2147483647)	`ifloat r60 → r117`	r117 ← 0x4f000000 (2.147483648e+9), INX flag set
r70 = 0x80000000 (-2147483648)	`ifloat r70 → r120`	r120 ← 0xcf000000 (-2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ifloat r80 → r122`	r122 ← 0x4f000000 (2.147483648e+9), INX flag set

ifloatflags IEEE status flags from convert signed integer to floating-point

SYNTAX

```
[ IF rguard ] ifloatflags rsrc1 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← ieee_flags((float) ((long)*rsrc1*))

ATTRIBUTES

Function unit	falu
Operation code	130
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

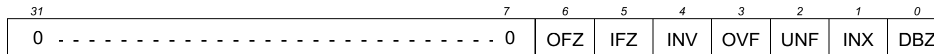
SEE ALSO

ifloat ifloatrzflags ufloatflags ufloatrzflags

DESCRIPTION

The `ifloatflags` operation computes the IEEE exceptions that would result from converting the signed integer in *rsrc1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into *rdest*. The bit vector stored in *rdest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW.

The `ifloatflags` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (-1)	`ifloatflags r40 → r105`	r105 ← 0
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatflags r50 → r115`	r115 ← 0
r60 = 0x7fffffff (2147483647)	`ifloatflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (-2147483648)	`ifloatflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ifloatflags r80 → r122`	r122 ← 0x02 (INX)

 © TriMedia Technologies, Inc.

Convert signed integer to floating-point with rounding toward zero

ifloatrz

SYNTAX

`[ IF rguard ] ifloatrz rsrc1 → rdest`

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	117
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`ifloat` `ufloatrz` `ifixieee` `ifloatflags`

DESCRIPTION

The `ifloatrz` operation converts the signed integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ifloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ifloatrzflags` operation computes the exception flags that would result from an individual `ifloatrz`.

The `ifloatrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (-1)	`ifloatrz r40 → r105`	r105 ← 0xbf800000 (-1.0)
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatrz r50 → r115`	r115 ← 0xc0400000 (–3.0)
r60 = 0x7fffffff (2147483647)	`ifloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (-2147483648)	`ifloatrz r70 → r120`	r120 ← 0xcf000000 (-2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ifloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set

ifloatrzflags

IEEE status flags from convert signed integer to floating-point with rounding toward zero

SYNTAX

```
[ IF rguard ] ifloatrzflags rsrc1 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← ieee_flags((float) ((long)*rsrc1*))

ATTRIBUTES

Function unit	falu
Operation code	118
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

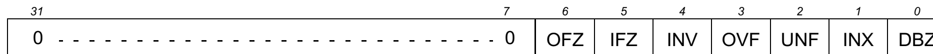
SEE ALSO

ifloatrz ifloatflags ufloatflags ufloatrzflags

DESCRIPTION

The `ifloatrzflags` operation computes the IEEE exceptions that would result from converting the signed integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored.

The `ifloatrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ifloatrzflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (-1)	`ifloatrzflags r40 → r105`	r105 ← 0
r10 = 0, r50 = 0xfffffffd	`IF r10 ifloatrzflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ifloatrzflags r50 → r115`	r115 ← 0
r60 = 0x7fffffff (2147483647)	`ifloatrzflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (-2147483648)	`ifloatrzflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ifloatrzflags r80 → r122`	r122 ← 0x02 (INX)

 © TriMedia Technologies, Inc.

Signed compare greater or equal

igeq

SYNTAX

```
[ IF rguard ] igeq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 >= rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ileq igeqi

DESCRIPTION

The igeq operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The igeq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	igeq r30 r40 → r80	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	IF r10 igeq r60 r30 → r50	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	IF r20 igeq r50 r60 → r90	r90 ← 1
r70 = 0x80000000, r40 = 4	igeq r70 r40 → r100	r100 ← 0
r70 = 0x80000000	igeq r70 r70 → r110	r110 ← 1

igeqi

Signed compare greater or equal with immediate

SYNTAX

```
[ IF rguard ] igeqi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* >= *n* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	1
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq iles ieqli

DESCRIPTION

The `igeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`igeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`igeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`igeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 igeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 igeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`igeqi(-64) r60 → r120`	r120 ← 0

© TriMedia Technologies, Inc.

Signed compare greater

igtr

SYNTAX

```
[ IF rguard ] igtr rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* > r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	15
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles igtri

DESCRIPTION

The igtr operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The igtr operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	igtr r30 r40 → r80	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	IF r10 igtr r60 r30 → r50	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	IF r20 igtr r50 r60 → r90	r90 ← 1
r70 = 0x80000000, r40 = 4	igtr r70 r40 → r100	r100 ← 0
r70 = 0x80000000	igtr r70 r70 → r110	r110 ← 0

igtri

Signed compare greater with immediate

SYNTAX

```
[ IF rguard ] igtri(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	0
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr igeqi

DESCRIPTION

The `igtri` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `igtri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`igtri(2) r30 → r80`	r80 ← 1
r30 = 3	`igtri(3) r30 → r90`	r90 ← 0
r30 = 3	`igtri(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 igtri(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 igtri(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`igtri(-64) r60 → r120`	r120 ← 0

 © TriMedia Technologies, Inc.

Signed immediate

iimm

SYNTAX

`iimm(n) → rdest`

FUNCTION

rdest ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0x80000000 ..0x7fffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

uimm

DESCRIPTION

The `iimm` operation stores the signed 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`iimm(2) → r10`	r10 ← 2
	`iimm(0x100) → r20`	r20 ← 0x100
	`iimm(0xfffc0000) → r30`	r30 ← 0xfffc0000

ijmpf Interruptible indirect jump on false

SYNTAX

```
[ IF rguard ] ijmpf rsrc1 rsrc2
```

ATTRIBUTES

Function unit	branch
Operation code	181
Number of operands	2
Modifier	no
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpt ijmpi

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 0 **then {**
    DPC ← r*src2*
    if exception is pending **then**
      service exception
    elseif interrupt is pending **then**
      service interrupts
    else
      PC, SPC ← r*src2*
  }
}

DESCRIPTION

The `ijmpf` operation conditionally changes the program flow and allows pending interrupts or exceptions to be serviced. If neither interrupts or exceptions are pending and the LSB of r*src1* is 0, the DPC, PC, and SPC registers are set equal to r*src2*. If an interrupt or exception is pending and the LSB of r*src1* is 0, DPC is set equal to r*src2* and the service routine is invoked, where exceptions have priorities over interrupts. If the LSB of r*src1* is 1, program execution continues with the next sequential instruction.

The `ijmpf` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 0, r70 = 0x330	`ijmpf r50 r70`	program execution continues at 0x330 after first servicing pending interrupts
r20 = 1, r70 = 0x330	`ijmpf r20 r70`	since r20 is true, program execution continues with next sequential instruction
r30 = 0, r50 = 0, r60 = 0x8000	`IF r30 ijmpf r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 0, r60 = 0x8000	`IF r40 ijmpf r50 r60`	program execution continues at 0x8000 after first servicing pending interrupts

 © TriMedia Technologies, Inc.

Interruptible jump immediate

ijmpi

ATTRIBUTES

Function unit	branch
Operation code	179
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Delay	3
Issue slots	2, 3, 4

SEE ALSO

`jmpf` `jmpt` `jmpi` `ijmpf` `ijmpt`

SYNTAX

```
[ IF rguard ] ijmpi(address)
```

FUNCTION

if r*guard* **then {**
  DPC ← *address*
  if exception is pending **then**
    service exception
  else if interrupt is pending **then**
    service interrupts
  else
    PC, SPC ← address
}

DESCRIPTION

The `ijmpi` operation changes the program flow and allows pending interrupts or exceptions to be serviced. If no interrupts or exceptions are pending, the DPC, PC, and SPC registers are set equal to *address*. If an exception or interrupts is pending, DPC is set equal to *address* and a service routine is invoked, where exceptions have priorities over interrupts. *address* is an immediate opcode modifier.

The `ijmpi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds a condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified.

EXAMPLES

Initial Values	Operation	Result
	`ijmpi(0x330)`	program execution continues at 0x330
r30 = 0	`IF r30 ijmpi(0x8000)`	since guard is false, program execution continues with next sequential instruction
r40 = 1	`IF r40 ijmpi(0x8000)`	program execution continues at 0x8000

ijmpt — Interruptible indirect jump on true

SYNTAX

```
[ IF rguard ] ijmpt rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    if (rsrc1 & 1) = 1 then {
        DPC ← rsrc2
        if exception is pending then
            service exception
        elseif interrupt is pending then
            service interrupts
        else
            PC, SPC ← rsrc2
    }
}
```

ATTRIBUTES

Function unit	branch
Operation code	177
Number of operands	2
Modifier	no
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpt jmpi ijmpf ijmpi

DESCRIPTION

The ijmpt operation conditionally changes the program flow and allows pending interrupts or exceptions to be serviced. If no interrupts or exceptions are pending and the LSB of r*src1* is 1, the DPC, PC, and SPC registers are set equal to r*src2*. If an exception or interrupt is pending and the LSB of r*src1* is 1, DPC is set equal to r*src2* and a service routine is invoked, where exceptions have priority over interrupts. If the LSB of r*src1* is 0, program execution continues with the next sequential instruction.

The ijmpt operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken and PC, DPC, and SPC are not modified regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 1, r70 = 0x330	`ijmpt r50 r70`	program execution continues at 0x330 after first servicing pending interrupts
r20 = 0, r70 = 0x330	`ijmpt r20 r70`	since r20 is false, program execution continues with next sequential instruction
r30 = 0, r50 = 1, r60 = 0x8000	`IF r30 ijmpt r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 1, r60 = 0x8000	`IF r40 ijmpt r50 r60`	program execution continues at 0x8000 after first servicing pending interrupts

© TriMedia Technologies, Inc.

Signed 16-bit load

pseudo-op for ild16d(0)

ild16

SYNTAX

```
[ IF rguard ] ild16 rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[(rsrc1 +(1 ⊕ bs)]
    temp<15:8> ← mem[(rsrc1 + (0 ⊕ bs)]
    rdest ← sign_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	6
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16d ild16r ild16x

DESCRIPTION

The `ild16` operation is a pseudo operation transformed by the scheduler into an `ild16d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ild16` operation loads the 16-bit memory value from the address contained in r*src1*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 2, the result of `ild16` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22, [0xd01] = 0x11	`ild16 r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r30 ild16 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r40 ild16 r20 → r80`	r80 ← 0xffff8433
r50 = 0xd01	`ild16 r50 → r90`	r90 undefined, since 0xd01 is not a multiple of 2

ild16d Signed 16-bit load with displacement

SYNTAX

```
[ IF rguard ] ild16d(d) rsrc1 → rdest
```

ATTRIBUTES

Function unit	dmem
Operation code	6
Number of operands	1
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	3
Issue slots	4, 5

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[(rsrc1 + d + (1 ⊕ bs)]
    temp<15:8> ← mem[(rsrc1 + d + (0 ⊕ bs)]
    rdest ← sign_ext16to32(temp<15:0>)
}
```

SEE ALSO

ild16 uld16 uld16d ild16r uld16r ild16x uld16x

DESCRIPTION

The ild16d operation loads the 16-bit memory value from the address computed by r*src1* + *d*, sign extends it to 32 bits, and stores the result in r*dest*. The *d* value is an opcode modifier, must be in the range –128 to 126 inclusive, and must be a multiple of 2. If the memory address computed by r*src1* + *d* is not a multiple of 2, the result of ild16d is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by ild16d to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The ild16d operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and ild16d has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22, [0xd03] = 0x11	`ild16d(2) r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r30 ild16d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r40 ild16d(-4) r20 → r80`	r80 ← 0xffff8433
r50 = 0xd01	`ild16d(-4) r50 → r90`	r90 undefined, since 0xd01 +(–4) is not a multiple of 2

 © TriMedia Technologies, Inc.

Signed 16-bit load with index

ild16r

SYNTAX

```
[ IF rguard ] ild16r rsrc1 rsrc2 → rdest
```

ATTRIBUTES

Function unit	dmem
Operation code	195
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[(rsrc1 + rsrc2 +(1 ⊕ bs)]
    temp<15:8> ← mem[(rsrc1 + rsrc2 + (0 ⊕ bs)]
    rdest ← sign_ext16to32(temp<15:0>)
}
```

SEE ALSO

ild16 uld16 ild16d uld16d uld16r ild16x uld16x

DESCRIPTION

The `ild16r` operation loads the 16-bit memory value from the address computed by r*src1* + r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 2, the result of `ild16r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22, [0xd03] = 0x11	`ild16r r10 r20 → r80`	r80 ← 0x00002211
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 ild16r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 ild16r r40 r30 → r100`	r100 ← 0xffff8433
r70 = 0xd01, r30 = 0xfffffffc	`ild16r r70 r30 → r110`	r110 undefined, since 0xd01 +(–4) is not a multiple of 2

ild16x

Signed 16-bit load with scaled index

SYNTAX

```
[ IF rguard ] ild16x rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 1
  else
    bs ← 0
  temp<7:0> ← mem[(*rsrc1* + (2 × *rsrc2*) + (1 ⊕ bs)]
  temp<15:8> ← mem[(*rsrc1* + (2 × *rsrc2*) + (0 ⊕ bs)]
  rdest ← sign_ext16to32(temp<15:0>)
}

ATTRIBUTES

Function unit	dmem
Operation code	196
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild16 uld16 ild16d uld16d ild16r uld16r uld16x

DESCRIPTION

The `ild16x` operation loads the 16-bit memory value from the address computed by r*src1* + 2×r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. If the memory address computed by r*src1* + 2×r*src2* is not a multiple of 2, the result of `ild16x` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `ild16x` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild16x` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild16x` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r30 = 1, [0xd02] = 0x22, [0xd03] = 0x11	`ild16x r10 r30 → r100`	r100 ← 0x00002211
r50 = 0, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 ild16x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 ild16x r40 r20 → r90`	r90 ← 0xffff8433
r70 = 0xd01, r30 = 1	`ild16x r70 r30 → r110`	r110 undefined, since 0xd01 + 2×1 is not a multiple of 2

 © TriMedia Technologies, Inc.

Signed 8-bit load

pseudo-op for ild8d(0)

ild8

SYNTAX

```
[ IF rguard ] ild8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(mem[r*src1*])

ATTRIBUTES

Function unit	dmem
Operation code	192
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

`uld8` `ild8d` `uld8d` `ild8r` `uld8r`

DESCRIPTION

The `ild8` operation is a pseudo operation transformed by the scheduler into an `ild8d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ild8` operation loads the 8-bit memory value from the address contained in r*src1*, sign extends it to 32 bits, and stores the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22	`ild8 r10 → r60`	r60 ← 0x00000022
r30 = 0, r20 = 0xd04, [0xd04] = 0x84	`IF r30 ild8 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84	`IF r40 ild8 r20 → r80`	r80 ← 0xffffff84
r50 = 0xd01, [0xd01] = 0x33	`ild8 r50 → r90`	r90 ← 0x00000033

ild8d

Signed 8-bit load with displacement

SYNTAX

```
[ IF rguard ] ild8d(d) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(mem[r*src1* + *d*])

ATTRIBUTES

Function unit	dmem
Operation code	192
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	3
Issue slots	4, 5

SEE ALSO

`ild8` `uld8` `uld8d` `ild8r` `uld8r`

DESCRIPTION

The `ild8d` operation loads the 8-bit memory value from the address computed by r*src1* + *d*, sign extends it to 32 bits, and stores the result in r*dest*. The *d* value is an opcode modifier in the range -64 to 63, inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22	`ild8d(2) r10 → r60`	r60 ← 0x000022
r30 = 0, r20 = 0xd04, [0xd00] = 0x84	`IF r30 ild8d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84	`IF r40 ild8d(-4) r20 → r80`	r80 ← 0xffffff84
r50 = 0xd05, [0xd01] = 0x33	`ild8d(-4) r50 → r90`	r90 ← 0x00000033

© TriMedia Technologies, Inc.

Signed 8-bit load with index

ild8r

SYNTAX

`[ IF rguard ] ild8r rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext8to32(mem[r*src1* + r*src2*])

ATTRIBUTES

Function unit	dmem
Operation code	193
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8 ild8d uld8d uld8r

DESCRIPTION

The `ild8r` operation loads the 8-bit memory value from the address computed by r*src1* + r*src2*, sign extends it to 32 bits, and stores the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `ild8r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `ild8r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ild8r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22	`ild8r r10 r20 → r80`	r80 ← 0x00000022
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r50 ild8r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r60 ild8r r40 r30 → r100`	r100 ← 0xffffff84
r70 = 0xd05, r30 = 0xfffffffc, [0xd01] = 0x33	`ild8r r70 r30 → r110`	r110 ← 0x00000033

ileq

Signed compare less or equal

pseudo-op for igeq

SYNTAX

```
[ IF rguard ] ileq rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
  if r*src1* <= r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	14
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`igeq` `ileqi`

DESCRIPTION

The `ileq` operation is a pseudo operation transformed by the scheduler into an `igeq` with the arguments exchanged (`ileq`'s r*src1* is `igeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ileq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ileq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ileq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	`IF r20 ileq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ileq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ileq r70 r70 → r110`	r110 ← 1

 © TriMedia Technologies, Inc.

Signed compare less or equal with immediate

ileqi

SYNTAX

```
[ IF rguard ] ileqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 <= n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	42
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`ileq` `igeqi`

DESCRIPTION

The `ileqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ileqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ileqi(2) r30 → r80`	r80 ← 0
r30 = 3	`ileqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ileqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ileqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ileqi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ileqi(-64) r60 → r120`	r120 ← 1

iles

Signed compare less

pseudo-op for igtr

SYNTAX

```
[ IF rguard ] iles rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 < rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	15
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr ilesi

DESCRIPTION

The iles operation is a pseudo operation transformed by the scheduler into an igtr with the arguments exchanged (iles's r*src1* is igtr's r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The iles operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The iles operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	iles r30 r40 → r80	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	IF r10 iles r60 r30 → r50	no change, since guard is false
r20 = 1, r50 = 0x1000, 0x100	IF r20 iles r50 r60 → r90	r90 ← 0
r70 = 0x80000000, r40 = 4	iles r70 r40 → r100	r100 ← 1
r70 = 0x80000000	iles r70 r70 → r110	r110 ← 0

© TriMedia Technologies, Inc.

Signed compare less with immediate

ilesi

SYNTAX

```
[ IF rguard ] ilesi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 < n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	2
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iles ileqi

DESCRIPTION

The ilesi operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The ilesi operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	ilesi(2) r30 → r80	r80 ← 0
r30 = 3	ilesi(3) r30 → r90	r90 ← 0
r30 = 3	ilesi(4) r30 → r100	r100 ← 1
r10 = 0, r40 = 0x100	IF r10 ilesi(63) r40 → r50	no change, since guard is false
r20 = 1, r40 = 0x100	IF r20 ilesi(63) r40 → r100	r100 ← 0
r60 = 0x80000000	ilesi(-64) r60 → r120	r120 ← 1

imax Signed maximum

SYNTAX

```
[ IF rguard ] imax rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* > r*src2* **then**
    r*dest* ← r*src1*
  else
    r*dest* ← r*src2*
}

ATTRIBUTES

Function unit	dspalu
Operation code	24
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

`imin`

DESCRIPTION

The `imax` operation sets the destination register, r*dest*, to the contents of r*src1* if r*src1*>r*src2*; otherwise, r*dest* is set to the contents of r*src2*. The arguments are treated as signed integers.

The `imax` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`imax r30 r20 → r80`	r80 ← 2
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 imax r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 imax r60 r40 → r90`	r90 ← 0x100
r70 = 0xffffff00, r40 = 0xffffff9c	`imax r70 r40 → r100`	r100 ← 0xffffff9c

© TriMedia Technologies, Inc.

Signed minimum

imin

SYNTAX

```
[ IF rguard ] imin rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	23
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

imax

DESCRIPTION

The `imin` operation sets the destination register, r*dest*, to the contents of r*src2* if r*src1*>r*src2*; otherwise, r*dest* is set to the contents of r*src1*. The arguments are treated as signed integers.

The `imin` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`imin r30 r20 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 imin r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 imin r60 r40 → r90`	r90 ← 0xffffff9c
r70 = 0xffffff00, r40 = 0xffffff9c	`imin r70 r40 → r100`	r100 ← 0xffffff00

imul

Signed multiply

SYNTAX

```
[ IF rguard ] imul rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
temp ← (sign_ext32to64(*rsrc1*) × sign_ext32to64(*rsrc2*))
rdest ← temp<31:0>

ATTRIBUTES

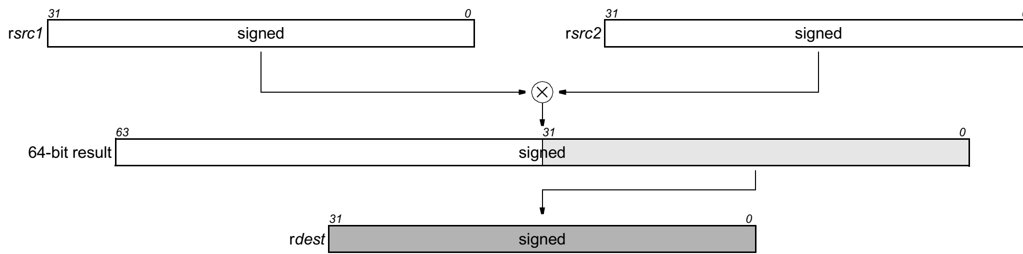
Function unit	ifmul
Operation code	27
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umul imulm umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the `imul` operation computes the product r*src1*×r*src2* and writes the least-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered signed integers. No overflow or underflow detection is performed.



The `imul` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	`imul r60 r60 → r80`	r80 ← 0x10000
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 imul r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	`IF r20 imul r60 r30 → r90`	r90 ← 0xf1100
r70 = 0xffffff00, r40 = 0xffffff9c	`imul r70 r40 → r100`	r100 ← 0x6400

 © TriMedia Technologies, Inc.

Signed multiply, return most-significant 32 bits

imulm

SYNTAX

```
[ IF rguard ] imulm rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
temp ← (sign_ext32to64(*rsrc1*) × sign_ext32to64(*rsrc2*))
rdest ← temp<63:32>

ATTRIBUTES

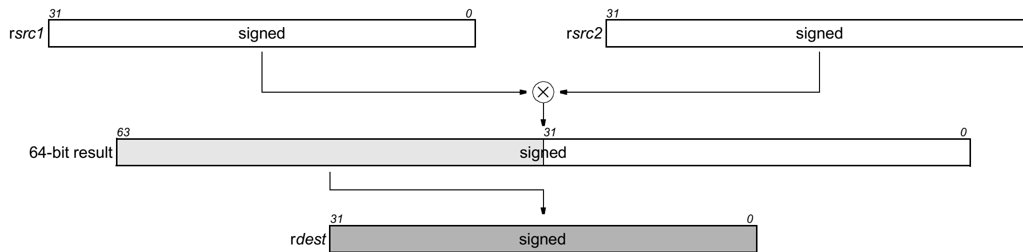
Function unit	ifmul
Operation code	139
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the `imulm` operation computes the product *rsrc1*×*rsrc2* and writes the most-significant 32 bits of the full 64-bit product into *rdest*. The operands are considered signed integers.



The `imulm` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10000	`imulm r60 r60 → r80`	r80 ← 0x00000001
r10 = 0, r60 = 0x100, r30 = 0xf11	`IF r10 imulm r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x10001000, r30 = 0xf1100000	`IF r20 imulm r60 r30 → r90`	r90 ← 0xff10ff11
r70 = 0xffffff00, r40 = 0x64	`imulm r70 r40 → r100`	r100 ← 0xffffffff

ineg

Signed negate
pseudo-op for isub

SYNTAX

```
[ IF rguard ] ineg rsrc1 → rdest
```

FUNCTION

if *rguard* **then**
  r*dest* ← –rsrc1

ATTRIBUTES

Function unit	alu
Operation code	13
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`isub`

DESCRIPTION

The `ineg` operation is a pseudo operation transformed by the scheduler into an `isub` with r0 (always contains 0) as the first argument and r*src1* as the second argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ineg` operation computes the negative of r*src1* and writes the result into r*dest*. The argument is a signed integer; the result is an unsigned integer. If rsrc1 = 0x80000000, then `ineg` returns 0x80000000 since the positive value is not representable.

The `ineg` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffffffff	`ineg r30 → r60`	r60 ← 0x00000001
r10 = 0, r40 = 0xfffffff4	`IF r10 ineg r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xfffffff4	`IF r20 ineg r40 → r90`	r90 ← 0xc
r50 = 0x80000001	`ineg r50 → r100`	r100 ← 0x7fffffff
r60 = 0x80000000	`ineg r60 → r110`	r110 ← 0x80000000
r20 = 1	`ineg r20 → r120`	r120 ← 0xffffffff

© TriMedia Technologies, Inc.

Signed compare not equal

ineq

SYNTAX

```
[ IF rguard ] ineq rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* != r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	39
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieql igtr ineqi

DESCRIPTION

The `ineq` operation sets the destination register, r*dest*, to 1 if the two arguments, r*src1* and r*src2*, are not equal; otherwise, r*dest* is set to 0.

The `ineq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ineq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x1000, r30 = 3	`IF r10 ineq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ineq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ineq r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ineq r70 r70 → r110`	r110 ← 0

ineqi

Signed compare not equal with immediate

SYNTAX

```
[ IF rguard ] ineqi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 != n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	3
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ineq igeqi ieqli

DESCRIPTION

The `ineqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as signed integers.

The `ineqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ineqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ineqi(3) r30 → r90`	r90 ← 0
r30 = 3	`ineqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ineqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ineqi(63) r40 → r100`	r100 ← 1
r60 = 0xffffffc0	`ineqi(-64) r60 → r120`	r120 ← 0

 © TriMedia Technologies, Inc.

If nonzero select zero

inonzero

SYNTAX

```
[ IF rguard ] inonzero rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 != 0 then
        rdest ← 0
    else
        rdest ← rsrc2
}
```

ATTRIBUTES

Function unit	alu
Operation code	47
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

izero iflip

DESCRIPTION

The inonzero operation writes 0 into r*dest* if the value of r*src1* is not zero; otherwise, r*src2* is copied to r*dest*. The operands are considered signed integers.

The inonzero operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`inonzero r30 r20 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 inonzero r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 inonzero r60 r40 → r90`	r90 ← 0
r10 = 0, r40 = 0xffffff9c	`inonzero r10 r40 → r100`	r100 ← 0xffffff9c
r20 = 1, r60 = 0x100	`inonzero r20 r60 → r110`	r110 ← 0
r10 = 0, r70 = 0x456789	`inonzero r10 r70 → r120`	r120 ← 0x456789

isub **Subtract**

SYNTAX

`[ IF rguard ] isub rsrc1 rsrc2 → rdest`

FUNCTION

if *rguard* **then**
rdest ← *rsrc1* – *rsrc2*

ATTRIBUTES

Function unit	alu
Operation code	13
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isubi borrow dspisub dspidualsub fsub

DESCRIPTION

The `isub` operation computes the difference r*src1*–r*src2* and writes the result into r*dest*. The operands can be either both signed or unsigned integers. No overflow or underflow detection is performed.

The `isub` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`isub r30 r40 → r80`	r80 ← 0xffffffff
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 isub r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 isub r50 r60 → r90`	r90 ← 0xf00
r70 = 0x80000000, r40 = 4	`isub r70 r40 → r100`	r100 ← 0x7ffffffc

 © TriMedia Technologies, Inc.

Subtract with immediate

isubi

SYNTAX

```
[ IF rguard ] isubi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← r*src1* – *n*

ATTRIBUTES

Function unit	alu
Operation code	32
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

isub borrow

DESCRIPTION

The `isubi` operation computes the difference of a single argument in r*src1* and an immediate modifier *n* and stores the result in r*dest*. The value of *n* must be between 0 and 127, inclusive.

The `isubi` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xf11	`isubi(127) r30 → r70`	r70 ← 0xe92
r10 = 0, r40 = 0xffffff9c	`IF r10 isubi(1) r40 → r80`	no change, since guard is false
r20 = 1, r40 = 0xffffff9c	`IF r20 isubi(1) r40 → r90`	r90 ← 0xffffff9b
r50 = 0x1000	`isubi(15) r50 → r120`	r120 ← 0x0ff1
r60 = 0xfffffff0	`isubi(2) r60 → r110`	r110 ← 0xffffffee
r20 = 1	`isubi(17) r20 → r120`	r120 ← 0xfffffff0

izero

If zero select zero

SYNTAX

```
[ IF rguard ] izero rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
  if r*src1* = 0 **then**
    r*dest* ← 0
  else
    r*dest* ← r*src2*
}

ATTRIBUTES

Function unit	alu
Operation code	46
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

inonzero iflip

DESCRIPTION

The `izero` operation writes 0 into r*dest* if the value of r*src1* is equal to zero; otherwise, r*src2* is copied to r*dest*. The operands are considered signed integers.

The `izero` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 2, r20 = 1	`izero r30 r20 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 2	`IF r10 izero r60 r30 → r50`	no change, since guard is false
r20 = 1, r60 = 0x100, r40 = 0xffffff9c	`IF r20 izero r60 r40 → r90`	r90 ← 0xffffff9c
r10 = 0, r40 = 0xffffff9c	`izero r10 r40 → r100`	r100 ← 0
r20 = 1, r60 = 0x100	`izero r20 r60 → r110`	r110 ← 0x100
r20 = 1, r70 = 0x456789	`izero r20 r70 → r120`	r120 ← 0x456789

© TriMedia Technologies, Inc.

Indirect jump on false

jmpf

SYNTAX

```
[ IF rguard ] jmpf rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 0 **then**
    PC ← r*src2*
}

ATTRIBUTES

Function unit	branch
Operation code	180
Number of operands	2
Modifier	No
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpt jmpi ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpf` operation conditionally changes the program flow. If the LSB of r*src1* is 0, the PC register is set equal to r*src2*; otherwise, program execution continues with the next sequential instruction.

The `jmpf` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 0, r70 = 0x330	`jmpf r50 r70`	program execution continues at 0x330
r20 = 1, r70 = 0x330	`jmpf r20 r70`	since r20 is true, program execution continues with next sequential instruction
r30 = 0, r50 = 0, r60 = 0x8000	`IF r30 jmpf r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 0, r60 = 0x8000	`IF r40 jmpf r50 r60`	program execution continues at 0x8000

jmpi

Jump immediate

SYNTAX

```
[ IF rguard ] jmpi(address)
```

FUNCTION

if *rguard* **then**
PC ← *address*

ATTRIBUTES

Function unit	branch
Operation code	178
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Delay	3
Issue slots	2, 3, 4

SEE ALSO

`jmpf` `jmpt` `ijmpf` `ijmpt` `ijmpi`

DESCRIPTION

The `jmpi` operation changes the program flow by setting the PC register equal to the immediate opcode modifier *address*.

The `jmpi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds a condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken.

EXAMPLES

Initial Values	Operation	Result
	`jmpi(0x330)`	program execution continues at 0x330
r30 = 0	`IF r30 jmpi(0x8000)`	since guard is false, program execution continues with next sequential instruction
r40 = 1	`IF r40 jmpi(0x8000)`	program execution continues at 0x8000

© TriMedia Technologies, Inc.

Indirect jump on true

jmpt

SYNTAX

```
[ IF rguard ] jmpt rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if (r*src1* & 1) = 1 **then**
    PC ← r*src2*
}

ATTRIBUTES

Function unit	branch
Operation code	176
Number of operands	2
Modifier	no
Modifier range	—
Delay	3
Issue slots	2, 3, 4

SEE ALSO

jmpf jmpi ijmpf ijmpt ijmpi

DESCRIPTION

The `jmpt` operation conditionally changes the program flow. If the LSB of r*src1* is 1, the PC register is set equal to r*src2*; otherwise, program execution continues with the next sequential instruction.

The `jmpt` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB adds another condition to the jump. If the LSB of r*guard* is 1, the instruction executes as previously described; otherwise, the jump will not be taken regardless of the value of r*src1*.

EXAMPLES

Initial Values	Operation	Result
r50 = 1, r70 = 0x330	`jmpt r50 r70`	program execution continues at 0x330
r20 = 0, r70 = 0x330	`jmpt r20 r70`	since r20 is false, program execution continues with next sequential instruction
r30 = 0, r50 = 1, r60 = 0x8000	`IF r30 jmpt r50 r60`	since guard is false, program execution continues with next sequential instruction
r40 = 1, r50 = 1, r60 = 0x8000	`IF r40 jmpt r50 r60`	program execution continues at 0x8000

ld32

32-bit load

pseudo-op for ld32d(0)

SYNTAX

```
[ IF rguard ] ld32 rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 3
    else
        bs ← 0
    rdest<7:0> ← mem[rsrc1 + (3 ⊕ bs)]
    rdest<15:8> ← mem[rsrc1 + (2 ⊕ bs)]
    rdest<23:16> ← mem[rsrc1 + (1 ⊕ bs)]
    rdest<31:24> ← mem[rsrc1 + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	7
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ld32d ld32r ld32x st32 st32d h_st32d

DESCRIPTION

The `ld32` operation is a pseudo operation transformed by the scheduler into an `ld32d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `ld32` operation loads the 32-bit memory value from the address contained in r*src1* and stores the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 4, the result of `ld32` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The `ld32` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `ld32`.

The `ld32` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ld32` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	`ld32 r10 → r60`	r60 ← 0x84332211
r30 = 0, r20 = 0xd04, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r30 ld32 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r40 ld32 r20 → r80`	r80 ← 0x48665544
r50 = 0xd01	`ld32 r50 → r90`	r90 undefined, since 0xd01 is not a multiple of 4

© TriMedia Technologies, Inc.

32-bit load with displacement

ld32d

SYNTAX

```
[ IF rguard ] ld32d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 3
    else
        bs ← 0
    rdest<7:0> ← mem[rsrc1 + d + (3 ⊕ bs)]
    rdest<15:8> ← mem[rsrc1 + d + (2 ⊕ bs)]
    rdest<23:16> ← mem[rsrc1 + d + (1 ⊕ bs)]
    rdest<31:24> ← mem[rsrc1 + d + (0 ⊕ bs)]
}
```

ATTRIBUTES

Attribute	Value
Function unit	dmem
Operation code	7
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	4, 5

SEE ALSO

`ld32` `ld32r` `ld32x` `st32` `st32d` `h_st32d`

DESCRIPTION

The `ld32d` operation loads the 32-bit memory value from the address computed by r*src1* + *d* and stores the result in r*dest*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4. If the memory address computed by r*src1* + *d* is not a multiple of 4, the result of `ld32d` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The `ld32d` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `ld32d`.

The `ld32d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ld32d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	`ld32d(4) r10 → r60`	r60 ← 0x84332211
r30 = 0, r20 = 0xd0c, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r30 ld32d(-8) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd0c, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r40 ld32d(-8) r20 → r80`	r80 ← 0x48665544
r50 = 0xd01	`ld32d(-8) r50 → r90`	r90 undefined, since 0xd01 +(–8) is not a multiple of 4

ld32r

32-bit load with index

SYNTAX

```
[ IF rguard ] ld32r rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 3
  else
    bs ← 0
  r*dest*<7:0> ← mem[r*src1* + r*src2* + (3 ⊕ bs)]
  r*dest*<15:8> ← mem[r*src1* + r*src2* + (2 ⊕ bs)]
  r*dest*<23:16> ← mem[r*src1* + r*src2* + (1 ⊕ bs)]
  r*dest*<31:24> ← mem[r*src1* + r*src2* + (0 ⊕ bs)]
}

ATTRIBUTES

Function unit	dmem
Operation code	200
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

`ld32` `ld32d` `ld32x` `st32` `st32d` `h_st32d`

DESCRIPTION

The `ld32r` operation loads the 32-bit memory value from the address computed by r*src1* + r*src2* and stores the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 4, the result of `ld32r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The `ld32r` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `ld32r`.

The `ld32r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ld32r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r20 = 0x4, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	`ld32r r10 r20 → r80`	r80 ← 0x84332211
r50 = 0, r40 = 0xd0c, r30 = 0xfffffff8, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r50 ld32r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd0c, r30 = 0xfffffff8, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r60 ld32r r40 r30 → r100`	r100 ← 0x48665544
r50 = 0xd01, r30 = 0xfffffff8	`ld32r r70 r30 → r110`	r110 undefined, since 0xd01 +(–8) is not a multiple of 2

 © TriMedia Technologies, Inc.

32-bit load with scaled index

ld32x

SYNTAX

```
[ IF rguard ] ld32x rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 3
    else
        bs ← 0
    rdest<7:0> ← mem[rsrc1 + (4 × rsrc2) +(3 ⊕ bs)]
    rdest<15:8> ← mem[rsrc1 + (4 × rsrc2) + (2 ⊕ bs)]
    rdest<23:16> ← mem[rsrc1 + (4 × rsrc2) + (1 ⊕ bs)]
    rdest<31:24> ← mem[rsrc1 + (4 × rsrc2) + (0 ⊕ bs)]
}
```

ATTRIBUTES

Function unit	dmem
Operation code	201
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

`ld32` `ld32d` `ld32r` `st32` `st32d` `h_st32d`

DESCRIPTION

The `ld32x` operation loads the 32-bit memory value from the address computed by r*src1* + 4×r*src2* and stores the result in r*dest*. If the memory address computed by r*src1* + 4×r*src2* is not a multiple of 4, the result of `ld32x` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The `ld32x` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `ld32x`.

The `ld32x` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `ld32x` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r30 = 0x1, [0xd00] = 0x84, [0xd01] = 0x33, [0xd02] = 0x22, [0xd03] = 0x11	`ld32x r10 r30 → r100`	r100 ← 0x84332211
r50 = 0, r40 = 0xd0c, r20 = 0xfffffffe, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r50 ld32x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd0c, r20 = 0xfffffffe, [0xd04] = 0x48, [0xd05] = 0x66, [0xd06] = 0x55, [0xd07] = 0x44	`IF r60 ld32x r40 r20 → r90`	r90 ← 0x48665544
r70 = 0xd01, r30 = 0x1	`ld32x r70 r30 → r110`	r110 undefined, since 0xd01 + 4×1 is not a multiple of 4

lsl

Logical shift left
pseudo-op for asl

SYNTAX
```
[ IF rguard ] lsl rsrc1 rsrc2 → rdest
```

FUNCTION
if *rguard* **then {**
  n ← *rsrc2*<4:0>
  rdest<31:n> ← *rsrc1*<31–n:0>
  rdest<n–1:0> ← 0
  if rsrc2<31:5> != 0 {
    rdest <- 0
  }
}

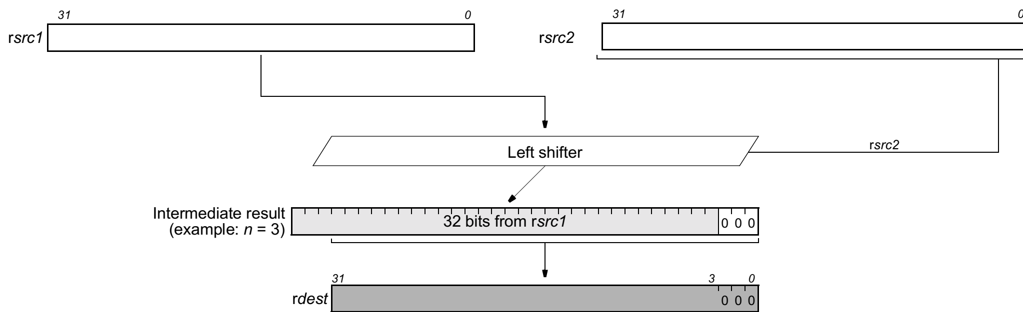
ATTRIBUTES

Function unit	shifter
Operation code	19
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO
asl asli asr asri lsli lsr lsri rol roli

DESCRIPTION
The `lsl` operation is a pseudo operation that is transformed by the scheduler into an `asl` with the same arguments. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `lsl` operation takes two arguments, r*src1* and r*src2*. R*src2* specify an unsigned shift amount, and rdest is set to r*src1* logically shifted left by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `lsl` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`lsl r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 lsl r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 lsl r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`lsl r70 r40 → r120`	r120 ← 0xfffffff0
r80 = 0xe, r50 = 0xfffffffe	`lsl r80 r50 → r125`	r125 ← 0x00000000 (shift by more than 32))
r30 = 0x7008000f, r45 = 0x20	`lsl r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x80000000	`lsl r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x23	`lsl r30 r45 → r100`	r100 ← 0x00000000

 © TriMedia Technologies, Inc.

Logical shift left immediate

pseudo-op for asli

SYNTAX

```
[ IF rguard ] lsli(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<31:*n*> ← r*src1*<31–*n*:0>
r*dest*<*n*–1:0> ← 0
}

ATTRIBUTES

Function unit	shifter
Operation code	11
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

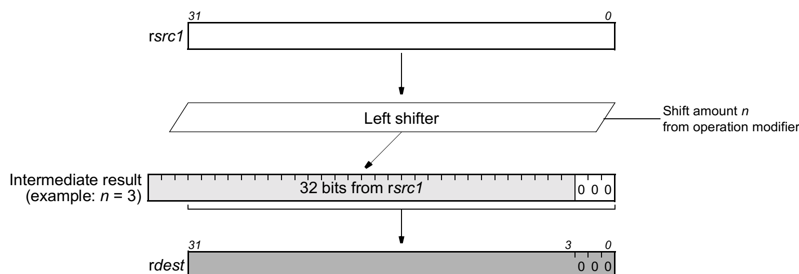
SEE ALSO

asl asli asr asri lsl lsr lsri rol roli

DESCRIPTION

The `lsli` operation is a pseudo operation that is transformed by the scheduler into an `asli` with the same argument and opcode modifier. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `lsli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* logically shifted left by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros are shifted into the LSBs of r*dest* while the MSBs shifted out of r*src1* are lost.



The `lsli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`lsli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 lsli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 lsli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`lsli(2) r70 → r120`	r120 ← 0xfffffff0
r80 = 0xe	`lsli(30) r80 → r125`	r125 ← 0x80000000

lsr

Logical shift right

SYNTAX

```
[ IF rguard ] lsr rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
n ← rsrc2<4:0>
rdest<31:32–n> ← 0
rdest<31–n:0> ← *rsrc1*<31:n>
if rsrc2<31:5> != 0 {
rdest <- 0
}
}

ATTRIBUTES

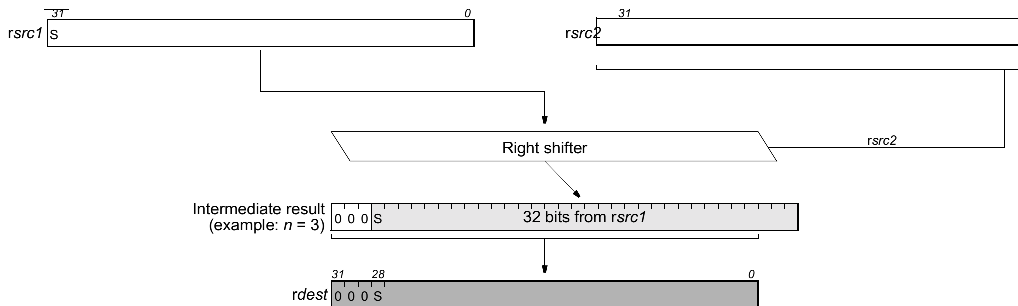
Function unit	shifter
Operation code	96
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr asri lsl lsli lsri rol roli

DESCRIPTION

As shown below, the `lsr` operation takes two arguments, r*src1* and r*src2*. R*src2* specifies an unsigned shift amount, and r*src1* is logically shifted right by this amount. If the rsrc2<31:5> value is not zero, then take this as a shift by 32 or more bits. Zeros fill vacated bits from the left.



The `lsr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f, r20 = 1	`lsr r30 r20 → r50`	r50 ← 0x38040007
r30 = 0x7008000f, r42 = 2	`lsr r30 r42 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f, r44 = 4	`IF r10 lsr r30 r44 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f, r44 = 4	`IF r20 lsr r30 r44 → r80`	r80 ← 0x07008000
r40 = 0x80030007, r44 = 4	`lsr r40 r44 → r90`	r90 ← 0x08003000
r30 = 0x7008000f, r45 = 0x1f	`lsr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x1f	`lsr r30 r45 → r100`	r100 ← 0x00000001
r30 = 0x7008000f, r45 = 0x20	`lsr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x80000000	`lsr r30 r45 → r100`	r100 ← 0x00000000
r30 = 0x8008000f, r45 = 0x23	`lsr r30 r45 → r100`	r100 ← 0x00000000

 © TriMedia Technologies, Inc.

Logical shift right immediate

SYNTAX

```
[ IF rguard ] lsri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  r*dest*<31:32–*n*> ← 0
  r*dest*<31–*n*:0> ← r*src1*<31:*n*>
}

ATTRIBUTES

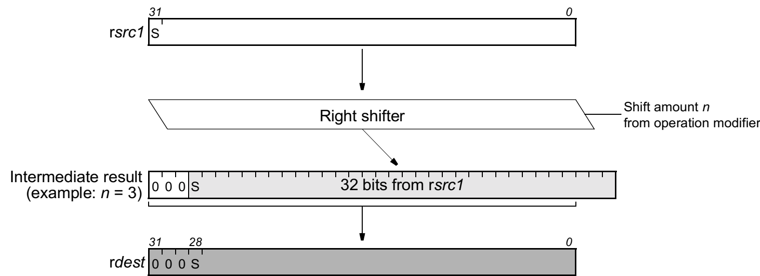
Function unit	shifter
Operation code	9
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

asl asli asr asri lsl lsli lsr rol roli

DESCRIPTION

As shown below, the `lsri` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* that is equal to r*src1* logically shifted right by *n* bits. The value of *n* must be between 0 and 31, inclusive. Zeros fill vacated bits from the left.



The `lsri` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x7008000f	`lsri(1) r30 → r50`	r50 ← 0x38040007
r30 = 0x7008000f	`lsri(2) r30 → r60`	r60 ← 0x1c020003
r10 = 0, r30 = 0x7008000f	`IF r10 lsri(4) r30 → r70`	no change, since guard is false
r20 = 1, r30 = 0x7008000f	`IF r20 lsri(4) r30 → r80`	r80 ← 0x07008000
r40 = 0x80030007	`lsri(4) r40 → r90`	r90 ← 0x08003000
r30 = 0x7008000f	`lsri(31) r30 → r100`	r100 ← 0x00000000
r40 = 0x80030007	`lsri(31) r40 → r110`	r110 ← 0x00000001

mergedual16lsb

Merge dual 16-bit lsb bytes

SYNTAX

```
[ IF rguard ] mergedual16lsb rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    rdest<31:24> <- rsrc1<23:16>
    rdest<23:16> <- rsrc1<7:0>
    rdest<15:8> <- rsrc2<23:16>
    rdest<7:0> <- rsrc2<7:0>
}
```

ATTRIBUTES

Function unit	shifter
Operation code	103
Number of operands	2
Modifier	No
Modifier range	-
Latency	1
Issue slots	1,2

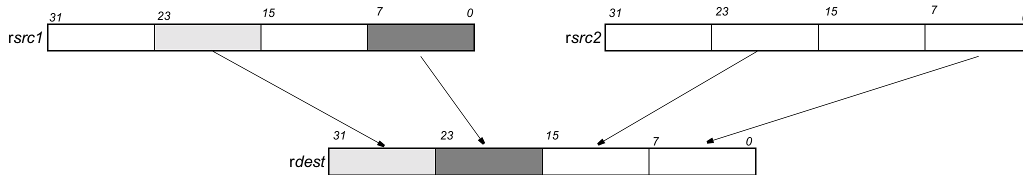
SEE ALSO

mergelsb mergemsb
pack16lsb pack16msb

DESCRIPTION

The arguments rsrc1 and rsrc2 are vectors of two 16-bit data. The mergedual16lsb operation merges the least significant bytes from each 16-bit data rsrc1 and rsrc2 into one 32-bit data in dest register, to convert to quad 8-bit.

The mergedual16lsb operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the modification of the destination register. If the LSB of rguard is 1, rdest is written; otherwise, rdest is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergedual16lsb r30 r40 -> r50`	r50 <- 0x3478bbdd
r10 = 0, r30 = 0x12345678, r40 = 0xaabbccdd	`IF r10 mergedual16lsb r30 r40 -> r50`	no change, since guard is false
r10 = 1, r30 = 0x01020304, r40 = 0x0a0b0c0d	`IF r10 mergedual16lsb r30 r40 -> r50`	r50 <- 0x02040b0d

 © TriMedia Technologies, Inc.

Merge least-significant byte

mergelsb

SYNTAX

```
[ IF rguard ] mergelsb rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    rdest<7:0> ← rsrc2<7:0>
    rdest<15:8> ← rsrc1<7:0>
    rdest<23:16> ← rsrc2<15:8>
    rdest<31:24> ← rsrc1<15:8>
}
```

ATTRIBUTES

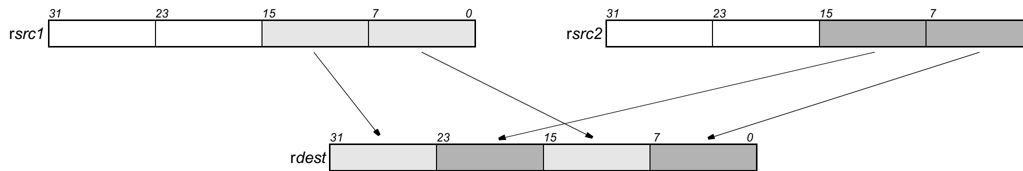
Function unit	alu
Operation code	57
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb packbytes mergemsb

DESCRIPTION

As shown below, the mergelsb operation interleaves the two pairs of least-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The least-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the least-significant byte from r*src1* is packed into the second-least-significant byte of r*dest*; the second-least-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the second-least-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The mergelsb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	mergelsb r30 r40 → r50	r50 ← 0x56cc78dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	IF r10 mergelsb r40 r30 → r60	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	IF r20 mergelsb r40 r30 → r70	r70 ← 0xcc56dd78

mergemsb

Merge most-significant byte

SYNTAX

```
[ IF rguard ] mergemsb rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
 rdest<7:0> ← *rsrc2*<23:15>
 rdest<15:8> ← *rsrc1*<23:15>
 rdest<23:16> ← *rsrc2*<31:24>
 rdest<31:24> ← *rsrc1*<31:24>
}

ATTRIBUTES

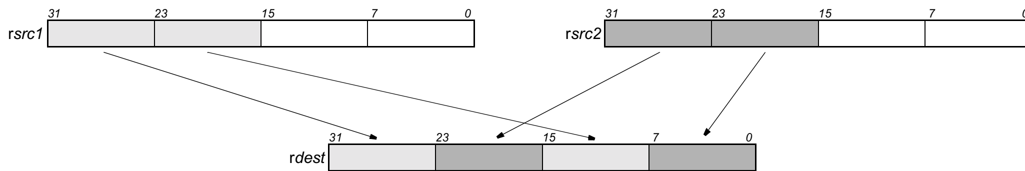
Attribute	Value
Function unit	alu
Operation code	58
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb packbytes mergelsb

DESCRIPTION

As shown below, the `mergemsb` operation interleaves the two pairs of most-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The second-most-significant byte from r*src2* is packed into the least-significant byte of r*dest*; the second-most-significant byte from r*src1* is packed into the second-least-significant byte of r*dest*; the most-significant byte from r*src2* is packed into the second-most-significant byte of r*dest*; and the most-significant byte from r*src1* is packed into the most-significant byte of r*dest*.



The `mergemsb` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergemsb r30 r40 → r50`	r50 ← 0x12aa34bb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergemsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergemsb r40 r30 → r70`	r70 ← 0xaa12bb34

 © TriMedia Technologies, Inc.

No operation

nop

SYNTAX

```
nop
```

FUNCTION

No operation

ATTRIBUTES

Function unit	-
Operation code	-
Number of operands	-
Modifier	-
Modifier range	-
Latency	1
Issue slots	1-5

SEE ALSO

DESCRIPTION

The NOP operation does not change any DSPCPU32 state. It is mainly used to fill-up the empty issue slots. Only two bits are used to code the NOP operation.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`nop`	No change in any regsiters

pack16lsb

Pack least-significant 16-bit halfwords

SYNTAX

```
[ IF rguard ] pack16lsb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
    r*dest*<15:0> ← r*src2*<15:0>
    r*dest*<31:16> ← r*src1*<15:0>
}

ATTRIBUTES

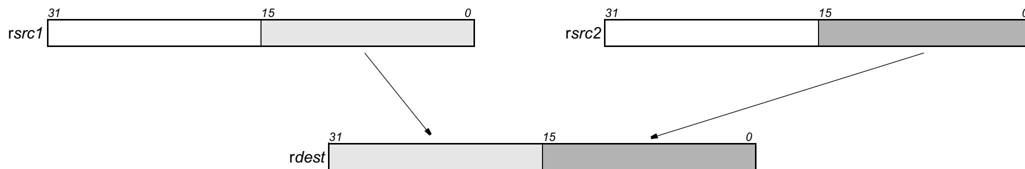
Function unit	alu
Operation code	53
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16msb packbytes mergelsb mergemsb

DESCRIPTION

As shown below, the `pack16lsb` operation packs the two least-significant halfwords from the arguments r*src1* and r*src2* into r*dest*. The halfword from r*src1* is packed into the most-significant halfword of r*dest*; the halfword from r*src2* is packed into the least-significant halfword of r*dest*.



The `pack16lsb` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`pack16lsb r30 r40 → r50`	r50 ← 0x5678ccdd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 pack16lsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 pack16lsb r40 r30 → r70`	r70 ← 0xccdd5678

 © TriMedia Technologies, Inc.

Pack most-significant 16 bits

pack16msb

SYNTAX

```
[ IF rguard ] pack16msb rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest*<15:0> ← r*src2*<31:16>
r*dest*<31:16> ← r*src1*<31:16>
}

ATTRIBUTES

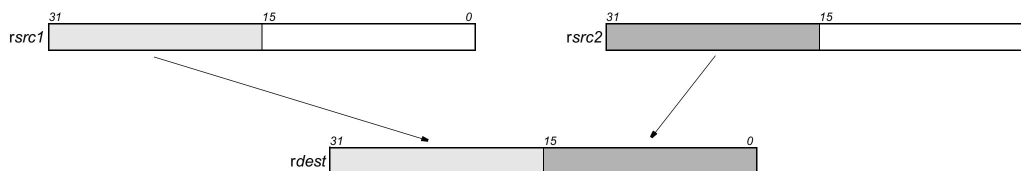
Function unit	alu
Operation code	54
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the pack16msb operation packs the two most-significant halfwords from the arguments r*src1* and r*src2* into r*dest*. The halfword from r*src1* is packed into the most-significant halfword of r*dest*; the halfword from r*src2* is packed into the least-significant halfword of r*dest*.



The pack16msb operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`pack16msb r30 r40 → r50`	r50 ← 0x1234aabb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 pack16msb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 pack16msb r40 r30 → r70`	r70 ← 0xaabb1234

packbytes

Pack least-significant byte

SYNTAX

```
[ IF rguard ] packbytes rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
   rdest<7:0> ← *rsrc2*<7:0>
   rdest<15:8> ← *rsrc1*<7:0>
}

ATTRIBUTES

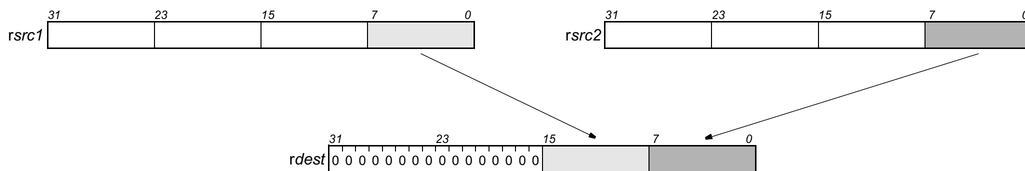
Function unit	alu
Operation code	52
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb mergelsb mergemsb

DESCRIPTION

As shown below, the `packbytes` operation packs the two least-significant bytes from the arguments r*src1* and r*src2* into r*dest*. The byte from r*src1* is packed into the second-least-significant byte of r*dest*; the byte from r*src2* is packed into the least-significant byte of r*dest*. The two most-significant bytes of r*dest* are filled with zeros.



The `packbytes` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`packbytes r30 r40 → r50`	r50 ← 0x000078dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 packbytes r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 packbytes r40 r30 → r70`	r70 ← 0x0000dd78

© TriMedia Technologies, Inc.

prefetch

pseudo-op for prefd(0)

pref

SYNTAX

```
[ IF rguard ] pref rsrc1
```

FUNCTION

if r*guard* **then {**
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + 0) & cache_block_mask]
}

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	-
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefd prefr allocd allocr allocx

DESCRIPTION

The pref operation is a pseudo operation transformed by the scheduler into an prefd(0) with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The pref operation loads the one full cache block size of memory value from the address computed by ((rsrc1+0) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A pref operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`pref r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 pref r10`	since guard is false, pref operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 pref r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.

pref16x prefetch with 16-bit scaled index

SYNTAX

```
[ IF rguard ] pref16x rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + (2 x rscr2)) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	211
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref32x prefd prefr allocd allocr allocx

DESCRIPTION

The pref16x operation loads one full cache block from the main memory at the address computed by ((rsrc1+ (2 x rscr2)) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. The data cache has hardware to simultaneously sustain two cache misses or prefetches. A pref16x operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref16x operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xc cache_block_size = 0x40	`pref16x r10 r12`	Loads a cache line for the address space from 0xabc0 to 0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xc, cache_block_size = 0x40	`IF r11 pref16x r10 r12`	since guard is false, pref16x operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 pref16x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

 © TriMedia Technologies, Inc.

prefetch with 32-bit scaled index

pref32x

SYNTAX

```
[ IF rguard ] pref32x rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + (4 x rscr2)) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	212
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x prefd prefr allocd allocr allocx

DESCRIPTION

The pref32x operation loads the one full cache block size of memory value from the address computed by ((rsrc1+ (4 x rscr2)) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A pref32x operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The pref32x operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`pref32x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 pref32x r10 r12`	since guard is false, pref32x operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 pref32x r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

prefd

prefetch with displacement

SYNTAX

`[ IF rguard ] prefd(d) rsrc1`

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + d) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	209
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefr allocd allocr allocx

DESCRIPTION

The prefd operation loads the one full cache block size of memory value from the address computed by ((rsrc1+d) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefd operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The prefd operation optionally takes a guard, specified in rguard. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of rguard is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, cache_block_size = 0x40	`prefd(0xd) r10`	Loads a cache line for the address space from 0xabc0 to 0x0xabff from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, cache_block_size = 0x40	`IF r11 prefd(0xd) r10`	since guard is false, prefd operation is not executed
r10 = 0xabff, r11 = 1, cache_block_size = 0x40	`IF r11 prefd(ox1) r10`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

© TriMedia Technologies, Inc.

prefetch with index

prefr

SYNTAX

```
[ IF rguard ] prefr rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    cache_block_mask = ~(cache_block_size - 1)
    data_cache <- mem[(rsrc1 + rscr2) & cache_block_mask]
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	210
Number of operands	2
Modifier	No
Modifier range	-
Latency	-
Issue slots	5

SEE ALSO

pref16x pref32x prefd allocd allocr allocx

DESCRIPTION

The prefr operation loads the one full cache block size of memory value from the address computed by ((rsrc1+rscr2) & cache_block_mask) and stores the data into the data cache. This operation is not guaranteed to be executed. The prefetch unit will not execute this operation when the data to be prefetched is already in the data cache. A prefr operation will not be executed when the cache is already occupied with 2 cache misses, when the operation is issued.

The `prefr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the execution of the prefetch operation. If the LSB of r*guard* is 1, prefetch operation is executed; otherwise, it is not executed..

EXAMPLES

Initial Values	Operation	Result
r10 = 0xabcd, r12 = 0xd cache_block_size = 0x40	`prefr r10 r12`	Loads a cache line for the address space from 0xabc0 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.
r10 = 0xabcd, r11 = 0, r12=0xd, cache_block_size = 0x40	`IF r11 prefr r10 r12`	since guard is false, prefr operation is not executed
r10 = 0xabff, r11 = 1, r12 =0x1, cache_block_size = 0x40	`IF r11 prefr r10 r12`	Loads a cache line for the address space from 0xac00 to 0x0xac3f from the main memory. If the data is already in the cache, the operation is not executed.

quadavg

Unsigned byte-wise quad average

SYNTAX

```
[ IF rguard ] quadavg rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then {**
temp ← (zero_ext8to32(r*src1*<7:0>) + zero_ext8to32(r*src2*<7:0>) + 1) / 2
r*dest*<7:0> ← temp<7:0>
temp ← (zero_ext8to32(r*src1*<15:8>) + zero_ext8to32(r*src2*<15:8>) + 1) / 2
r*dest*<15:8> ← temp<7:0>
temp ← (zero_ext8to32(r*src1*<23:16>) + zero_ext8to32(r*src2*<23:16>) + 1) / 2
r*dest*<23:16> ← temp<7:0>
temp ← (zero_ext8to32(r*src1*<31:24>) + zero_ext8to32(r*src2*<31:24>) + 1) / 2
r*dest*<31:24> ← temp<7:0>
}

ATTRIBUTES

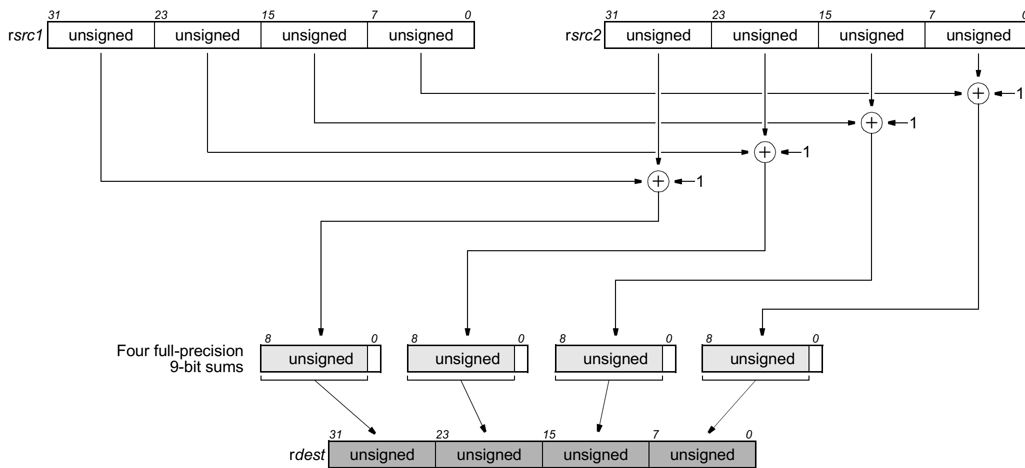
Function unit	dspalu
Operation code	73
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iavgonep dspuquadaddui ifir8ii

DESCRIPTION

As shown below, the quadavg operation computes four separate averages of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The least-significant 8 bits of each average is written to the corresponding byte in r*dest*. No overflow or underflow detection is performed.



The quadavg operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xffffff02	`quadavg r30 r40 → r50`	r50 ← 0x81808008
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	`IF r10 quadavg r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	`IF r20 quadavg r60 r70 → r90`	r90 ← 0x809c6480

© TriMedia Technologies, Inc.

Unsigned byte-wise quad maximum

quadumax

SYNTAX

`[ IF rguard ] quadumax rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then {**
r*dest*<7:0> ← **if** r*src1*<7:0> > r*src2*<7:0> **then** r*src1*<7:0> **else** r*src2*<7:0>
r*dest*<15:8> ← **if** r*src1*<15:8> > r*src2*<15:8> **then** r*src1*<15:8> **else** r*src2*<15:8>
r*dest*<23:16> ← **if** r*src1*<23:16> > r*src2*<23:16> **then** r*src1*<23:16> **else** r*src2*<23:16>
r*dest*<31:24> ← **if** r*src1*<31:24> > r*src2*<31:24> **then** r*src1*<31:24> **else** r*src2*<31:24>
}

ATTRIBUTES

Function unit	dspalu
Operation code	81
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1,3

SEE ALSO

`imax` `imin` `quadumin`

DESCRIPTION

The `quadumax` operation computes four separate maximum values of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The `quadumax` operation is particularly suited to implement median computation on packed pixel data structures:

MEDIAN_Q(a,b,c) (QUADUMIN(QUADUMAX(QUADUMIN((a),(b)), (c)), QUADUMAX((a),(b))))

The `quadumax` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xff00ff02	`quadumax r30 r40 → r50`	r50 ← 0xff01ff0e
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r10 quadumax r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r20 quadumax r60 r70 → r90`	r90 ← 0x9c9d649c

quadumin — Unsigned bytewise quad minimum

SYNTAX

`[ IF rguard ] quadumin rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then {**
  r*dest*<7:0> ← **if** r*src1*<7:0> < r*src2*<7:0> **then** r*src1*<7:0> **else** r*src2*<7:0>
  r*dest*<15:8> ← **if** r*src1*<15:8> < r*src2*<15:8> **then** r*src1*<15:8> **else** r*src2*<15:8>
  r*dest*<23:16> ← **if** r*src1*<23:16> < r*src2*<23:16> **then** r*src1*<23:16> **else** r*src2*<23:16>
  r*dest*<31:24> ← **if** r*src1*<31:24> < r*src2*<31:24> **then** r*src1*<31:24> **else** r*src2*<31:24>
}

ATTRIBUTES

Function unit	dspalu
Operation code	80
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1,3

SEE ALSO

imin imax quadumax

DESCRIPTION

The `quadumin` operation computes four separate minimum values of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The `quadumin` operation is particularly suited to implement median computation on packed pixel data structures:

MEDIAN_Q(a,b,c) (QUADUMIN(QUADUMAX(QUADUMIN((a),(b)), (c)), QUADUMAX((a),(b))))

The `quadumin` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xff00ff02	`quadumin r30 r40 → r50`	r50 ← 0x02000002
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r10 quadumin r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649d649c	`IF r20 quadumin r60 r70 → r90`	r90 ← 0x649c6464

© TriMedia Technologies, Inc.

Unsigned quad 8-bit multiply most significant

quadumulmsb

SYNTAX

```
[ IF rguard ] quadumulmsb rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    temp ← (zero_ext8to32(rsrc1<7:0>) × zero_ext8to32(rsrc2<7:0>))
    rdest<7:0> ← temp<15:8>
    temp ← (zero_ext8to32(rsrc1<15:8>) × zero_ext8to32(rsrc2<15:8>))
    rdest<15:8> ← temp<15:8>
    temp ← (zero_ext8to32(rsrc1<23:16>) × zero_ext8to32(rsrc2<23:16>))
    rdest<23:16> ← temp<15:8>
    temp ← (zero_ext8to32(rsrc1<31:24>) × zero_ext8to32(rsrc2<31:24>))
    rdest<31:24> ← temp<15:8>
}
```

ATTRIBUTES

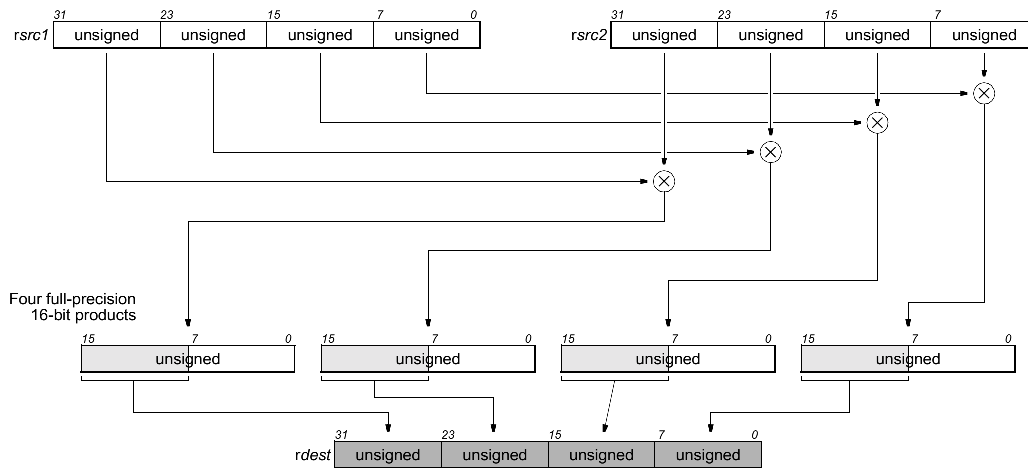
Function unit	dspmul
Operation code	89
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

quadavg dspuquadaddui ifir8ii

DESCRIPTION

As shown below, the `quadumulmsb` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*. All bytes are considered unsigned. The most-significant 8 bits of each 16-bit product is written to the corresponding byte in r*dest*.



The `quadumulmsb` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0210800e, r40 = 0xffffff02	`quadumulmsb r30 r40 → r50`	r50 ← 0x010f7f00
r10 = 0, r60 = 0x80ff1010, r70 = 0x80ff100f	`IF r10 quadumulmsb r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x80ff1010, r70 = 0x80ff100f	`IF r20 quadumulmsb r60 r70 → r90`	r90 ← 0x40fe0100

© TriMedia Technologies, Inc.

rdstatus

Read data cache status bits

SYNTAX

`[ IF rguard ] rdstatus(d) rsrc1 → rdest`

FUNCTION

```
if rguard then {
    set_addr ← rsrc1 + d

        /* set_addr<10:6> selects set */

    rdest<9:0> ← dcache_LRU_set(set_addr)
    rdest<17:10> ← dcache_dirty_set(set_addr)
    rdest<31:18> ← 0
}
```

ATTRIBUTES

Function unit	dmemspec
Operation code	203
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

SEE ALSO

`rdtag`

DESCRIPTION

The `rdstatus` operation reads the LRU and dirty bits associated with a set in the data cache and writes these bits into the destination register r*dest*. The target set in the data cache is determined by bits 10..6 of the result of r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

The result of `rdstatus` contains LRU information in bits 9..0 and dirty-bit information in bits 17..10. All other bits of r*dest* are set to zero.

`rdstatus` requires two stall cycles to complete.

The dual-ported data cache uses two separate copies of tag and status information. A `rdstatus` operation returns the LRU and dirty information stored in the cache port that corresponds to the operation slot in which the `rdstatus` operation is issued.

The `rdstatus` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`rdstatus(0) r30 → r60`	
r10 = 0	`IF r10 rdstatus(4) r40 → r70`	no change, since guard is false
r20 = 1	`IF r20 rdstatus(8) r50 → r80`	

© TriMedia Technologies, Inc.

Read data cache address tag

rdtag

SYNTAX

```
[ IF rguard ] rdtag(d) rsrc1 → rdest
```

ATTRIBUTES

Function unit	dmemspec
Operation code	202
Number of operands	1
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	3
Issue slots	5

FUNCTION

```
if rguard then {
    block_addr ← rsrc1 + d

        /* block_addr<13:11> selects element, block_addr<10:6> selects set */

    rdest<21:0> ← dcache_tag_block(block_addr)
    rdest<31:22> ← 0
}
```

SEE ALSO

`rdstatus`

DESCRIPTION

The `rdtag` operation reads the address tag associated with a block in the data cache and writes these bits into the destination register r*dest*. The target block in the data cache is determined by bits 13..6 of the result of r*src1* + *d*. Bits 10..6 of r*src1* + *d* select the cache set and 13..11 of r*src1* + *d* select the element within that set. The *d* value is an opcode modifier, must be in the range –256 to 252 inclusive, and must be a multiple of 4.

`rdtag` writes the address tag for the selected block in bits 21..0 of r*dest*. All other bits of r*dest* are set to zero.

`rdtag` requires no stall cycles to complete.

The dual-ported data cache uses two separate copies of tag and status information. A `rdtag` operation returns the address tag information stored in the cache port that corresponds to the operation slot in which the `rdtag` operation is issued.

The `rdtag` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
	`rdtag(0) r30 → r60`	
r10 = 0	`IF r10 rdtag(4) r40 → r70`	no change, since guard is false
r20 = 1	`IF r20 rdtag(8) r50 → r80`	

readdpc Read destination program counter

SYNTAX

```
[ IF rguard ] readdpc → rdest
```

FUNCTION

if *rguard* **then {**
rdest ← DPC
}

ATTRIBUTES

Function unit	fcomp
Operation code	156
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

writedpc readspc ijmpf ijmpi ijmpt

DESCRIPTION

The `readdpc` writes the current value of the DPC (Destination Program Counter) processor register to r*dest*.

Interruptible jumps write their target address to the DPC. If an interrupt or exception is taken at an interruptible jump, execution of the interrupted program can be resumed by jumping to the value contained in DPC. This operation can be used to save state before idling a task in a multi-tasking environment.

The `readdpc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
DPC = 0xbeebee	`readdpc → r100`	r100 ← 0xbeebee
r20 = 0, DPC = 0xabba	`IF r20 readdpc → r101`	no change, since guard is false
r21 = 1, DPC = 0xabba	`IF r21 readdpc → r102`	r102 ← 0xabba

 © TriMedia Technologies, Inc.

Read program control and status word

readpcsw

SYNTAX

```
[ IF rguard ] readpcsw → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← PCSW
}

ATTRIBUTES

Function unit	fcomp
Operation code	158
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

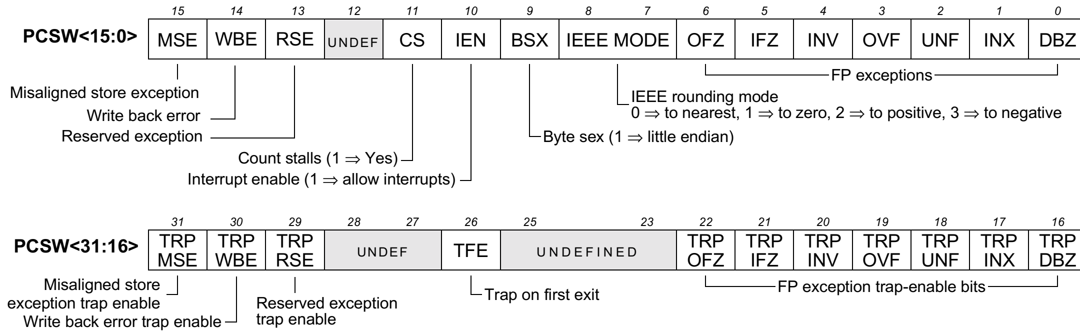
`writepcsw`

DESCRIPTION

The `readpcsw` writes the current value of the PCSW (Program Control and Status Word) processor register to r*dest*. The layout of PCSW is shown below.

Fields in the PCSW have two chief purposes: to control aspects of processor operation and to record events that occur during program execution. Thus, `readpcsw` can be used to determine current processor operating modes and what events have occurred; this operation can also be used to save state before idling a task in a multi-tasking environment.

The `readpcsw` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.



EXAMPLES

Initial Values	Operation	Result
PCSW = 0x80110642	`readpcsw → r100`	`r100` ← 0x80110642 (trap on MSE, INV and DBZ enabled, IEN=1 - interrupts enabled, BSX=1 - little endian mode of operation, OFZ=1 - a denormalized result was produced somewhere, INX=1 - an inexact result was produced somewhere)
r20 = 0, PCSW = 0x80000000	`IF r20 readpcsw → r101`	no change, since guard is false
r21 = 1, PCSW = 0x80000000	`IF r21 readpcsw → r102`	`r102` ← 0x80000000 (trap on MSE enabled)

readspc

Read source program counter

SYNTAX

```
[ IF rguard ] readspc → rdest
```

FUNCTION

if *rguard* **then {**
rdest ← SPC
}

ATTRIBUTES

Function unit	fcomp
Operation code	157
Number of operands	0
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

writespc readdpc ijmpf ijmpi ijmpt

DESCRIPTION

The `readspc` writes the current value of the SPC (Source Program Counter) processor register to r*dest*.

An interruptible jump that is not interrupted (no NMI, INT, or EXC event was pending when the jump was executed) writes its target address to SPC. The value of SPC allows an exception-handling routine to determine the start address of the block of scheduled code (called a decision tree) that was executing before the exception was taken.This operation can be used to save state before idling a task in a multi-tasking environment.

The `readspc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
SPC = 0xbeebee	`readspc → r100`	r100 ← 0xbeebee
r20 = 0, SPC = 0xabba	`IF r20 readspc → r101`	no change, since guard is false
r21 = 1, SPC = 0xabba	`IF r21 readspc → r102`	r102 ← 0xabba

© TriMedia Technologies, Inc.

Rotate left

rol

SYNTAX

```
[ IF rguard ] rol rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
n ← r*src2*<4:0>
r*dest*<31:n> ← r*src1*<31–n:0>
r*dest*<n–1:0> ← r*src1*<31:32–n>
}

ATTRIBUTES

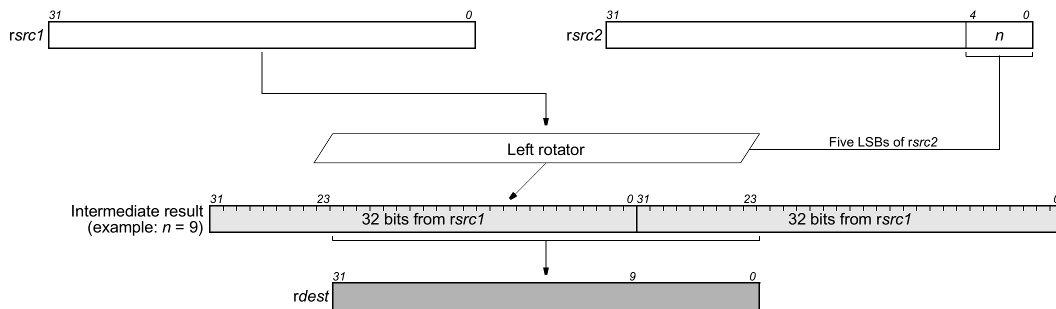
Function unit	shifter
Operation code	97
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

roli asr asri lsl lsli lsr lsri

DESCRIPTION

As shown below, the `rol` operation takes two arguments, r*src1* and r*src2*. The least-significant five bits of r*src2* specify an unsigned rotate amount, and r*dest* is set to r*src1* rotated left by this amount. The most-significant n bits of r*src1*, where n is the rotate amount, appear as the least-significant n bits in r*dest*.



The `rol` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20, r30 = 3	`rol r60 r30 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20, r30 = 3	`IF r10 rol r60 r30 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20, r30 = 3	`IF r20 rol r60 r30 → r110`	r110 ← 0x100
r70 = 0xfffffffc, r40 = 2	`rol r70 r40 → r120`	r120 ← 0xfffffff3
r80 = 0xe, r50 = 0xfffffffe	`rol r80 r50 → r125`	r125 ← 0x80000003 (r50 is effectively equal to 0x1e)

roli

Rotate left by immediate

SYNTAX

`[ IF rguard ] roli(n) rsrc1 → rdest`

FUNCTION

if *rguard* **then {**
  rdest<31:*n*> ← *rsrc1*<31–*n*:0>
  rdest<*n*–1:0> ← *rsrc1*<31:32–*n*>
}

ATTRIBUTES

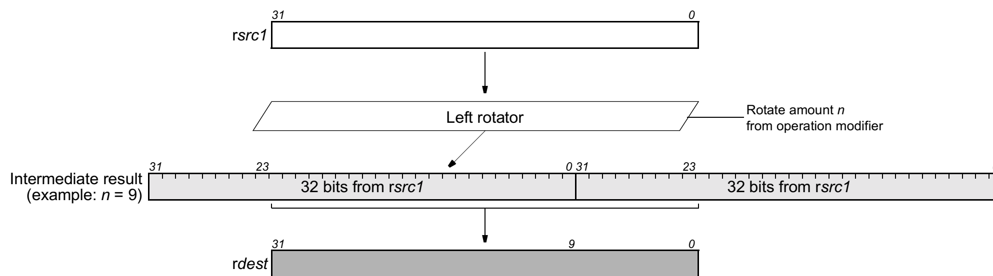
Function unit	shifter
Operation code	98
Number of operands	1
Modifier	7 bits
Modifier range	0..31
Latency	1
Issue slots	1, 2

SEE ALSO

rol asl asli asr asri lsl lsli lsr lsri

DESCRIPTION

As shown below, the `roli` operation takes a single argument in r*src1* and an immediate modifier *n* and produces a result in r*dest* equal to r*src1* rotated left by *n* bits. The value of *n* must be between 0 and 31, inclusive. The most-significant *n* bits of r*src1* appear as the least-significant *n* bits in r*dest*.



The `roli` operations optionally take a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x20	`roli(3) r60 → r90`	r90 ← 0x100
r10 = 0, r60 = 0x20	`IF r10 roli(3) r60 → r100`	no change, since guard is false
r20 = 1, r60 = 0x20	`IF r20 roli(3) r60 → r110`	r110 ← 0x100
r70 = 0xfffffffc	`roli(2) r70 → r120`	r120 ← 0xfffffff3
r80 = 0xe	`roli(30) r80 → r125`	r125 ← 0x80000003

 © TriMedia Technologies, Inc.

Sign extend 16 bits

sex16

SYNTAX

```
[ IF rguard ] sex16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← sign_ext16to32(r*src1*<15:0>)

ATTRIBUTES

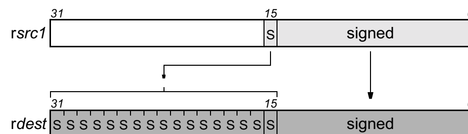
Function unit	alu
Operation code	51
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

zex16 sex8 zex8

DESCRIPTION

As shown below, the sex16 operation sign extends the least-significant 16bit halfword of the argument, r*src1*, to 32 bits and stores the result in r*dest*.



The sex16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of the guard is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	sex16 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 sex16 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 sex16 r40 → r100	r100 ← 0xffffff91
r50 = 0x00000091	sex16 r50 → r110	r110 ← 0x00000091

sex8

Sign extend 8 bits
pseudo-op for ibytesel

SYNTAX

```
[ IF rguard ] sex8 rsrc1 → rdest
```

ATTRIBUTES

Function unit	alu
Operation code	56
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

FUNCTION

if r*guard* **then**
 r*dest* ← sign_ext8to32(r*src1*<7:0>)

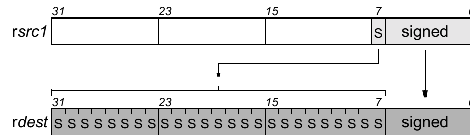
SEE ALSO

ibytesel sex16 zex8 zex16

DESCRIPTION

The sex8 operation is a pseudo operation transformed by the scheduler into a ibytesel with r*src1* as the first argument and r0 (always contains 0) as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the sex8 operation sign extends the least-significant halfword of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The sex8 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	sex8 r30 → r60	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	IF r10 sex8 r40 → r70	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	IF r20 sex8 r40 → r100	r100 ← 0xffffff91
r50 = 0x00000091	sex8 r50 → r110	r110 ← 0xffffff91

 © TriMedia Technologies, Inc.

16-bit store
pseudo-op for h_st16d(0)

st16

SYNTAX

```
[ IF rguard ] st16 rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 1
  else
    bs ← 0
  mem[r*src1* + (1 ⊕ bs)] ← r*src2*<7:0>
  mem[r*src1* + (0 ⊕ bs)] ← r*src2*<15:8>
}

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16d h_st16d st8 st8d st32 st32d

DESCRIPTION

The `st16` operation is a pseudo operation transformed by the scheduler into an `h_st16d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st16` operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1*. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st16` is misaligned (the memory address in r*src1* is not a multiple of 2), the result of `st16` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by `st16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st16 r10 r80`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st16 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st16 r30 r70`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

st16d

16-bit store with displacement

pseudo-op for h_st16d

SYNTAX

```
[ IF rguard ] st16d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    mem[rsrc1 + d + (1 ⊕ bs)] ← rsrc2<7:0>
    mem[rsrc1 + d + (0 ⊕ bs)] ← rsrc2<15:8>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	30
Number of operands	2
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	n/a
Issue slots	4, 5

SEE ALSO

st16 h_st16d st8 st8d st32 st32d

DESCRIPTION

The `st16d` operation is a pseudo operation transformed by the scheduler into an `h_st16d` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st16d` operation stores the least-significant 16-bit halfword of r*src2* into the memory locations pointed to by the address in r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st16d` is misaligned (the memory address computed by r*src1* + *d* is not a multiple of 2), the result of `st16d` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The result of an access by `st16d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st16d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfe, r80 = 0x44332211	`st16d(2) r10 r80`	[0xd00] ← 0x22, [0xd01] ← 0x11
r50 = 0, r20 = 0xd05, r70 = 0xaabbccdd	`IF r50 st16d(-4) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd06, r70 = 0xaabbccdd	`IF r60 st16d(-4) r30 r70`	[0xd02] ← 0xcc, [0xd03] ← 0xdd

© TriMedia Technologies, Inc.

32-bit store
pseudo-op for h_st32d(0)

st32

SYNTAX

```
[ IF rguard ] st32 rsrc1 rsrc2
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 3
  else
    bs ← 0
  mem[r*src1* + (3 ⊕ bs)] ← r*src2*<7:0>
  mem[r*src1* + (2 ⊕ bs)] ← r*src2*<15:8>
  mem[r*src1* + (1 ⊕ bs)] ← r*src2*<23:16>
  mem[r*src1* + (0 ⊕ bs)] ← r*src2*<31:24>
}

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st32d st32d st16 st16d st8 st8d

DESCRIPTION

The `st32` operation is a pseudo operation transformed by the scheduler into an `h_st32d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st32` operation stores all 32 bits of r*src2* into the memory locations pointed to by the address in r*src1*. The *d* value is an opcode modifier and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st32` is misaligned (the memory address in r*src1* is not a multiple of 4), the result of `st32` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The `st32` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `st32`.

The `st32` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st32 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st32 r10 r80`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st32 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd04, r70 = 0xaabbccdd	`IF r60 st32 r30 r70`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

st32d

32-bit store with displacement

pseudo-op for h_st32d

SYNTAX

```
[ IF rguard ] st32d(d) rsrc1 rsrc2
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 3
    else
        bs ← 0
    mem[rsrc1 + d + (3 ⊕ bs)] ← rsrc2<7:0>
    mem[rsrc1 + d + (2 ⊕ bs)] ← rsrc2<15:8>
    mem[rsrc1 + d + (1 ⊕ bs)] ← rsrc2<23:16>
    mem[rsrc1 + d + (0 ⊕ bs)] ← rsrc2<31:24>
}
```

ATTRIBUTES

Function unit	dmem
Operation code	31
Number of operands	2
Modifier	7 bits
Modifier range	–256..252 by 4
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st32d st32 st16 st16d st8 st8d

DESCRIPTION

The `st32d` operation is a pseudo operation transformed by the scheduler into an `h_st32d` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st32d` operation stores all 32 bits of r*src2* into the memory locations pointed to by the address in r*src1* + *d*. The *d* value is an opcode modifier, must be in the range –256 and 252 inclusive, and must be a multiple of 4. This store operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

If `st32d` is misaligned (the memory address computed by r*src1* + *d* is not a multiple of 4), the result of `st32d` is undefined, and the MSE (Misaligned Store Exception) bit in the PCSW register is set to 1. Additionally, if the TRPMSE (TRaP on Misaligned Store Exception) bit in PCSW is 1, exception processing will be requested on the next interruptible jump.

The `st32d` operation can be used to access the MMIO address aperture (the result of MMIO access by 8- or 16-bit memory operations is undefined). The state of the BSX bit in the PCSW has no effect on MMIO access by `st32d`.

The `st32d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory locations (and the modification of cache if the locations are cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st32d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xcfc, r80 = 0x44332211	`st32d(4) r10 r80`	[0xd00] ← 0x44, [0xd01] ← 0x33, [0xd02] ← 0x22, [0xd03] ← 0x11
r50 = 0, r20 = 0xd0b, r70 = 0xaabbccdd	`IF r50 st32d(-8) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd0c, r70 = 0xaabbccdd	`IF r60 st32d(-8) r30 r70`	[0xd04] ← 0xaa, [0xd05] ← 0xbb, [0xd06] ← 0xcc, [0xd07] ← 0xdd

© TriMedia Technologies, Inc.

8-bit store
pseudo-op for h_st8d(0)

st8

SYNTAX

```
[ IF rguard ] st8 rsrc1 rsrc2
```

FUNCTION

if r*guard* **then**
mem[r*src1*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	No
Modifier range	—
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st8d st8d st16 st16d st32 st32d

DESCRIPTION

The `st8` operation is a pseudo operation transformed by the scheduler into an `h_st8d(0)` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st8` operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address in r*src1*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by `st8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8 has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st8 r10 r80`	[0xd00] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st8 r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st8 r30 r70`	[0xd02] ← 0xdd

st8d

8-bit store with displacement

pseudo-op for h_st8d

SYNTAX

```
[ IF rguard ] st8d(d) rsrc1 rsrc2
```

FUNCTION

if *rguard* **then**
mem[r*src1* + *d*] ← r*src2*<7:0>

ATTRIBUTES

Function unit	dmem
Operation code	29
Number of operands	2
Modifier	7 bits
Modifier range	–64..63
Latency	n/a
Issue slots	4, 5

SEE ALSO

h_st8d st8 st16 st16d st32 st32d

DESCRIPTION

The `st8d` operation is a pseudo operation transformed by the scheduler into an `h_st8d` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `st8d` operation stores the least-significant 8-bit byte of r*src2* into the memory location pointed to by the address formed from the sum r*src1* + *d*. The value of the opcode modifier *d* must be in the range -64 and 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is stored.

The result of an access by `st8d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `st8d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the addressed memory location (and the modification of cache if the location is cacheable). If the LSB of r*guard* is 1, the store takes effect. If the LSB of r*guard* is 0, st8d has no side effects whatever; in particular, the LRU and other status bits in the data cache are not affected.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r80 = 0x44332211	`st8d(3) r10 r80`	[0xd03] ← 0x11
r50 = 0, r20 = 0xd01, r70 = 0xaabbccdd	`IF r50 st8d(-4) r20 r70`	no change, since guard is false
r60 = 1, r30 = 0xd02, r70 = 0xaabbccdd	`IF r60 st8d(-4) r30 r70`	[0xcfe] ← 0xdd

© TriMedia Technologies, Inc.

Select unsigned byte

ubytesel

SYNTAX

```
[ IF rguard ] ubytesel rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc2 = 0 then
        rdest ← zero_ext8to32(rsrc1<7:0>)
    else if rsrc2 = 1 then
        rdest ← zero_ext8to32(rsrc1<15:8>)
    else if rsrc2 = 2 then
        rdest ← zero_ext8to32(rsrc1<23:15>)
    else if rsrc2 = 3 then
        rdest ← zero_ext8to32(rsrc1<31:24>)
}
```

ATTRIBUTES

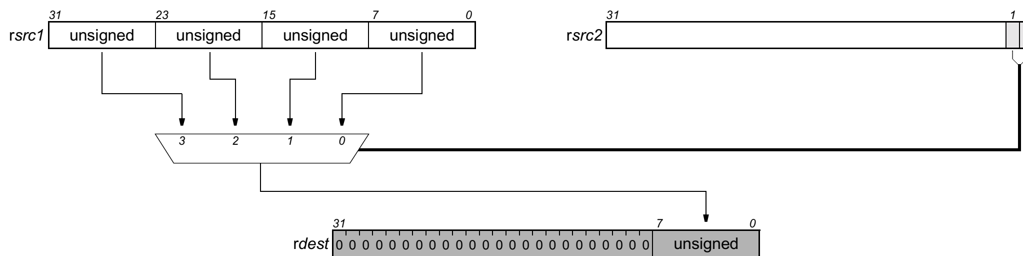
Function unit	alu
Operation code	55
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ibytesel sex8 packbytes

DESCRIPTION

As shown below, the ubytesel operation selects one byte from the argument, r*src1*, zero-extends the byte to 32 bits, and stores the result in r*dest*. The value of r*src2* determines which byte is selected, with r*src2*=0 selecting the LSB of r*src1* and r*src2*=3 selecting the MSB of r*src1*. If rsrc2 is not between 0 and 3 inclusive, the result of ubytesel is undefined.



The ubytesel operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x44332211, r40 = 1	`ubytesel r30 r40 → r50`	r50 ← 0x00000022
r10 = 0, r60 = 0xddccbbaa, r70 = 2	`IF r10 ubytesel r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0xddccbbaa, r70 = 2	`IF r20 ubytesel r60 r70 → r90`	r90 ← 0x000000cc
r100 = 0xffffff7f, r110 = 0	`ubytesel r100 r110 → r120`	r120 ← 0x0000007f

uclipi

Clip signed to unsigned

SYNTAX

`[ IF rguard ] uclipi rsrc1 rsrc2 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← min(max(r*src1*, 0), r*src2*)

ATTRIBUTES

Function unit	dspalu
Operation code	75
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

`iclipi uclipu imin imax`

DESCRIPTION

The `uclipi` operation returns the value of r*src1* clipped into the unsigned integer range 0 to r*src2*, inclusive. The argument r*src1* is considered a signed integer; r*src2* is considered an unsigned integer.

The `uclipi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`uclipi r100 r110 → r120`	r120 ← 0

 © TriMedia Technologies, Inc.

Clip unsigned to unsigned

uclipu

SYNTAX

```
[ IF rguard ] uclipu rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	76
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipi imin imax

DESCRIPTION

The `uclipu` operation returns the value of r*src1* clipped into the unsigned integer range 0 to r*src2*, inclusive. The arguments r*src1* and r*src2* are considered unsigned integers.

The `uclipu` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipu r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipu r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipu r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	`uclipu r100 r110 → r120`	r120 ← 0x3fffff

ueql

Unsigned compare equal

pseudo-op for ieql

SYNTAX

```
[ IF rguard ] ueql rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if r*src1* = r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	37
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`ieql` `ueqli` `igeq` `uneq`

DESCRIPTION

The `ueql` operation is a pseudo operation transformed by the scheduler into an `ieql` with the same arguments. (Note: pseudo operations cannot be used in assembly files.)

The `ueql` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueql` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ueql r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ueql r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	`IF r20 ueql r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ueql r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ueql r70 r70 → r110`	r110 ← 1

 © TriMedia Technologies, Inc.

Unsigned compare equal with immediate

ueqli

SYNTAX

```
[ IF rguard ] ueqli(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 = n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	38
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ieqli ueql igeqi uneqi

DESCRIPTION

The `ueqli` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ueqli` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ueqli(2) r30 → r80`	r80 ← 0
r30 = 3	`ueqli(3) r30 → r90`	r90 ← 1
r30 = 3	`ueqli(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ueqli(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ueqli(63) r40 → r100`	r100 ← 0
r60 = 0x07f	`ueqli(127) r60 → r120`	r120 ← 1

ufir16 Sum of products of unsigned 16-bit halfwords

SYNTAX

```
[ IF rguard ] ufir16 rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← zero_ext16to32(r*src1*<31:16>) × zero_ext16to32(r*src2*<31:16>) +
zero_ext16to32(r*src1*<15:0>) × zero_ext16to32(r*src2*<15:0>)

ATTRIBUTES

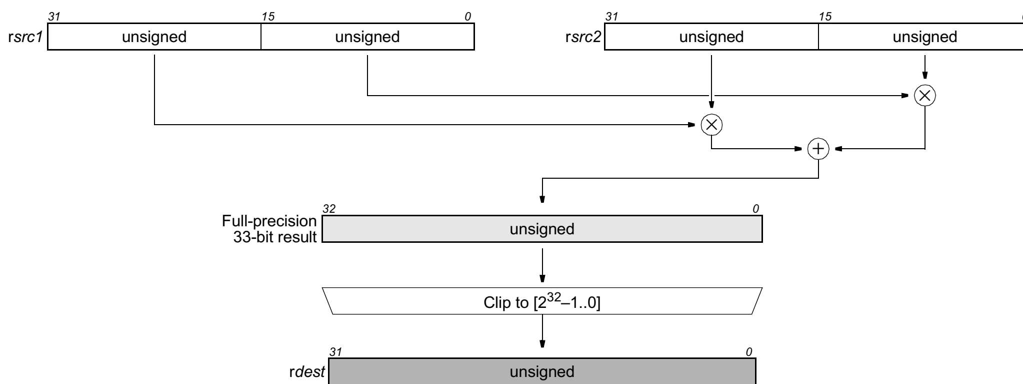
Attribute	Value
Function unit	dspmul
Operation code	94
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir16 ifir8ii ifir8ui ufir8uu

DESCRIPTION

As shown below, the `ufir16` operation computes two separate products of the two pairs of corresponding 16-bit halfwords of r*src1* and r*src2*; the two products are summed, and the result is written to r*dest*. All halfwords are considered unsigned; thus, the intermediate products and the final sum of products are unsigned. All intermediate computations are performed without loss of precision; the final sum of products is clipped into the range [0xffffffff..0] before being written into r*dest*.



The `ufir16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x00020003, r40 = 0x00010002	`ufir16 r30 r40 → r50`	r50 ← 8
r10 = 0, r60 = 0x80000064, r70 = 0x00648000	`IF r10 ufir16 r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x80000064, r70 = 0x00648000	`IF r20 ufir16 r60 r70 → r90`	r90 ← 0x00640000
r30 = 0x00020003, r70 = 0x00648000	`ufir16 r30 r70 → r100`	r100 ← 0x000180c8

© TriMedia Technologies, Inc.

Unsigned sum of products of unsigned bytes

ufir8uu

SYNTAX

```
[ IF rguard ] ufir8uu rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext8to32(r*src1*<31:24>) × zero_ext8to32(r*src2*<31:24>) +
zero_ext8to32(r*src1*<23:16>) × zero_ext8to32(r*src2*<23:16>) +
zero_ext8to32(r*src1*<15:8>) × zero_ext8to32(r*src2*<15:8>) +
zero_ext8to32(r*src1*<7:0>) × zero_ext8to32(r*src2*<7:0>)

ATTRIBUTES

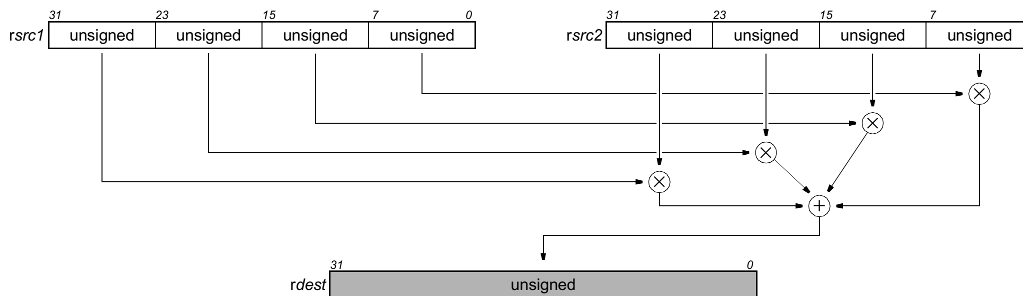
Function unit	dspmul
Operation code	90
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ui ifir8ii ifir16 ufir16

DESCRIPTION

As shown below, the `ufir8uu` operation computes four separate products of the four pairs of corresponding 8-bit bytes of r*src1* and r*src2*; the four products are summed, and the result is written to r*dest*. All values are considered unsigned. All computations are performed without loss of precision.



The `ufir8uu` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ufir8uu r70 r30 → r90`	r90 ← 0x1efa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ufir8uu r70 r30 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ufir8uu r80 r40 → r110`	r110 ← 0xf3c0
r50 = 0x80808080, r60 = 0xffffffff	`ufir8uu r50 r60 → r120`	r120 ← 0x1fe00

ufixieee Convert floating-point to unsigned integer using PCSW rounding mode

SYNTAX

```
[ IF rguard ] ufixieee rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
 r*dest* ← (unsigned long) ((float)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	123
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ifixrz ufixrz

DESCRIPTION

The ufixieee operation converts the single-precision IEEE floating-point value in r*src1* to an unsigned integer and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If ufixieee causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit writepcsw operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The ufixieeeflags operation computes the exception flags that would result from an individual ufixieee.

The ufixieee operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	ufixieee r30 → r100	r100 ← 3
r35 = 0x40247ae1 (2.57)	ufixieee r35 → r102	r102 ← 3, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	IF r10 ufixieee r40 → r105	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	IF r20 ufixieee r40 → r110	r110 ← 0x0, INV flag set
r45 = 0x7f800000 (+INF))	ufixieee r45 → r112	r112 ← 0xffffffff (2^{32}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	ufixieee r50 → r115	r115 ← 0, INV flag set
r60 = 0x00400000 (5.877471754e-39)	ufixieee r60 → r117	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	ufixieee r70 → r120	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	ufixieee r80 → r122	r122 ← 0, INV flag set

 © TriMedia Technologies, Inc.

IEEE status flags from convert floating-point to unsigned integer using PCSW rounding mode

ufixieeeflags

SYNTAX

```
[ IF rguard ] ufixieeeflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((unsigned long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	124
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

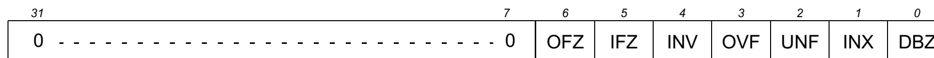
SEE ALSO

ufixieee ifixieeeflags ifixrzflags ufixrzflags

DESCRIPTION

The `ufixieeeflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to an unsigned integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW. If an argument is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ufixieeeflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixieeeflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ufixieeeflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixieeeflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixieeeflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ufixieeeflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ufixieeeflags r50 → r115`	r115 ← 0x10 (INV)
r60 = 0x00400000 (5.877471754e-39)	`ufixieeeflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ufixieeeflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ufixieeeflags r80 → r122`	r122 ← 0x10 (INV)

ufixrz

Convert floating-point to unsigned integer with round toward zero

SYNTAX

```
[ IF rguard ] ufixrz rsrc1 → rdest
```

FUNCTION

if *rguard* **then {**
  rdest ← (unsigned long) ((float)*rsrc1*)
}

ATTRIBUTES

Function unit	falu
Operation code	125
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifixieee ufixieee ifixrz

DESCRIPTION

The `ufixrz` operation converts the single-precision IEEE floating-point value in r*src1* to an unsigned integer and writes the result into r*dest*. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If r*src1* is denormalized, zero is substituted before conversion, and the IFZ flag in the PCSW is set. If `ufixrz` causes an IEEE exception, such as overflow or underflow, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufixrzflags` operation computes the exception flags that would result from an individual `ufixrz`.

The `ufixrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixrz r30 → r100`	r100 ← 3
r35 = 0x40247ae1 (2.57)	`ufixrz r35 → r102`	r102 ← 2, INX flag set
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixrz r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixrz r40 → r110`	r110 ← 0x0, INV flag set
r45 = 0x7f800000 (+INF))	`ufixrz r45 → r112`	r112 ← 0xffffffff (2^{32}-1), INV flag set
r50 = 0xbfc147ae (-1.51)	`ufixrz r50 → r115`	r115 ← 0, INV flag set
r60 = 0x00400000 (5.877471754e-39)	`ufixrz r60 → r117`	r117 ← 0, IFZ set
r70 = 0xffffffff (QNaN)	`ufixrz r70 → r120`	r120 ← 0, INV flag set
r80 = 0xffbfffff (SNaN)	`ufixrz r80 → r122`	r122 ← 0, INV flag set

 © TriMedia Technologies, Inc.

IEEE status flags from convert floating-point to unsigned integer with round toward zero

ufixrzflags

SYNTAX

```
[ IF rguard ] ufixrzflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← ieee_flags((unsigned long) ((float)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	126
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

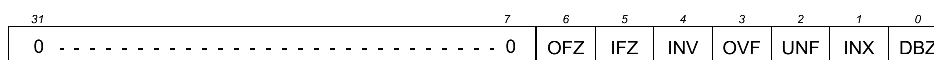
SEE ALSO

ufixrz ifixrzflags ifixieeeflags ufixieeeflags

DESCRIPTION

The `ufixrzflags` operation computes the IEEE exceptions that would result from converting the single-precision IEEE floating-point value in r*src1* to an unsigned integer, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding toward zero is performed; the IEEE rounding mode bits in PCSW are ignored. If an argument is denormalized, zero is substituted before computing the conversion, and the IFZ bit in the result is set.

The `ufixrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x40400000 (3.0)	`ufixrzflags r30 → r100`	r100 ← 0
r35 = 0x40247ae1 (2.57)	`ufixrzflags r35 → r102`	r102 ← 0x02 (INX)
r10 = 0, r40 = 0xff4fffff (–3.402823466e+38)	`IF r10 ufixrzflags r40 → r105`	no change, since guard is false
r20 = 1, r40 = 0xff4fffff (–3.402823466e+38)	`IF r20 ufixrzflags r40 → r110`	r110 ← 0x10 (INV)
r45 = 0x7f800000 (+INF))	`ufixrzflags r45 → r112`	r112 ← 0x10 (INV)
r50 = 0xbfc147ae (-1.51)	`ufixrzflags r50 → r115`	r115 ← 0x10 (INV)
r60 = 0x00400000 (5.877471754e-39)	`ufixrzflags r60 → r117`	r117 ← 0x20 (IFZ)
r70 = 0xffffffff (QNaN)	`ufixrzflags r70 → r120`	r120 ← 0x10 (INV)
r80 = 0xffbfffff (SNaN)	`ufixrzflags r80 → r122`	r122 ← 0x10 (INV)

ufloat

Convert unsigned integer to floating-point

SYNTAX

```
[ IF rguard ] ufloat rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
r*dest* ← (float) ((unsigned long)r*src1*)
}

ATTRIBUTES

Function unit	falu
Operation code	127
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`ifloat` `ifloatrz` `ufloatrz` `ifixieee` `ufloatflags`

DESCRIPTION

The `ufloat` operation converts the unsigned integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is according to the IEEE rounding mode bits in PCSW. If `ufloat` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatflags` operation computes the exception flags that would result from an individual `ufloat`.

The `ufloat` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloat r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloat r40 → r105`	r105 ← 0x4f800000 (4.294967296e+9), INX flag set
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloat r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloat r50 → r115`	r115 ← 0x4f800000 (4.294967296e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloat r60 → r117`	r117 ← 0x4f000000 (2.147483648e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloat r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ufloat r80 → r122`	r122 ← 0x4f000000 (2.147483648e+9), INX flag set

© TriMedia Technologies, Inc.

IEEE status flags from convert unsigned integer to floating-point

ufloatflags

SYNTAX

```
[ IF rguard ] ufloatflags rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float) ((unsigned long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	128
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

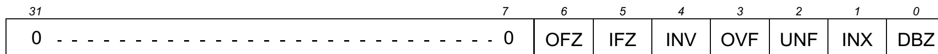
SEE ALSO

ufloat ifloatflags ifloatrzflags ufloatrzflags

DESCRIPTION

The `ufloatflags` operation computes the IEEE exceptions that would result from converting the unsigned integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is according to the IEEE rounding mode bits in PCSW.

The `ufloatflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (4294967295)	`ufloatflags r40 → r105`	r105 ← 0x02 (INX)
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x7fffffff (2147483647)	`ufloatflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (2147483648)	`ufloatflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ufloatflags r80 → r122`	r122 ← 0x02 (INX)

ufloatrz

Convert unsigned integer to floating-point with rounding toward zero

SYNTAX

`[ IF rguard ] ufloatrz rsrc1 → rdest`

FUNCTION

if *rguard* **then {**
rdest ← (float) ((unsigned long)*rsrc1*)
}

ATTRIBUTES

Function unit	falu
Operation code	119
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

`ifloatrz` `ifloat` `ufloat` `ifixieee` `ufloatflags`

DESCRIPTION

The `ufloatrz` operation converts the unsigned integer value in r*src1* to single-precision IEEE floating-point format and writes the result into r*dest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ufloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as r*dest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatrzflags` operation computes the exception flags that would result from an individual `ufloatrz`.

The `ufloatrz` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* and the exception flags in PCSW are written; otherwise, r*dest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloatrz r40 → r105`	r105 ← 0x4f7fffff (4.294967040e+9), INX flag set
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatrz r50 → r115`	r115 ← 0x4f7fffff (4.294967040e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloatrz r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7ffffff1 (2147483633)	`ufloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set

© TriMedia Technologies, Inc.

IEEE status flags from convert unsigned integer to floating-point with rounding toward zero

ufloatrzflags

SYNTAX

`[ IF rguard ] ufloatrzflags rsrc1 → rdest`

FUNCTION

if r*guard* **then**
r*dest* ← ieee_flags((float) ((unsigned long)r*src1*))

ATTRIBUTES

Function unit	falu
Operation code	120
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

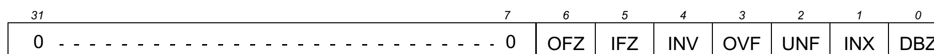
SEE ALSO

ufloatrz ifloatflags
ufloatflags ifloatrzflags

DESCRIPTION

The `ufloatrzflags` operation computes the IEEE exceptions that would result from converting the unsigned integer in r*src1* to a single-precision IEEE floating-point value, and an integer bit vector representing the computed exception flags is written into r*dest*. The bit vector stored in r*dest* has the same format as the IEEE exception bits in the PCSW. The exception flags in PCSW are left unchanged by this operation. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored.

The `ufloatrzflags` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.



EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrzflags r30 → r100`	r100 ← 0
r40 = 0xffffffff (4294967295)	`ufloatrzflags r40 → r105`	r105 ← 0x02 (INX)
r10 = 0, r50 = 0xfffffffd	`IF r10 ufloatrzflags r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xfffffffd	`IF r20 ufloatrzflags r50 → r115`	r115 ← 0x02 (INX)
r60 = 0x7fffffff (2147483647)	`ufloatrzflags r60 → r117`	r117 ← 0x02 (INX)
r70 = 0x80000000 (2147483648)	`ufloatrzflags r70 → r120`	r120 ← 0
r80 = 0x7ffffff1 (2147483633)	`ufloatrzflags r80 → r122`	r122 ← 0x02 (INX)

ugeq

Unsigned compare greater or equal

SYNTAX

```
[ IF rguard ] ugeq rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (unsigned)r*src1* >= (unsigned)r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	35
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igeq ugeqi

DESCRIPTION

The ugeq operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The ugeq operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	ugeq r30 r40 → r80	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	IF r10 ugeq r60 r30 → r50	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	IF r20 ugeq r50 r60 → r90	r90 ← 1
r70 = 0x80000000, r40 = 4	ugeq r70 r40 → r100	r100 ← 1
r70 = 0x80000000	ugeq r70 r70 → r110	r110 ← 1

 © TriMedia Technologies, Inc.

Unsigned compare greater or equal with immediate

ugeqi

SYNTAX

```
[ IF rguard ] ugeqi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
if (unsigned)r*src1* >= (unsigned)*n* **then**
r*dest* ← 1
else
r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	36
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ugeq igeqi

DESCRIPTION

The `ugeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugeqi(2) r30 → r80`	r80 ← 1
r30 = 3	`ugeqi(3) r30 → r90`	r90 ← 1
r30 = 3	`ugeqi(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugeqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugeqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugeqi(127) r60 → r120`	r120 ← 1

ugtr

Unsigned compare greater

SYNTAX

```
[ IF rguard ] ugtr rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (unsigned)r*src1* > (unsigned)r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtr ugtri

DESCRIPTION

The `ugtr` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugtr` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ugtr r30 r40 → r80`	r80 ← 0
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ugtr r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ugtr r50 r60 → r90`	r90 ← 1
r70 = 0x80000000, r40 = 4	`ugtr r70 r40 → r100`	r100 ← 1
r70 = 0x80000000	`ugtr r70 r70 → r110`	r110 ← 0

 © TriMedia Technologies, Inc.

Unsigned compare greater with immediate

ugtri

SYNTAX

```
[ IF rguard ] ugtri(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if (unsigned)r*src1* > (unsigned)*n* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	34
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

igtri ugtr

DESCRIPTION

The `ugeqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is greater than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ugeqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ugtri(2) r30 → r80`	r80 ← 1
r30 = 3	`ugtri(3) r30 → r90`	r90 ← 0
r30 = 3	`ugtri(4) r30 → r100`	r100 ← 0
r10 = 0, r40 = 0x100	`IF r10 ugtri(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ugtri(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`ugtri(127) r60 → r120`	r120 ← 1

uimm

Unsigned immediate

SYNTAX

`uimm(n) → rdest`

FUNCTION

rdest ← *n*

ATTRIBUTES

Function unit	const
Operation code	191
Number of operands	0
Modifier	32 bits
Modifier range	0..0xffffffff
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

iimm

DESCRIPTION

The `uimm` operation writes the unsigned 32-bit opcode modifier *n* into r*dest*. Note: this operation is not guarded.

EXAMPLES

Initial Values	Operation	Result
	`uimm(2) → r10`	r10 ← 2
	`uimm(0x100) → r20`	r20 ← 0x100
	`uimm(0xfffc0000) → r30`	r30 ← 0xfffc0000

© TriMedia Technologies, Inc.

Unsigned 16-bit load

pseudo-op for uld16d(0)

SYNTAX

```
[ IF rguard ] uld16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 1
  else
    bs ← 0
  temp<7:0> ← mem[r*src1* + (1 ⊕ bs)]
  temp<15:8> ← mem[r*src1* + (0 ⊕ bs)]
  r*dest* ← zero_ext16to32(temp<15:0>)
}

ATTRIBUTES

Function unit	dmem
Operation code	197
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16d ild16 ild16d uld16r ild16r uld16x ild16x

DESCRIPTION

The `uld16` operation is a pseudo operation transformed by the scheduler into an `uld16d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `uld16` operation loads the 16-bit memory value from the address contained in r*src1*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address contained in r*src1* is not a multiple of 2, the result of `uld16` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22, [0xd01] = 0x11	`uld16 r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r30 uld16 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84, [0xd05] = 0x33	`IF r40 uld16 r20 → r80`	r80 ← 0x00008433
r50 = 0xd01	`uld16 r50 → r90`	r90 undefined (0xd01 is not a multiple of 2)

uld16d

Unsigned 16-bit load with displacement

SYNTAX

```
[ IF rguard ] uld16d(d) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[rsrc1 + d + (1 ⊕ bs)]
    temp<15:8> ← mem[rsrc1 + d + (0 ⊕ bs)]
    rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	197
Number of operands	1
Modifier	7 bits
Modifier range	–128..126 by 2
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 ild16d uld16r ild16r uld16x ild16x

DESCRIPTION

The `uld16d` operation loads the 16-bit memory value from the address computed by r*src1* + *d*, zero extends it to 32 bits, and writes the result in r*dest*. The *d* value is an opcode modifier, must be in the range –128 and 126 inclusive, and must be a multiple of 2. If the memory address computed by r*src1* + *d* is not a multiple of 2, the result of `uld16d` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22, [0xd03] = 0x11	`uld16d(2) r10 → r60`	r60 ← 0x00002211
r30 = 0, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r30 uld16d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84, [0xd01] = 0x33	`IF r40 uld16d(-4) r20 → r80`	r80 ← 0x00008433
r50 = 0xd01	`uld16d(-4) r50 → r90`	r90 undefined (0xd01 +(–4) is not a multiple of 2)

 © TriMedia Technologies, Inc.

Unsigned 16-bit load with index

uld16r

SYNTAX

```
[ IF rguard ] uld16r rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if PCSW.bytesex = LITTLE_ENDIAN then
        bs ← 1
    else
        bs ← 0
    temp<7:0> ← mem[rsrc1 + rsrc2 + (1 ⊕ bs)]
    temp<15:8> ← mem[rsrc1 + rsrc2 + (0 ⊕ bs)]
    rdest ← zero_ext16to32(temp<15:0>)
}
```

ATTRIBUTES

Function unit	dmem
Operation code	198
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

`uld16` `ild16` `uld16d` `ild16d` `ild16r` `uld16x` `ild16x`

DESCRIPTION

The `uld16r` operation loads the 16-bit memory value from the address computed by r*src1* + r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address computed by r*src1* + r*src2* is not a multiple of 2, the result of `uld16r` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22, [0xd03] = 0x11	`uld16r r10 r20 → r80`	r80 ← 0x00002211
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 uld16r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 uld16r r40 r30 → r100`	r100 ← 0x00008433
r70 = 0xd01, r30 = 0xfffffffc	`uld16r r70 r30 → r110`	r110 undefined (0xd01 +(–4) is not a multiple of 2)

uld16x

Unsigned 16-bit load with scaled index

SYNTAX

```
[ IF rguard ] uld16x rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if PCSW.bytesex = LITTLE_ENDIAN **then**
    bs ← 1
  else
    bs ← 0
  temp<7:0> ← mem[r*src1* + (2 × r*src2*) + (1 ⊕ bs)]
  temp<15:8> ← mem[r*src1* + (2 × r*src2*) + (0 ⊕ bs)]
  r*dest* ← zero_ext16to32(temp<15:0>)
}

ATTRIBUTES

Function unit	dmem
Operation code	199
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld16 ild16 uld16d ild16d uld16r ild16r ild16x

DESCRIPTION

The `uld16x` operation loads the 16-bit memory value from the address computed by r*src1* + 2×r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. If the memory address computed by r*src1* + 2×r*src2* is not a multiple of 2, the result of `uld16x` is undefined but no exception will be raised. This load operation is performed as little-endian or big-endian depending on the current setting of the bytesex bit in the PCSW.

The result of an access by `uld16x` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld16x` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed locations are cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld16x` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r30 = 1, [0xd02] = 0x22, [0xd03] = 0x11	`uld16x r10 r30 → r100`	r100 ← 0x00002211
r50 = 0, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r50 uld16x r40 r20 → r80`	no change, since guard is false
r60 = 1, r40 = 0xd04, r20 = 0xfffffffe, [0xd00] = 0x84, [0xd01] = 0x33	`IF r60 uld16x r40 r20 → r90`	r90 ← 0x00008433
r70 = 0xd01, r30 = 1	`uld16x r70 r30 → r110`	r110 undefined (0xd01 + 2×1 is not a multiple of 2)

 © TriMedia Technologies, Inc.

Unsigned 8-bit load

pseudo-op for uld8d(0)

uld8

SYNTAX

```
[ IF rguard ] uld8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(mem[r*src1*])

ATTRIBUTES

Function unit	dmem
Operation code	8
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

ild8 uld8d ild8d uld8r ild8r

DESCRIPTION

The `uld8` operation is a pseudo operation transformed by the scheduler into an `uld8d(0)` with the same argument. (Note: pseudo operations cannot be used in assembly source files.)

The `uld8` operation loads the 8-bit memory value from the address contained in r*src1*, zero extends it to 32 bits, and writes the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd00] = 0x22	`uld8 r10 → r60`	r60 ← 0x00000022
r30 = 0, r20 = 0xd04, [0xd04] = 0x84	`IF r30 uld8 r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd04] = 0x84	`IF r40 uld8 r20 → r80`	r80 ← 0x00000084
r50 = 0xd01, [0xd01] = 0x33	`uld8 r50 → r90`	r90 ← 0x00000033

uld8d Unsigned 8-bit load with displacement

SYNTAX

`[ IF rguard ] uld8d(d) rsrc1 → rdest`

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(mem[r*src1* + *d*])

ATTRIBUTES

Function unit	dmem
Operation code	8
Number of operands	1
Modifier	7 bits
Modifier range	–64..63
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8 ild8d uld8r ild8r

DESCRIPTION

The `uld8d` operation loads the 8-bit memory value from the address computed by r*src1* + *d*, zero extends it to 32 bits, and writes the result in r*dest*. The *d* value is an opcode modifier in the range –64 to 63 inclusive. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8d` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8d` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8d` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, [0xd02] = 0x22	`uld8d(2) r10 → r60`	r60 ← 0x000022
r30 = 0, r20 = 0xd04, [0xd00] = 0x84	`IF r30 uld8d(-4) r20 → r70`	no change, since guard is false
r40 = 1, r20 = 0xd04, [0xd00] = 0x84	`IF r40 uld8d(-4) r20 → r80`	r80 ← 0x00000084
r50 = 0xd05, [0xd01] = 0x33	`uld8d(-4) r50 → r90`	r90 ← 0x00000033

© TriMedia Technologies, Inc.

Unsigned 8-bit load with index

uld8r

SYNTAX

```
[ IF rguard ] uld8r rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext8to32(mem[r*src1* + r*src2*])

ATTRIBUTES

Function unit	dmem
Operation code	194
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	4, 5

SEE ALSO

uld8 ild8 uld8d ild8d ild8r

DESCRIPTION

The `uld8r` operation loads the 8-bit memory value from the address computed by r*src1* + r*src2*, zero extends it to 32 bits, and writes the result in r*dest*. This operation does not depend on the bytesex bit in the PCSW since only a single byte is loaded.

The result of an access by `uld8r` to the MMIO address aperture is undefined; access to the MMIO aperture is defined only for 32-bit loads and stores.

The `uld8r` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register and the occurrence of side effects. If the LSB of r*guard* is 1, r*dest* is written and the data cache status bits are updated if the addressed location is cacheable. if the LSB of r*guard* is 0, r*dest* is not changed and `uld8r` has no side effects whatever.

EXAMPLES

Initial Values	Operation	Result
r10 = 0xd00, r20 = 2, [0xd02] = 0x22	`uld8r r10 r20 → r80`	r80 ← 0x00000022
r50 = 0, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r50 uld8r r40 r30 → r90`	no change, since guard is false
r60 = 1, r40 = 0xd04, r30 = 0xfffffffc, [0xd00] = 0x84	`IF r60 uld8r r40 r30 → r100`	r100 ← 0x00000084
r70 = 0xd05, r30 = 0xfffffffc, [0xd01] = 0x33	`uld8r r70 r30 → r110`	r110 ← 0x00000033

uleq

Unsigned compare less or equal

pseudo-op for ugeq

SYNTAX

```
[ IF rguard ] uleq rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then {**
  if (unsigned)r*src1* <= (unsigned)r*src2* **then**
    r*dest* ← 1
  else
    r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	35
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`ileq` `uleqi`

DESCRIPTION

The `uleq` operation is a pseudo operation transformed by the scheduler into an `ugeq` with the arguments exchanged (`uleq`'s r*src1* is `ugeq`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `uleq` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uleq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`uleq r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 uleq r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 uleq r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`uleq r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`uleq r70 r70 → r110`	r110 ← 1

© TriMedia Technologies, Inc.

Unsigned compare less or equal with immediate

uleqi

SYNTAX

```
[ IF rguard ] uleqi(n) rsrc1 → rdest
```

FUNCTION

if r*guard* **then {**
if (unsigned)r*src1* <= (unsigned)*n* **then**
r*dest* ← 1
else
r*dest* ← 0
}

ATTRIBUTES

Function unit	alu
Operation code	43
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`uleq` `ileqi`

DESCRIPTION

The `uleqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than or equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uleqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`uleqi(2) r30 → r80`	r80 ← 0
r30 = 3	`uleqi(3) r30 → r90`	r90 ← 1
r30 = 3	`uleqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 uleqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 uleqi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`uleqi(127) r60 → r120`	r120 ← 0

ules

Unsigned compare less
pseudo-op for ugtr

SYNTAX

```
[ IF rguard ] ules rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 < (unsigned)rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	33
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`iles` `ugtr`

DESCRIPTION

The `ules` operation is a pseudo operation transformed by the scheduler into an `ugtr` with the arguments exchanged (`ules`'s r*src1* is `ugtr`'s r*src2* and vice versa). (Note: pseudo operations cannot be used in assembly source files.)

The `ules` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the second argument, r*src2*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ules` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	`ules r30 r40 → r80`	r80 ← 1
r10 = 0, r60 = 0x100, r30 = 3	`IF r10 ules r60 r30 → r50`	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x100	`IF r20 ules r50 r60 → r90`	r90 ← 0
r70 = 0x80000000, r40 = 4	`ules r70 r40 → r100`	r100 ← 0
r70 = 0x80000000	`ules r70 r70 → r110`	r110 ← 0

© TriMedia Technologies, Inc.

Unsigned compare less with immediate

ulesi

SYNTAX

```
[ IF rguard ] ulesi(n) rsrc1 → rdest
```

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 < (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	41
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ules ilesi

DESCRIPTION

The `ulesi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is less than the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `ulesi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ulesi(2) r30 → r80`	r80 ← 0
r30 = 3	`ulesi(3) r30 → r90`	r90 ← 0
r30 = 3	`ulesi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 ulesi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 ulesi(63) r40 → r100`	r100 ← 0
r60 = 0x80000000	`ulesi(127) r60 → r120`	r120 ← 0

ume8ii

Unsigned sum of absolute values of signed 8-bit differences

SYNTAX

```
[ IF rguard ] ume8ii rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
rdest ← abs_val(sign_ext8to32(*rsrc1*<31:24>) – sign_ext8to32(*rsrc2*<31:24>)) +
abs_val(sign_ext8to32(*rsrc1*<23:16>) – sign_ext8to32(*rsrc2*<23:16>)) +
abs_val(sign_ext8to32(*rsrc1*<15:8>) – sign_ext8to32(*rsrc2*<15:8>)) +
abs_val(sign_ext8to32(*rsrc1*<7:0>) – sign_ext8to32(*rsrc2*<7:0>))

ATTRIBUTES

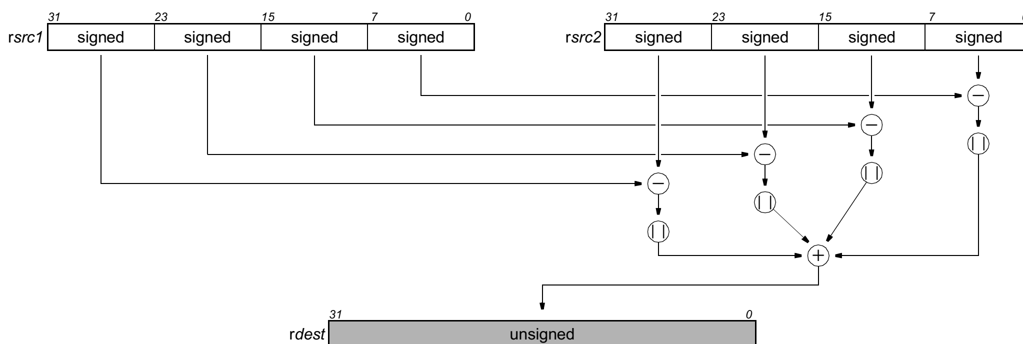
Function unit	dspalu
Operation code	64
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

ume8uu

DESCRIPTION

As shown below, the ume8ii operation computes four separate differences of the four pairs of corresponding signed 8-bit bytes of *rsrc1* and *rsrc2*; the absolute values of the four differences are summed, and the sum is written to *rdest*. All computations are performed without loss of precision.



The ume8ii operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`ume8ii r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8ii r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8ii r90 r40 → r110`	r110 ← 0x190
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8ii r40 r90 → r120`	r120 ← 0x190
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8ii r50 r60 → r125`	r125 ← 0x3fc

© TriMedia Technologies, Inc.

Sum of absolute values of unsigned 8-bit differences

ume8uu

SYNTAX

```
[ IF rguard ] ume8uu rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← abs_val(zero_ext8to32(r*src1*<31:24>) – zero_ext8to32(r*src2*<31:24>)) +
abs_val(zero_ext8to32(r*src1*<23:16>) – zero_ext8to32(r*src2*<23:16>)) +
abs_val(zero_ext8to32(r*src1*<15:8>) – zero_ext8to32(r*src2*<15:8>)) +
abs_val(zero_ext8to32(r*src1*<7:0>) – zero_ext8to32(r*src2*<7:0>))

ATTRIBUTES

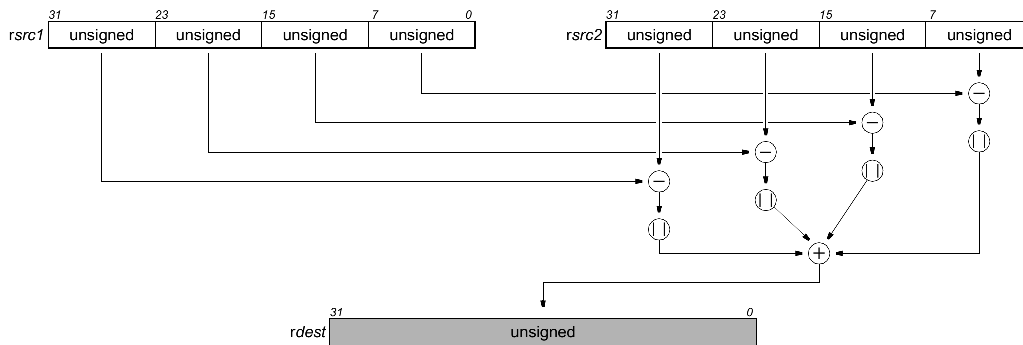
Attribute	Value
Function unit	dspalu
Operation code	26
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

ume8ii

DESCRIPTION

As shown below, the ume8uu operation computes four separate differences of the four pairs of corresponding unsigned 8-bit bytes of r*src1* and r*src2*. The absolute values of the four differences are summed and the result is written to r*dest*. All computations are performed without loss of precision.



The ume8uu operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`ume8uu r80 r30 → r100`	r100 ← 0x14
r10 = 0, r80 = 0x0a14f6f6, r30 = 0x1414ecf6	`IF r10 ume8uu r80 r30 → r70`	no change, since guard is false
r20 = 1, r90 = 0x64649c9c, r40 = 0x649c649c	`IF r20 ume8uu r90 r40 → r110`	r110 ← 0x70
r40 = 0x649c649c, r90 = 0x64649c9c	`ume8uu r40 r90 → r120`	r120 ← 0x70
r50 = 0x80808080, r60 = 0x7f7f7f7f	`ume8uu r50 r60 → r125`	r125 ← 0x4

umin

Minimum of unsigned values

pseudo-op for uclipu

SYNTAX

```
[ IF rguard ] umin rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 > rsrc2 then
        rdest ← rsrc2
    else
        rdest ← rsrc1
}
```

ATTRIBUTES

Function unit	dspalu
Operation code	76
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipi imin imax

DESCRIPTION

The umin operation returns the minimum value of r*src1* and r*src2*. The arguments r*src1* and r*src2* are considered unsigned integers.

The umin operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	umin r30 r40 → r50	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	IF r10 umin r60 r70 → r80	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	IF r20 umin r60 r70 → r90	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3fffff	umin r100 r110 → r120	r120 ← 0x3fffff

© TriMedia Technologies, Inc.

Unsigned multiply

umul

SYNTAX

```
[ IF rguard ] umul rsrc1 rsrc2 → rdest
```

FUNCTION

if r*guard* **then**
temp ← zero_ext32to64(r*src1*) × zero_ext32to64(r*src2*)
r*dest* ← temp<31:0>

ATTRIBUTES

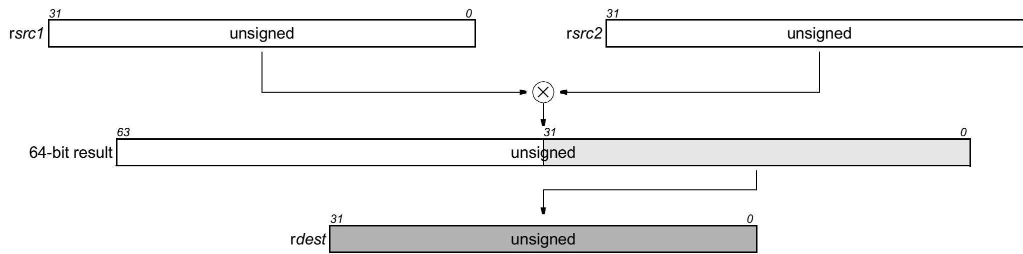
Function unit	ifmul
Operation code	138
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

imul imulm umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the umul operation computes the product r*src1*×r*src2* and writes the least-significant 32 bits of the full 64-bit product into r*dest*. The operands are considered unsigned integers. No overflow or underflow detection is performed.



The umul operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x100	umul r60 r60 → r80	r80 ← 0x10000
r10 = 0, r60 = 0x100, r30 = 0xf11	IF r10 umul r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x100, r30 = 0xf11	IF r20 umul r60 r30 → r90	r90 ← 0xf1100
r70 = 0x100, r40 = 0xffffff9c	umul r70 r40 → r100	r100 ← 0xffff9c00

umulm Unsigned multiply, return most-significant 32 bits

SYNTAX

```
[ IF rguard ] umulm rsrc1 rsrc2 → rdest
```

FUNCTION

if *rguard* **then**
temp ← zero_ext32to64(r*src1*) × zero_ext32to64(r*src2*)
r*dest* ← temp<63:32>

ATTRIBUTES

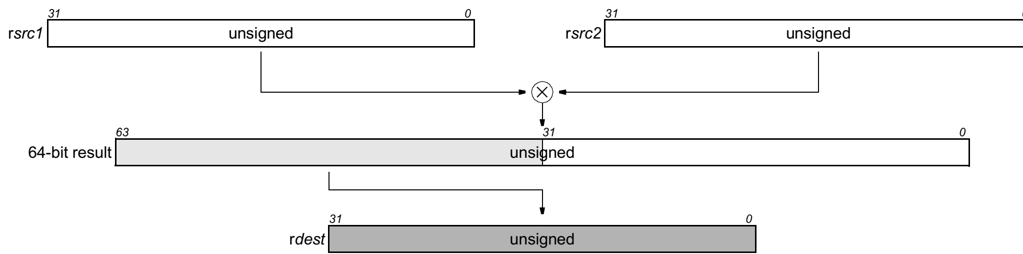
Function unit	ifmul
Operation code	140
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

umulm dspimul dspumul dspidualmul quadumulmsb fmul

DESCRIPTION

As shown below, the umulm operation computes the product r*src1*×r*src2* and writes the most-significant 32 bits of the 64-bit product into r*dest*. The operands are considered unsigned integers.



The umulm operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r60 = 0x10000	umulm r60 r60 → r80	r80 ← 0x00000001
r10 = 0, r60 = 0x100, r30 = 0xf11	IF r10 umulm r60 r30 → r50	no change, since guard is false
r20 = 1, r60 = 0x10001000, r30 = 0xf1100000	IF r20 umulm r60 r30 → r90	r90 ← 0xf110f11
r70 = 0xffffff00, r40 = 0x100	umulm r70 r40 → r100	r100 ← 0xff

 © TriMedia Technologies, Inc.

Unsigned compare not equal

pseudo-op for ineq

uneq

SYNTAX

```
[ IF rguard ] uneq rsrc1 rsrc2 → rdest
```

FUNCTION

```
if rguard then {
    if rsrc1 != rsrc2 then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	39
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

ineq igtr uneqi

DESCRIPTION

The `uneq` operation is a pseudo operation transformed by the scheduler into an `ineq`. (Note: pseudo operations cannot be used in assembly source files.)

The `uneq` operation sets the destination register, r*dest*, to 1 if the two arguments, r*src1* and r*src2*, are not equal; otherwise, r*dest* is set to 0.

The `uneq` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3, r40 = 4	uneq r30 r40 → r80	r80 ← 1
r10 = 0, r60 = 0x1000, r30 = 3	IF r10 uneq r60 r30 → r50	no change, since guard is false
r20 = 1, r50 = 0x1000, r60 = 0x1000	IF r20 uneq r50 r60 → r90	r90 ← 0
r70 = 0x80000000, r40 = 4	uneq r70 r40 → r100	r100 ← 1
r70 = 0x80000000	uneq r70 r70 → r110	r110 ← 0

uneqi

Unsigned compare not equal with immediate

SYNTAX

`[ IF rguard ] uneqi(n) rsrc1 → rdest`

FUNCTION

```
if rguard then {
    if (unsigned)rsrc1 != (unsigned)n then
        rdest ← 1
    else
        rdest ← 0
}
```

ATTRIBUTES

Function unit	alu
Operation code	40
Number of operands	1
Modifier	7 bits
Modifier range	0..127
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

`uneq` `ineqi` `ueqli`

DESCRIPTION

The `uneqi` operation sets the destination register, r*dest*, to 1 if the first argument, r*src1*, is not equal to the opcode modifier, *n*; otherwise, r*dest* is set to 0. The arguments are treated as unsigned integers.

The `uneqi` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`uneqi(2) r30 → r80`	r80 ← 1
r30 = 3	`uneqi(3) r30 → r90`	r90 ← 0
r30 = 3	`uneqi(4) r30 → r100`	r100 ← 1
r10 = 0, r40 = 0x100	`IF r10 uneqi(63) r40 → r50`	no change, since guard is false
r20 = 1, r40 = 0x100	`IF r20 uneqi(63) r40 → r100`	r100 ← 1
r60 = 0x80000000	`uneqi(127) r60 → r120`	r120 ← 1

© TriMedia Technologies, Inc.

Write destination program counter

writedpc

SYNTAX

```
[ IF rguard ] writedpc rsrc1
```

FUNCTION

if r*guard* **then {**
DPC ← r*src1*
}

ATTRIBUTES

Function unit	fcomp
Operation code	160
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

readdpc writespc ijmpf ijmpi ijmpt

DESCRIPTION

The writedpc copies the value of r*src1* to the DPC (Destination Program Counter) processor register. Whenever a hardware update (during an interruptible jump) and a software update (through a writedpc) coincide, the software update takes precedence.

Interruptible jumps write their target address to the DPC. The value of DPC is intended to be used by an exception-handling routine as a jump address to resume execution of the program that was running before the exception was taken.

The writedpc operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of DPC. If the LSB of r*guard* is 1, DPC is written; otherwise, DPC is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xbeebee	writedpc r30	DPC ← 0xbeebee
r20 = 0, r31 = 0xabba	IF r20 writedpc r31	no change, since guard is false
r21 = 1, r31 = 0xabba	IF r21 writedpc r31	DPC ← 0xabba

writepcsw Write program control and status word

SYNTAX

```
[ IF rguard ] writepcsw rsrc1 rsrc2
```

FUNCTION

if *rguard* **then {**
PCSW ← (PCSW & ~*rsrc2*) | (*rsrc1* & *rsrc2*)
}

ATTRIBUTES

Function unit	fcomp
Operation code	161
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

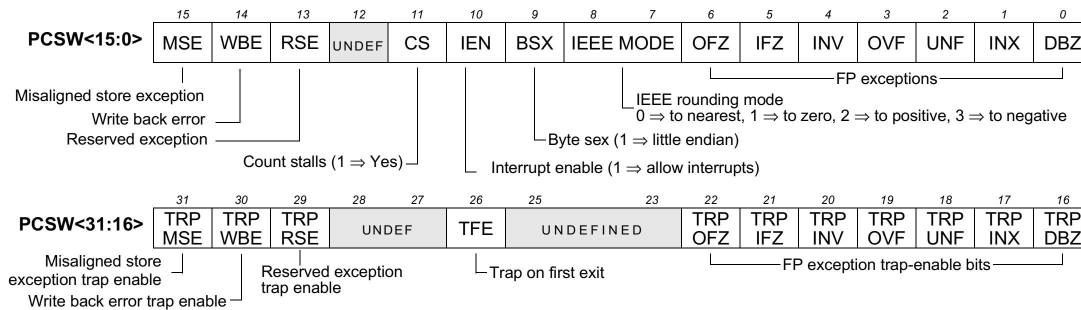
readpcsw fadd faddflags
ijmpf cycles hicycles

DESCRIPTION

The `writepcsw` copies the value of *rsrc1* to the PCSW (Program Control and Status Word) processor register using rsrc2 as a mask. A bit in PCSW is affected by `writepcsw` only if the corresponding bit in rsrc2 is set to 1; the value of any bit in PCSW with a corresponding 0-bit in rsrc2 will not be changed by `writepcsw`. Whenever a hardware update (e.g., when a floating-point exception is raised) and a software update (through a `writepcsw`) coincide, the PCSW bits currently being updated by hardware will reflect the hardware-determined value while the bits not being affected by hardware will reflect the value in the `writepcsw` operand. The layout of PCSW is shown below. The programmer should take care not to alter UNDEF fields in the PCSW.

Fields in the PCSW have two chief purposes: to control aspects of processor operation and to record events that occur during program execution. Thus, `writepcsw` can be used to effect changes in some aspects of processor operation and to clear fields that record events; this operation can also be used to restore state before resuming an idled task in a multi-tasking environment. Note: The latency of writepcsw is 1, i.e. the PCSW reflects the new value in the next cycle. But it takes additional 3 cycles for updates to the exception flags and exception enable bits to take effect in the hardware. Therefore 3 delay slots / nops shall be inserted between writepcsw and the next interruptible jump, if exception flags or enable bits are changed. This guarantees that the new state is recognized in the interrupt logic during execution of the ijump.

The `writepcsw` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of PCSW. If the LSB of *rguard* is 1, PCSW is written; otherwise, PCSW is unchanged.



EXAMPLES

Initial Values	Operation	Result
r30 = 0x100, r40 = 0x180	`writepcsw r30 r40`	PCSW.IEEE MODE = to positive infinity
r20 = 0, r50 = 0x0, r60 = 0x400	`IF r20 writepcsw r50 r60`	no change, since guard is false
r21 = 1, r50 = 0x0, r60 = 0x400	`IF r21 writepcsw r50 r60`	PCSW.IEN = 0 (disable interrupts)
r70 = 0x80110000, r80 = 0xffff0000	`writepcsw r70 r80`	enable trap on MSE, INV and DBZ exclusively

© TriMedia Technologies, Inc.

Write source program counter

writespc

SYNTAX

```
[ IF rguard ] writespc rsrc1
```

FUNCTION

if r*guard* **then**
SPC ← r*src1*

ATTRIBUTES

Function unit	fcomp
Operation code	159
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	3

SEE ALSO

readspc writedpc ijmpf ijmpi ijmpt

DESCRIPTION

The `writespc` copies the value of r*src1* to the SPC (Source Program Counter) processor register. Whenever a hardware update (during an interruptible jump) and a software update (through a `writespc`) coincide, the software update takes precedence.

An interruptible jump that is not interrupted (no NMI, INT, or EXC event was pending when the jump was executed) writes its target address to SPC. The value of SPC is intended to allow an exception-handling routine to determine the start address of the block of scheduled code (called a decision tree) that was executing before the exception was taken.

The `writespc` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of SPC. If the LSB of r*guard* is 1, SPC is written; otherwise, SPC is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xbeebee	`writespc r30`	SPC ← 0xbeebee
r20 = 0, r31 = 0xabba	`IF r20 writespc r31`	no change, since guard is false
r21 = 1, r31 = 0xabba	`IF r21 writespc r31`	SPC ← 0xabba

zex16

Zero extend 16 bits

pseudo-op for pack16lsb

SYNTAX

```
[ IF rguard ] zex16 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
r*dest* ← zero_ext16to32(r*src1*<15:0>)

ATTRIBUTES

Function unit	alu
Operation code	53
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

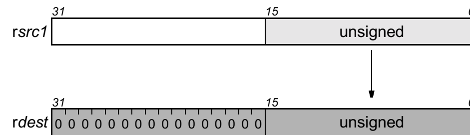
SEE ALSO

sex16 sex8 zex8

DESCRIPTION

The zex16 operation is a pseudo operation transformed by the scheduler into a pack16lsb with 0 as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the zex16 operation zero extends the least-significant 16-bit halfword of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The zex16 operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	`zex16 r30 → r60`	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	`IF r10 zex16 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	`IF r20 zex16 r40 → r100`	r100 ← 0x0000ff91
r50 = 0x00000091	`zex16 r50 → r110`	r110 ← 0x00000091

 © TriMedia Technologies, Inc.

Zero extend 8 bits

pseudo-op for ubytesel

SYNTAX

```
[ IF rguard ] zex8 rsrc1 → rdest
```

FUNCTION

if r*guard* **then**
  r*dest* ← zero_ext8to32(r*src1*<7:0>)

ATTRIBUTES

Function unit	alu
Operation code	55
Number of operands	1
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

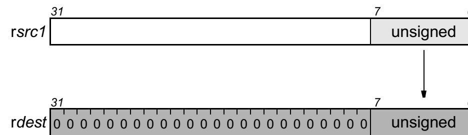
SEE ALSO

ubytesel sex16 sex8 zex16

DESCRIPTION

The `zex8` operation is a pseudo operation transformed by the scheduler into a `ubytesel` with r0 (always contains 0) as the first argument and r*src1* as the second. (Note: pseudo operations cannot be used in assembly source files.)

As shown below, the `zex8` operation zero extends the least-significant byte of the argument, r*src1*, to 32 bits and writes the result in r*dest*.



The `zex8` operation optionally takes a guard, specified in r*guard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of r*guard* is 1, r*dest* is written; otherwise, r*dest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xffff0040	`zex8 r30 → r60`	r60 ← 0x00000040
r10 = 0, r40 = 0xff0fff91	`IF r10 zex8 r40 → r70`	no change, since guard is false
r20 = 1, r40 = 0xff0fff91	`IF r20 zex8 r40 → r100`	r100 ← 0x00000091
r50 = 0x00000091	`zex8 r50 → r110`	r110 ← 0x00000091

 © TriMedia Technologies, Inc.

Appendix B MMIO Registers

The following table lists all the MMIO registers implemented inside the TM32 CPU core.

Programmer's note: For compatibility with future devices, any undefined MMIO bits should be ignored when read, and written as zeroes.

Table 0-1 MMIO Registers

MMIO Register Name	MMIO Address Offset (in hex)	Accessibility		Description
		DSP CPU 32	Other PI masters	
DSPCPU32 Registers				
TM32_CTL	10 0030	R/-	R/W	Controls (start, stop, frequency setting) the TM32 CPU core.
TM32_DRAM_LO	10 0034	R/-	R/W	Lowest DRAM address that TM32 CPU can access.
TM32_DRAM_CLIMIT	10 003C	R/-	R/W	Start of non-cacheable region in DRAM (very similar to old DRAM_CACHEABLE_LIMIT at 0x10,0008)
TM32_DRAM_HI	10 0038	R/-	R/W	Highest DRAM address that TM32 CPU can access + 1
TM32_APERT1_LO	10 0040	R/-	R/W	Lowest TM32 CPU address that resolves to a PI bus access
TM32_APERT1_HI	10 0044	R/-	R/W	Highest TM32 CPU address that resolves to a PI bus access + 1
TM32_START_ADDR	10 0048	R/-	R/W	Address that TM32 CPU execution starts from upon START.
TM32_PC	10 004C	R/-	R/-	Read-only observation of Program Counter value of TM32 CPU
EXCVEC	10 0800	R/W	R/W	Interrupt vector (handler start address) for exceptions
ISETTING0	10 0810	R/W	R/W	Interrupt mode & priority settings for sources 0-7
ISETTING1	10 0814	R/W	R/W	Interrupt mode & priority settings for sources 8-15
ISETTING2	10 0818	R/W	R/W	Interrupt mode & priority settings for sources 16-23
ISETTING3	10 081c	R/W	R/W	Interrupt mode & priority settings for sources 24-31
IPENDING	10 0820	R/W	R/W	Interrupt-pending status bit for all 32 sources
ICLEAR	10 0824	R/W	R/W	Interrupt-clear bit for all 32 sources
IMASK	10 0828	R/W	R/W	Interrupt-mask bit for all 32 sources

Table 0-1 MMIO Registers

MMIO Register Name	MMIO Address Offset (in hex)	Accessibility		Description
		DSP CPU 32	Other PI masters	
INTVEC0	10 0880	R/W	R/W	Interrupt vector (handler start address) for source 0
INTVEC1	10 0884	R/W	R/W	Interrupt vector (handler start address) for source 1
INTVEC2	10 0888	R/W	R/W	Interrupt vector (handler start address) for source 2
INTVEC3	10 088c	R/W	R/W	Interrupt vector (handler start address) for source 3
INTVEC4	10 0890	R/W	R/W	Interrupt vector (handler start address) for source 4
INTVEC5	10 0894	R/W	R/W	Interrupt vector (handler start address) for source 5
INTVEC6	10 0898	R/W	R/W	Interrupt vector (handler start address) for source 6
INTVEC7	10 089c	R/W	R/W	Interrupt vector (handler start address) for source 7
INTVEC8	10 08a0	R/W	R/W	Interrupt vector (handler start address) for source 8
INTVEC9	10 08a4	R/W	R/W	Interrupt vector (handler start address) for source 9
INTVEC10	10 08a8	R/W	R/W	Interrupt vector (handler start address) for source 10
INTVEC11	10 08ac	R/W	R/W	Interrupt vector (handler start address) for source 11
INTVEC12	10 08b0	R/W	R/W	Interrupt vector (handler start address) for source 12
INTVEC13	10 08b4	R/W	R/W	Interrupt vector (handler start address) for source 13
INTVEC14	10 08b8	R/W	R/W	Interrupt vector (handler start address) for source 14
INTVEC15	10 08bc	R/W	R/W	Interrupt vector (handler start address) for source 15
INTVEC16	10 08c0	R/W	R/W	Interrupt vector (handler start address) for source 16
INTVEC17	10 08c4	R/W	R/W	Interrupt vector (handler start address) for source 17
INTVEC18	10 08c8	R/W	R/W	Interrupt vector (handler start address) for source 18
INTVEC19	10 08cc	R/W	R/W	Interrupt vector (handler start address) for source 19
INTVEC20	10 08d0	R/W	R/W	Interrupt vector (handler start address) for source 20
INTVEC21	10 08d4	R/W	R/W	Interrupt vector (handler start address) for source 21
INTVEC22	10 08d8	R/W	R/W	Interrupt vector (handler start address) for source 22
INTVEC23	10 08dc	R/W	R/W	Interrupt vector (handler start address) for source 23
INTVEC24	10 08e0	R/W	R/W	Interrupt vector (handler start address) for source 24
INTVEC25	10 08e4	R/W	R/W	Interrupt vector (handler start address) for source 25
INTVEC26	10 08e8	R/W	R/W	Interrupt vector (handler start address) for source 26
INTVEC27	10 08ec	R/W	R/W	Interrupt vector (handler start address) for source 27
INTVEC28	10 08f0	R/W	R/W	Interrupt vector (handler start address) for source 28
INTVEC29	10 08f4	R/W	R/W	Interrupt vector (handler start address) for source 29
INTVEC30	10 08f8	R/W	R/W	Interrupt vector (handler start address) for source 30
INTVEC31	10 08fc	R/W	R/W	Interrupt vector (handler start address) for source 31
TIMER1_TMODULUS	10 0c00	R/W	R/W	Contains: (maximum count value for timer 1) + 1

 © TriMedia Technologies, Inc.

Table 0-1 MMIO Registers

MMIO Register Name	MMIO Address Offset (in hex)	Accessibility		Description
		DSP CPU 32	Other PI masters	
TIMER1_TVALUE	10 0c04	R/W	R/W	Current value of timer 1 counter
TIMER1_TCTL	10 0c08	R/W	R/W	Timer 1 control (prescale value, source select, run bit)
TIMER2_TMODULUS	10 0c20	R/W	R/W	Contains: (maximum count value for timer 2) + 1
TIMER2_TVALUE	10 0c24	R/W	R/W	Current value of timer 2 counter
TIMER2_TCTL	10 0c28	R/W	R/W	Timer 2 control (prescale value, source select, run bit)
TIMER3_TMODULUS	10 0c40	R/W	R/W	Contains: (maximum count value for timer 3) + 1
TIMER3_TVALUE	10 0c44	R/W	R/W	Current value of timer 3 counter
TIMER3_TCTL	10 0c48	R/W	R/W	Timer 3 control (prescale value, source select, run bit)
SYSTIMER_TMODULUS	10 0c60	R/W	R/W	Contains: (maximum count value for system timer) + 1
SYSTIMER_TVALUE	10 0c64	R/W	R/W	Current value of system timer/counter
SYSTIMER_TCTL	10 0c68	R/W	R/W	System timer control (prescale value, source select, run bit)
BICTL	10 1000	R/W	R/W	Instruction breakpoint control
BINSTLOW	10 1004	R/W	R/W	Start of address range that causes instruction breakpoints
BINSTHIGH	10 1008	R/W	R/W	End of address range that causes instruction breakpoints
BDCTL	10 1020	R/W	R/W	Data breakpoint control
BDATAALOW	10 1030	R/W	R/W	Start of address range that causes data breakpoints
BDATAAHIGH	10 1034	R/W	R/W	End of address range that causes data breakpoints
BDATAVAL	10 1038	R/W	R/W	Compare value for data breakpoints
BDATAMASK	10 103c	R/W	R/W	Compare mask for compare value for data breakpoints
Cache And Memory System				
MEM_EVENTS	10 000c	R/W	R/W	Selects two cache-related events for counting
DC_LOCK_CTL	10 0010	R/W	R/W	Enable bit for data-cache locking, also PCI hole disable
DC_LOCK_ADDR	10 0014	R/W	R/W	Start of address range that will be locked into the data cache
DC_LOCK_SIZE	10 0018	R/W	R/W	Size of address range that will be locked into the data cache
DC_PARAMS	10 001c	R/—	R/—	Data-cache geometry (blocksize, associativity, # of sets)
IC_PARAMS	10 0020	R/—	R/—	Instruction-cache geometry (blocksize, assoc., # of sets)
POWER_DOWN	10 0108	R/W	-/-	Write to this register initiates TM32 CPU power down
IC_LOCK_CTL	10 0210	R/W	R/W	Enable bit for instruction-cache locking
IC_LOCK_ADDR	10 0214	R/W	R/W	Start of address range that will be locked into the instruction cache
IC_LOCK_SIZE	10 0218	R/W	R/W	Size of address range that will be locked into the instruction cache

Table 0-1 MMIO Registers

MMIO Register Name	MMIO Address Offset (in hex)	Accessibility		Description
		DSP CPU 32	Other PI masters	
Miscellaneous				
SEM	10 0500	R/W	R/W	semaphore assist device
TM32_MODID	10 0ffc	R/-	R/-	Module ID register for software PI-bus device enumeration

 © TriMedia Technologies, Inc.

Appendix C Signal Interface

C.1 Signals

Table 0-1 Signal List

Signal Name	Direc-tion	Description
System Interface		
clk_tm32	IN	Main Input Clock. TM32 derives its internal clocks from this clock using a module internal, programmable PLL clock system. Hence, this clock must NEVER be stopped, unless the TM32 CPU itself is first stopped.
tm32_reset_n	IN	A ‘low’ on this input sets the TM32 core in its initial state. This is the primary hardware reset of the core. De-assertion does *not* start execution - execution can only be started by writing to the TM32_CTL MMIO register.
pi_reset_n	IN	A ‘low’ on this input resets the PI-bus interface of the TM32 CPU.
tm32_pwrdwn_req	IN	Asserting this asynchronous input causes the TM32 CPU to stop the internal clock. Negating it causes the TM32 CPU to restart its internal clock and continue execution where it left off.
tm32_pwrdwn_ack	OUT	Asserted when externally initiated powerdown is accomplished. De-asserted when restart is accomplished.
tm32_mmio_base[31:21]	IN	Value on this bus sets the TM32 CPU MMIO Aperture origin. This value may not be changed while TM32 CPU is executing.
tm32_irq_31to9	IN	Interrupt request lines, for sources 31..9. A ‘1’ (level sensitive mode) or positive edge (edge triggered mode) on this line asserts the corresponding request, and will cause instruction execution to go to the associated interrupt vector address. NOTE: source 8..5 are TM32 CPU core internal sources (internal timers).
tm32_irq_4to0	IN	Interrupt request lines, for sources 4..0. A ‘1’ (level sensitive mode) or positive edge (edge triggered mode) on this line asserts the corresponding request, and will cause instruction execution to go to the associated interrupt vector address. NOTE: source 8..5 are TM32 CPU core internal sources (internal timers).

Table 0-1 Signal List

Signal Name	Direc-tion	Description
tm32_timerin[15:7]	IN	general purpose timer input wires for resp. timer source selection 7..15
tm32_tri_timer_clkin	IN	general purpose timer input wire for timer source selection 2
Trimedia Memory Bus (synchronous to clk_tm32)		
tm32_mreq	OUT	- see timing section 1.11.3 -
tm32_msize[6:3]	OUT	transfer length code, as per MMI table
tm32_mread	OUT	
tm32_wmask[7:0	OUT	
tm32_maddr[31:0]	OUT	Full 32 bit address
tm32_wdata[63:0]	OUT	
tm32_mack	IN	
tm32_rlast	IN	
tm32_rvalid	IN	
tm32_wlast	IN	
tm32_wvalid	IN	
tm32_rdata[63:0]	IN	
PI-bus master/slave interface (synchronous to clk_pi_tm32)		
clk_pi_tm32	IN	
pi_reset_n	IN	see System Interface Group above (repeated for clarity only)
pi_tout	IN	
pi_sel	IN	
pi_ack[2:0]	I/O	
pi_read	I/O	
pi_opc[4:0]	I/O	
pi_a[31:2]	I/O	Full 32 bit address, except for byte within a 32 bit word
pi_d[31:0]	I/O	
pi_lock	I/O	
pi_req	OUT	
pi_gnt	IN	
pi_bigendian	IN	
Hardware System Debug Provisions (spy)		
tm32_core_spy[9:0]	OUT	This bus can be set to emit selected signals from the Dcache, VIC, Icache or CPU bridge for core debug. Refer to the document titled “TM32 CPU Core Spy Signals”.
Module Test Interface		
TCB_TDI	IN	Test Control Block Input data

 © TriMedia Technologies, Inc.

Table 0-1 Signal List

Signal Name	Direc- tion	Description
TCB_TDO	OUT	Test Control Block Outpu data
TCB_TCK	IN	Test Control Block Clock
TCB_TRSTN	IN	Test Control Block reset (active low)
TCB_UPDATE	IN	Test Control Block Update signal
TCB_HOLD	IN	Test Control Block Hold signal
TCB_TC	IN	Test Control Block Test Control Toggle
tm32_si[39:0]		core scan chain inputs
tm32_so[39:0]	OUT	core scan chain outputs
tm32_ssi[1:0]	IN	core surround scan chain inputs
tm32_sso[1:0]	OUT	core surround scan chain outputs
Test Chip Signals		
tm32_dvp_mode	IN	Should be tied to ‘1’ to enable DVP mode operation of the TM32 CPU core.
tm32_vic_wakeup	OUT	leave unconnected
tm32_hwy_address[31:0]	I/O	connect to buskeeper
tm32_hwy_valid	IN	tie to ‘0’
tm32_hwy_transfer	IN	tie to ‘0’
tm32_hwy_byte_en[3:0]	I/O	connect to buskeeper
tm32_hwy_data[31:0]	I/O	connect to buskeeper
tm32_hwy_dc_ack	IN	tie to ‘0’
tm32_hwy_ic_ack	IN	tie to ‘0’
tm32_hwy_mmio	IN	tie to ‘0’
tm32_hwy_dc_mmio_req	OUT	leave unconnected
tm32_hwy_dc_req_pref	OUT	leave unconnected
tm32_hwy_dc_req	OUT	leave unconnected
tm32_hwy_ic_req	OUT	leave unconnected
tm32_hwy_opcode	I/O	connect to buskeeper
tm32_hwy_dc_mmio_reply	OUT	leave unconnected
tm32_hwy_ic_mmio_reply	OUT	leave unconnected
tm32_hwy_vic_mmio_reply	OUT	leave unconnected

Table 0-1 Signal List

Signal Name	Direc-tion	Description
tm32_CR[2:0]	IN	tie to ‘0’
tm32_CB	IN	tie to ‘0’
tm32_CD	IN	tie to ‘0’
tm32_powerdown	IN	tie to ‘0’
tm32_pci_dram_base [31:21]	IN	tie to ‘0’
tm32_pci_ready	IN	tie to ‘0’
tm32_bti_cpu_arstB	IN	tie to ‘0’
tm32_mm_notreversed	IN	tie to ‘0’
tm32_dc_pwr_det	OUT	leave unconnected
tm32_dc_pci_op	OUT	leave unconnected
tm32_vic_powerdown	IN	tie to ‘0’
Pci_mmi_ack	IN	tie to ‘0’
Pci_req_mmi	OUT	leave unconnected
Pci_mod_req	IN	tie to ‘0’

C.1.1 Timing

Trimedia memory bus will be 143 MHz minimum, with a target operating frequency of 167 MHz.

PI clock will be 75 MHz. TM32 CPU obeys PI-bus AC timing specs for this frequency.

All TM32 CPU core outputs come directly from a register clocked with a buffered clk_tm32, *with the exception of* the ‘mreq’ output, which has a few gates delay after its register. Hence ‘mreq’ needs to be specially handled to avoid a global critical path.

C.1.2 Interface Description

C.1.2.1 PI bus - master

PI transactions initiated by the TM32 CPU core as bus master are:

 © TriMedia Technologies, Inc.

- WDU read
- WDU write
- HW0, HW1 write
- BY0, BY1, BY2, BY3 write

The relation between CPU load operations in the PI-aperture and PI bus transactions is specified in Table 0-2. For stores, refer to Table 0-3. The WD16 read transactions are generated only by Icache misses due to TM32 CPU PI-aperture execution.

Table 0-2 . PI-aperture operation (CPU loads)

TM32 operation	endian mode	address	PI-bus opcode	PI-data	CPU register
32 bit load	little	0x1000	WDU	01020304	01020304
16 bit load	little	0x1000	WDU	01020304	00000304
16 bit load	little	0x1002	WDU	01020304	00000102
8 bit load	little	0x1000	WDU	01020304	00000004
8 bit load	little	0x1001	WDU	01020304	00000003
8 bit load	little	0x1002	WDU	01020304	00000002
8 bit load	little	0x1003	WDU	01020304	00000001
32 bit load	big	0x1000	WDU	01020304	01020304
16 bit load	big	0x1000	WDU	01020304	00000102
16 bit load	big	0x1002	WDU	01020304	00000304
8 bit load	big	0x1000	WDU	01020304	00000001
8 bit load	big	0x1001	WDU	01020304	00000002
8 bit load	big	0x1002	WDU	01020304	00000003
8 bit load	big	0x1003	WDU	01020304	00000004

Table 0-3 . PI-aperture operation (CPU stores)

TM32 operation	endian mode	address	CPU register	PI-bus opcode	PI-data
32 bit store	either	0x1000	01020304	WDU	01020304
16 bit store	either	0x1000	01020304	HW0	xxxx0304
16 bit store	either	0x1002	01020304	HW1	xxxx0304
8 bit store	either	0x1000	01020304	BY0	xxxxxx04
8 bit store	either	0x1001	01020304	BY1	xxxxxx04
8 bit store	either	0x1002	01020304	BY2	xxxxxx04
8 bit store	either	0x1003	01020304	BY3	xxxxxx04

Note that the TM32 CPU always retrieves a full 32-bit word across the PI-bus on 8 and 16 bit load operations, and selects the appropriate byte(s) based on address and endian mode.

The core handles slave initiated retract by re-issueing the request forever.

C.1.2.2 PI bus - slave

PI transactions supported by the TM32 CPU core as bus slave are:

- WDU read and WDU write

The only transactions supported as target are WDU read and write from/to the TM32 CPU core MMIO registers. Any transactions other than 32 bit wide are not supported, and lead to an error ack.

The TM32 CPU core will frequently issue a slave retract on both WDU reads and WDU writes. This is done to avoid locking down internal CPU buses for the relatively long duration of PI transactions.

C.1.2.3 PI-bus deadlock issues

The TM32 as master will retry a slave retracted transaction forever.

For a TM32 PI-bus aperture load/store, internal resources are not held during such a retry period, and the TM32 core will allow incoming PI-bus MMIO reads/writes while the retract/retry is going on.

For a MMIO aperture load/store, the TM32 does NOT allow incoming PI-bus transactions during a retract/retry sequence. This is a limitation that can in certain systems cuase a risk of PI-bus deadlock due to infinite mutual retry.

C.1.2.4 Trimedia memory bus

Trimedia memory bus transactions generated by the TM32 CPU are:

- read/write 64 byte, 64 byte aligned, sequential order
- read 64 byte, 64 byte aligned, critical word first, Xor style

 © TriMedia Technologies, Inc.

Index

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

address fields,instruction cache 11
address mapping
  instruction cache 11
    picture 11
addressing modes 6
alignment 4
alloc 3
allocate on write 4
allocd 4
allocr 5
allocx 6
aperture
  DRAM 1
asi 7
asli 8
asr 9
asri 10

B

BDATAAHIGH
  picture 12, 13, 21
BDATAALOW
  picture 12, 13, 15, 21
BDATAMASK
  picture 21
BDATAVAL
  picture 12, 21
BDCTL
  picture 21
BICTL
  picture 21
binary compatibility 7
BINSTHIGH
  picture 21
BINSTLOW
  picture 21
bitand 11
bitandinv 12
bitinv 13
bitor 14
bitxor 15
boolean representation 5
borrow 16
breakpoints 20
byte ordering
  DSPCPU 3
bytesex 3

C

cache
  address mapping,instruction cache 11
  alignment 2, 4
  associativity 2
  block size 2
  blocksize 2
  coherency 2, 3, 4, 16
  copyback 3
  copyback operation 6
  CPU stall 9
  data cache characteristics,table 2
  data cache initialization 9
  data cache,description 2
  dcb opcode 6
  dinvalid opcode 6
  dirty bit 3
  dirty bits 2
  dual port 4
  endian-ness 2, 4
  hidden concurrency 8
  iclr operation 12
  initialization 9
  instruction cache 10
  instruction cache coherency 12
  instruction cache initialization and boot 14
  instruction cache parameters 10
  instruction cache summary 10
  instruction cache tag 11
  invalidate operation 6
  latency 9
  locking 2, 4
  LRU replacement 14
  miss processing order 3, 11
  miss transfer order 2
  MMIO registers summary 18
  noncachable region 2

- non-cacheable region 5
- number of sets 2
- operation ordering 8
- overview 1
- parameters 2
- partial word transfers 4
- partial words 2
- performance evaluation support 17
- performance events
  - table 17
- ports 2
- replacement policies 2, 3
- replacement policy 11
- scheduling constraint 4
- size 2
- special data cache operations 6
- special opcodes 4
- special operation ordering 8
- status operations 7, 8
- tag operations 7, 8
- valid bits 2
- write misses 4

carry 17
CCCOUNT
- definition 5

coherency 4
coherency,instruction cache 12
compatibility
- software 7

concurrency,hidden 8
copyback 3
counter 19
CPU stall 9
curcycles 18
cycles 19

D

data breakpoint 20
data cache
- coherency 16
- dcb operation 6
- dinvalid operation 6
- initialization 9
- LRU replacement 14
- performance evaluation support 17
- rdstatus operation 7
- rdtag operation 7

DC_LOCK_ADDR
- description table 18

DC_LOCK_CTL
- description table 18

DC_LOCK_SIZE
- description table 18

DC_PARAMS
- description table 18
- picture 3, 10

DC_PARAMS register 2
dcb 6, 20
dcb operation 6
debug support 20
device control 11
device interrupts 19
dinvalid 6, 21
dinvalid operation 6
dirty bit 3
DPC
- definition 4

DRAM aperture 1
DRAM base 1
DRAM limit 1
DRAM_BASE
- description table 18

DRAM_CACHEABLE_LIMIT
- description table 18

DRAM_LIMIT
- description table 18

DSPCPU
- addressing modes 6
- byte ordering 3
- register model 1
- software compatibility 7

DSPCPU operations
- listed by function 2

dspiabs 22
dspiadd 23
dspidualabs 24
dspidualadd 25
dspidualmul 26
dspidualsub 27
dspimul 28
dspisub 29
dspuadd 30
dspumul 31
dspuquadaddui 32
dspusub 33
dual port 4

E

edge sensitive interrupts 16
endian-ness 4
endianness 3
exceptions
- definition 15

F

fabsval 37
fabsvalflags 38
fadd 39
faddflags 40
fdiv 41

 © TriMedia Technologies, Inc.

fdivflags 42
feql 43
feqlflags 44
fgeq 45
fgeqflags 46
fgtr 47
fgtrflags 48
fleq 49
fleqflags 50
fles 51
flesflags 52
floating point
  exception flags 3
  IEEE rounding mode 3
  representation 5
fmul 53
fmulflags 54
fneq 55
fneqflags 56
four-way LRU 15
fsign 57
fsignflags 58
fsqrt 59
fsqrtflags 60
fsub 61
fsubflags 62
funshift1 63
funshift2 64
funshift3 65

G

general purpose registers 1
general purpose timer/counter 19
guarding
  definition 7

H

h_dspiabs 66
h_dspidualabs 67
h_iabs 68
h_st16d 69
h_st32d 70
h_st8d 71
hicycles 72
hidden concurrency 8
hierarchical LRU 4

I

iabs 73
iadd 74
iaddi 75
iavgonep 76
ibytesel 77
IC_LOCK_ADDR
  description table 18
  picture 13
IC_LOCK_CTL
  description table 18
  picture 13
IC_LOCK_SIZE
  description table 18
  picture 13
IC_PARAMS
  description table 18
IC_PARAMS fields 10
ICLEAR
  picture 17
iclipi 78
iclr 12, 79
ident 80
IEEE rounding mode 3
ieql 81
ieqli 82
ifir16 83
ifir8ii 84
ifir8ui 85
ifixieee 86
ifixieeeflags 87
ifixrz 88
ifixrzflags 89
iflip 90
ifloat 91
ifloatflags 92
ifloatrz 93
ifloatrzflags 94
igeq 95
igeqi 96
igtr 97
igtri 98
iimm 99
ijmpf 100
ijmpi 101
ijmpt 102
ild16 103
ild16d 104
ild16r 105
ild16x 106
ild8 107
ild8d 108
ild8r 109
ileq 110
ileqi 111
iles 112
ilesi 113
IMASK
  picture 17
imax 114

imin 115
imul 116
imulm 117
ineg 118
ineq 119
ineqi 120
initialization
  instruction cache 14
initialization,cache 9
inonzero 121
instruction breakpoint 20
instruction cache 10
  address mapping 11
    picture 11
  coherency 16
  initialization and boot 14
  LRU replacement 14
  performance evaluation support 17
instruction cache parameters 10
instruction cache set 11
instruction cache tag 11
instruction cache,summary 10
integer representation 5
interrupt mask 17
interrupt mode 16
interrupt priority 17
interrupt vectors 15
interrupts 15
  definition 15
  DSPCPU enable bit 4
INTVEC[31:0]
  picture 16
IPENDING
  picture 17
ISETTING0
  picture 16
ISETTING1
  picture 16
ISETTING2
  picture 16
ISETTING3
  picture 16
isub 122
isubi 123
izero 124

J

jmpf 125
jmpi 126
jmpt 127

L

latency,memory operation 9
ld32 128
ld32d 129
ld32r 130
ld32x 131
level sensitive interrupts 16
load store ordering 7
locking conditions 4
locking range 4
LRU bit definition 15
LRU bit definitions,picture 15
LRU bit update ordering 15
LRU initialization 15
LRU replacement,cache 14
LRU, hierarchical 4
LRU,four-way 15
LRU,two-way 14
lsl 132
lsli 133
lsr 134
lsri 135

M

MEM_EVENTS
  description table 18
  picture 17
memory
  operation ordering 8
memory map 11
  picture 11
memory mapped devices 11
mergelsb 137
mergemsb 138
misaligned
  store 4
miss processing,order 11
mmio 11
MMIO references,non-cached 9
MMIO registers
  BDATAAHIGH
    picture 12, 13, 21
  BDATAALOW
    picture 12, 13, 15, 21
  BDATAMASK
    picture 21
  BDATAVAL
    picture 12, 21
  BDCTL
    picture 21
  BICTL
    picture 21
  BINSTHIGH
    picture 21
  BINSTLOW
    picture 21
  cache registers summary 18
  DC_LOCK_ADDR

© TriMedia Technologies, Inc.

description table 18
DC_LOCK_CTL
description table 18
DC_LOCK_SIZE
description table 18
DC_PARAMS 2
description table 18
picture 3, 10
DRAM_BASE
description table 18
DRAM_CACHEABLE_LIMIT
description table 18
DRAM_LIMIT
description table 18
IC_LOCK_ADDR
description table 18
picture 13
IC_LOCK_CTL
description table 18
picture 13
IC_LOCK_SIZE
description table 18
picture 13
IC_PARAMS
description table 18
fields 10
ICLEAR
picture 17
IMASK
picture 17
INTVEC[31:0]
picture 16
IPENDING
picture 17
ISETTING0
picture 16
ISETTING1
picture 16
ISETTING2
picture 16
ISETTING3
picture 16
MEM_EVENTS
description table 18
picture 17
summary table 1
TCTL
picture 20
TMODULUS
picture 20
TVALUE
picture 20

N
non cacheable region 5
noncachable region 2
non-maskable interrupt 17
nop 139

O
offset byte in set 11
operation ordering,special 8
operations
DSPCPU 2
order,miss processing 11
ordering
memory operations 8
ordering,special operation 8

P
pack16lsb 140
pack16msb 141
packbytes 142
partial words 4
PCI references,non-cached 9
PCSW
definition 3
performance events,cache 17
pref 143
pref16x 144
pref32x 145
prefd 146
prefr 147

Q
quadavg 148, 149
quadumulmsb 150, 151
quasi-dual 4

R
rdstatus 152
rdstatus operation 7
result format picture 7
rdtag 153
rdtag operation 7
result format picture 7
readdpc 154
readpcsw 155
readspc 156
region
noncachable 2
region,non-cacheable 5
register model 1, 2
replacement 4
representation
boolean 5
floating point 5
integer 5
rol 157
roli 158

S
sex16 159
sex8 160
software compatibility 7
software interrupt 18
SPC
  definition 4
speculative loads 7
st16 161
st16d 162
st32 163
st32d 164
st8 165
st8d 166
stall,CPU 9
status operations,cache 7, 8
store
  misaligned 4

T
tag operations 7, 8
TCTL
  picture 20
TFE
  definition 4
timer 19
TMODULUS
  picture 20
TVALUE
  picture 20
two-way LRU 14

U
ubytesel 167
uclipi 168
uclipu 169
ueql 170
ueqli 171
ufir16 172
ufir8uu 173
ufixieee 174
ufixieeeflags 175
ufixrz 176
ufixrzflags 177
ufloat 178
ufloatflags 179
ufloatrz 180
ufloatrzflags 181
ugeq 182
ugeqi 183, 185
ugtr 184
uimm 186
uld16 187
uld16d 188
uld16r 189
uld16x 190
uld8 191
uld8d 192
uld8r 193
uleq 194
uleqi 195
ules 196
ulesi 197
ume8ii 198
ume8uu 199
umul 201
umulm 202
uneq 203
uneqi 204

V
vectored interrupts 15
victim of replacement 4

W
write misses 4
writedpc 205
writepcsw 206
writespc 207

Z
zex16 208
zex8 209

 © TriMedia Technologies, Inc.