

TriCore™

32-bit Unified Processor

DSP Optimization Guide

Part 1: Instruction Set

IP Cores



Never stop thinking.

Edition 2003-01

**Published by Infineon Technologies AG,
St.-Martin-Strasse 53,
D-81541 München, Germany**

**© Infineon Technologies AG 2003.
All Rights Reserved.**

Attention please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide (see www.infineon.com).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

TriCore™

32-bit Unified Processor

DSP Optimization Guide

Part 1: Instruction Set

DSP Optimization Guide

Revision History: **2003-01** v1.6.4


v1.6.4

Previous Version: v1.6.3

Page	Subjects (major changes since last revision)
All	Revised following internal review

We Listen to Your Comments

Is there any information within this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.

Please send your comments (referencing this document) to: 

ipdoc@infineon.com



Table of Contents	Page
1 General Information	16
1.1 Block Diagram Overview	17
1.2 Parallel Execution of Instructions	18
1.3 Programming Model	19
1.4 Memory Space	20
1.5 Little-endian	20
1.6 Interrupt Latency and Context-Switch	21
1.7 Fraction	21
1.8 Overflow and Sticky Overflow	21
1.9 Saturation	22
1.10 Advanced Overflow	23
1.11 Packed Arithmetic	24
1.12 16-bit Opcode	25
1.13 Notation	26
1.14 Document Conventions	27
1.14.1 TriCore Instruction Syntax	27
1.14.2 TriCore Instruction Characteristics	27
1.14.3 Examples	27
1.14.4 Symbolic Addresses	28
2 Simple Arithmetic	29
2.1 Addition	30
2.1.1 32-bit Addition (ADD, ADDS, ADDS.U)	30
2.1.2 32-bit Addition with 8, 16 & 32-bit Signed Constant (ADD, ADDI, ADDIH)	31
2.1.3 16-bit Addition	33
2.1.4 16-bit Addition using Packed Arithmetic (ADD.H, ADDS.H, ADDS.HU)	34
2.1.5 8-bit Addition using Packed Arithmetic (ADD.B)	35
2.1.6 64-bit Addition (ADDC, ADDX)	35
2.1.7 64-bit Addition with 64-bit Constant	36
2.1.8 Carry Flag Table	36
2.2 Subtraction	37
2.2.1 32-bit Subtraction (SUB, SUBS, SUBS.U)	37
2.2.2 32-bit Subtraction with Constant	37
2.2.3 16-bit Subtraction	38
2.2.4 16-bit Subtraction using Packed Arithmetic (SUB.H, SUBS.H, SUBS.HU)	39
2.2.5 8-bit Subtraction Using Packed Arithmetic (SUB.B)	39
2.2.6 64-bit Subtraction (SUBC, SUBX)	40
2.2.7 64-bit Subtraction with Constant	41
2.2.8 Reverse Subtraction (RSUB)	42
2.3 Negate	42
2.4 Move	43

Table of Contents	Page
2.4.1	Moves between Data Registers (MOV) 43
2.4.2	16-bit Constant into a Data Register (MOV.U, MOV, MOVH) 43
2.4.3	32-bit Constant into a Data Register 44
2.5	Applications 45
2.5.1	Arithmetic Series 45
2.5.2	Multi Byte Operations 46
3	Further Arithmetic 47
3.1	Absolute Value 47
3.1.1	32-bit (ABS, ABSS) 48
3.1.2	Packed 16-bit (ABS.H, ABSS.H) 49
3.1.3	Packed 8-bit (ABS.B) 49
3.2	Absolute Difference 50
3.2.1	32-bit Value (ABSDIF, ABSDIFS) 50
3.2.2	Packed 16-bit (ABSDIF.H, ABSDIFS.H) 51
3.2.3	Packed 8-bit (ABSDIF.B) 51
3.3	Minimum, Maximum 52
3.3.1	32-bit Signed/Unsigned (MIN, MIN.U, MAX, MAX.U) 52
3.3.2	Packed 16-bit Signed/Unsigned (MIN.H, MAX.H, MIN.HU, MAX.HU) . . 53
3.3.3	Packed 8-bit Signed/Unsigned (MIN.B, MAX.B, MIN.BU, MAX.BU) . . 53
3.4	Saturate 54
3.4.1	32-bit Saturation 54
3.4.2	16-bit and 8-bit Saturation (SAT.H, SAT.HU, SAT.B, SAT.BU) 54
3.5	Multiplication 55
3.5.1	32 * 32-bit Multiplication (MUL, MULS, MULS.U) 55
3.5.2	32 * 32-bit Multiplication with a 64-bit Result (MUL, MUL.U) 57
3.5.3	16 * 16-bit Multiplication 57
3.5.4	8 * 8-bit Multiplication 58
3.5.5	Multiplication by a Power of 2 58
3.5.6	Multiplication/Addition (MADD, MADDS) 58
3.5.7	Multiplication/Subtraction (MSUB, MSUBS) 59
3.6	Division 60
3.6.1	32 / 32-bit Division (DVINIT, DVSTEP, DVADJ) 60
3.6.2	16 / 16-bit Division (DVINIT.H, DVSTEP) 61
3.6.3	8 / 8-bit Division (DVINIT.B, DVSTEP) 61
3.6.4	32 / 16-bit Division 62
3.6.5	Division By a Power of 2 62
3.7	Applications 63
3.7.1	Find Maximum of Value in an Unsigned Array 63
3.7.2	Template Matching 63
3.7.3	40 * 40-bit Multiplication 65
3.7.4	Leapyear 66

Table of Contents	Page
4	Conditional Instructions
4.1	'Hidden' Conditional Instructions
4.2	Select (SEL, SELN)
4.3	Conditional Move (CMOV, CMOVN)
4.4	Conditional Add and Sub (CADD, CADDN, CSUB, CSUBN)
4.5	Application
4.5.1	Multiple If
5	Logical and Compare
5.1	Logical Instructions
5.1.1	NOT
5.1.2	AND, OR, XOR
5.1.3	NAND, NOR, ANDN, ORN, XNOR
5.2	Compare Instructions
5.2.1	Equal and Not Equal (EQ, NE)
5.2.2	Packed Equal (EQ.W, EQ.H, EQ.B)
5.2.3	Equal Any (EQANY.H, EQANY.B)
5.2.4	Less Than & Greater Than or Equal (LT, LT.U, GE, GE.U)
5.2.5	Packed Less Than (LT.B, LT.BU, LT.H, LT.HU, LT.W, LT.WU)
5.3	Compound Compare Instructions
5.3.1	Shift.Compare (SH.EQ, SH.NE, SH.GE, SH.GE.U, SH.LT, SH.LT.U)
5.3.2	Logical.Compare (AND.comp, OR.comp, XOR.comp)
5.4	Application
5.4.1	If... Else
6	Bit Field Operations
6.1	Logical Shift (SH)
6.1.1	Left and Right Shifts
6.1.2	Shift by a Variable Number
6.1.3	Packed Halfword Shift (SH.H)
6.2	Arithmetic Shift (SHA)
6.2.1	Left and Right Shifts
6.2.1.1	3 Differences between logical and arithmetic shifts
6.2.2	Shift by a Variable Number
6.2.3	Packed Halfword Arithmetic Shifts (SHA.H)
6.2.4	Arithmetic Shift with Saturation (SHAS)
6.3	64-bit Shift
6.3.1	64-bit Left Shift
6.3.2	64-bit Right Shift
6.3.3	64-bit Arithmetic Left Shift
6.3.4	64-bit Arithmetic Right Shift
6.3.5	64-bit Arithmetic Left Shift with Saturation
6.4	Insert (INSERT)

Table of Contents	Page
6.4.1 Corner Cases	95
6.4.2 Duplicate	95
6.4.3 Clear or Set a Bit Field	96
6.4.4 32-bit Shuffle	96
6.4.5 64-bit Interleave	97
6.5 Extract (EXTR , EXTR.U)	100
6.5.1 Corner Cases	101
6.5.2 Unpack 16-bit Values	101
6.6 Double Extract (DEXTR)	102
6.6.1 Corner Cases	103
6.6.2 Rotate	103
6.6.3 Swap	104
6.6.4 Normalization	104
6.7 Count Leading bits	105
6.7.1 Counting Zeros (CLZ, CLZ.H)	105
6.7.2 Counting Ones (CLO, CLO.H)	105
6.7.3 Counting Signs (CLS, CLS.H)	106
6.7.4 Corner Cases	106
6.8 Application Specific	107
6.8.1 Bit Merge & Split (BMERGE, BSPLIT)	107
6.8.2 Parity (PARITY)	108
6.9 Applications	109
6.9.1 Normalization	109
6.9.2 Left Shift Filled with 1s	109
6.9.3 Convolutional coder	110
6.9.4 Converting Little-endian to Big-endian	110
6.9.5 Counting High Bits	111
7 Boolean Processing	113
7.1 Bit Insert	114
7.1.1 Bit Insert (INS.T)	114
7.1.2 Bit Negate and Insert (INSN.T)	114
7.2 Single-operand Boolean Functions	115
7.2.1 Bit Invert	115
7.2.2 Bit Set	116
7.2.3 Bit Clear	116
7.3 2-Operand Boolean Functions	117
7.3.1 Bit and (AND.T)	117
7.3.2 Bit or (OR.T)	118
7.3.3 Bit nand, nor (NAND.T, NOR.T)	118
7.3.4 Bit andn, orn (ANDN.T, ORN.T)	118
7.3.5 Bit xor, xnor (XOR.T, XNOR.T)	118
7.3.6 Summary of Examples	119

Table of Contents	Page
7.4 2-Operand Boolean Functions (With Shift)	120
7.4.1 SH.op.T	120
7.5 3-Operand Boolean Functions	121
7.5.1 AND.op2.T	121
7.5.2 OR.op2.T	122
7.6 Summary Table: Five Instances of ANDing Bits	123
7.6.1 AND.T	123
7.6.2 SH.AND.T	123
7.6.3 OR.AND.T	124
7.6.4 AND.op.T	124
7.6.5 AND.comp	124
7.7 Applications	125
7.7.1 Logic Gates	125
7.7.2 Typical State Machine	126
7.7.3 Bit Testing	128
8 Pointer Arithmetic and Addressing Modes	129
8.1 Addressing Modes	130
8.1.1 Absolute Addressing	130
8.1.2 Extended Absolute Addressing	131
8.1.3 Base + Offset Addressing	132
8.1.4 Pre-Increment / Decrement Addressing	133
8.1.5 Post-Increment/Decrement Addressing	134
8.1.6 Circular Addressing	135
8.1.7 Bit-Reverse Addressing	136
8.1.8 Indexed Addressing (ADDSC.A)	138
8.1.9 PC-Relative Addressing	140
8.1.10 Bit Addressing	140
8.2 Data Load and Store	141
8.2.1 Load/Store Double Word (LD.D, ST.D)	142
8.2.2 Load/Store Word (LD.W, ST.W)	142
8.2.3 Load/Store Halfword (LD.H, LD.HU, ST.H)	142
8.2.4 Load/Store Halfword (LD.Q, ST.Q)	143
8.2.5 Load/Store Byte (LD.B, LD.BU, ST.B)	143
8.2.6 Swap (SWAP.W)	143
8.3 Bit Load and Store	144
8.3.1 Storing a bit (ST.T)	144
8.3.2 Storing a Bit Field (IMASK, LDMST)	145
8.3.3 Storing a Single Bit (ADDSC.AT)	146
8.3.4 Loading a Single Bit	147
8.4 Address Register Load and Store	148
8.4.1 Load/Store Address Register (LD.A, ST.A)	148
8.4.2 Load/Store Double Address Registers (LD.DA, ST.DA)	149

Table of Contents	Page
8.5	Address Arithmetic and Initialization 149
8.5.1	LEA (Initializing to a Segmented 18-bit Address) 149
8.5.2	Initializing to a 32-bit Address 150
8.5.3	Addition (ADD.A, ADDIH.A) 152
8.5.4	Subtraction (SUB.A) 153
8.5.5	Comparison (EQ.A, EQZ.A, GE.A, LT.A, NE.A, NEZ.A) 154
8.5.6	Move (MOV.AA) 155
8.5.7	Mixed Registers Moves (MOV.D, MOV.A) 155
8.6	Address Register as a GP Register 156
8.6.1	Loading to a 16-bit Value (MOVH.A) 156
8.7	Application 157
8.7.1	Find Index of Maximum Value in Array 157
9	Program Control and Context Switch 158
9.1	Unconditional Branches 158
9.1.1	Jump Absolute (JA) 159
9.1.2	Jump PC Relative (J) 160
9.1.3	Jump Indirect (JI) 161
9.2	Conditional Branches 162
9.2.1	Jump If (eq, ne, lt, le, ge, gt) Zero 162
9.2.2	Jump If (eq, ne, lt, lt.u, ge, ge.u) Small Constant 164
9.2.3	Jump If (eq, ne, lt, lt.u, ge, ge.u) Any Value 165
9.2.4	Jump On Bit (JZ.T, JNZ.T) 166
9.2.5	Jump On Address (JZ.A, JNZ.A, JEQ.A, JNE.A) 167
9.3	Loops 168
9.3.1	Loop Using Jump Non Equal Decrement/Increment JNED/JNEI 168
9.3.2	Zero Overhead Loop (LOOP) 169
9.3.3	Infinite Loop (LOOPU) 170
9.4	Subroutine Calls 171
9.4.1	Jump and Link (JL, JLA, JLI) 171
9.4.2	Call, Return (CALL, CALLA, CALLI, RET) 172
9.5	Context Switch 175
9.5.1	Context Saving and Restoring (SVLCX, RSLCX) 175
9.5.2	Context Loading and Storing (STUCX, LDUCX, STLCX, LDLCX) . . . 176
9.6	Applications 178
9.6.1	Arithmetic Equation Using Subroutine Call 178
9.6.2	Sweet64 179
9.7	Summary Table: Displacement Field Width 181
10	DSP and MAC 183
10.1	Introduction to DSP Arithmetic 184
10.1.1	Fractional Arithmetic 184
10.1.2	Multiplication Problem: Word Growth 187

Table of Contents	Page
10.1.3	Multiplication Problem: Signed Multiplication 189
10.1.4	Q-Format 190
10.1.5	Left-Alignment 193
10.1.6	Multiplication Saturation 193
10.1.7	Rounding and Multi-Precision 194
10.1.8	Growth Bits 195
10.1.9	32-bit DSP 195
10.2	Multiplication 196
10.2.1	16*16-bit \pm 32 Multiplication (MUL.Q) 196
10.2.2	16*32 \pm 32 Multiplication (MUL.Q) 197
10.2.3	32*32 \pm 32 Multiplication (MUL.Q) 198
10.2.4	16*32 \pm 48 Multiplication (MUL.Q) 199
10.2.5	32*32 \pm 64 Multiplication (MUL.Q) 200
10.2.6	16*16 \pm Round(32) Multiplication (MULR.Q) 201
10.3	Multiply-Add 203
10.3.1	32 + 16*16 Multiply-Add (MADD.Q, MADDS.Q) 203
10.3.2	32 + 16*32 Multiply-Add (MADD.Q, MADDS.Q) 205
10.3.3	32 + 32*32 Multiply-Add (MADD.Q, MADDS.Q) 207
10.3.4	48 + 16*16 Multiply-Add (MADD.Q, MADDS.Q) 208
10.3.5	64 + 16*32 Multiply-Add (MADD.Q, MADDS.Q) 210
10.3.6	64 + 32*32 Multiply-Add (MADD.Q, MADDS.Q) 212
10.3.7	32 + 16*16 \pm Round(32) Multiply-Add (MADDR.Q, MADDRS.Q) 213
10.4	Multiply-Subtract (MSUB.Q, MSUBR.Q) 214
10.5	Multiply-Accumulate 215
10.6	Applications 216
10.6.1	Square Root 216
10.6.2	Sine (coefficient mixed 1q31 and 3q29, result in 1q31) 218
10.6.3	Sine of a Value (coefficient in 2q14, result in 1q31) 220
10.6.4	Sine of a Value (coefficient in 1q15 Mixed with 2q14, result in 1q31) . 220
10.6.5	Sine of a Value (coefficient in 1q15, result in 1q31) 220
10.6.6	Sine of a Value (coefficient in 2q14, result in 1q31 with saturation) . . 221
10.6.7	Sine of a Value (coefficient in 1q15, result in 1q31 with saturation) . . 221
10.6.8	Sine of a Value (coefficient in 1q15, result in 1q15 with truncation) . . 221
10.6.9	Sine of a Value (coefficient in 1q15, result in 1q15 with rounding) . . . 222
10.6.10	Sine of a Value (coefficient in 1q15, result 1q15 with rounding) 222
10.6.11	Sine of a Value: Comparing Precision 222
10.7	Summary Tables 224
11	DSP and Dual MAC 226
11.1	Notation 226
11.2	Dual MAC Structure 229
11.2.1	Block Diagram 229
11.2.2	Differences between Packed & Merged/Packed 231

Table of Contents	Page
11.3 Packed Dual-MAC Instructions (MUL, MADD)	232
11.3.1 Packed Multiply (MUL.H)	232
11.3.2 Packed Multiply-Add (MADD(S).H)	234
11.3.3 Packed Multiply-Subtract (MSUB(S).H)	236
11.3.4 Packed Multiply-Add/Subtract (MADDSU(S).H, MSUBAD(S).H)	237
11.3.5 Packed Multiply - Round (MULR.H)	240
11.3.6 Packed Multiply-Add-Round (MADDR(S).H)	241
11.3.7 Packed Multiply-Subtract-Round (MSUBR(S).H)	242
11.3.8 Packed Multiply-Add/Sub-Round (MADDSUR(s).H, MSUBADR(s).H) .	243
11.4 Merged/Packed MAC Instructions (MUL, MADD)	244
11.4.1 Merged Multiplication (MULM(S).H)	245
11.4.2 Merged Multiply-Add/Subtract (MADDM(S).H, MSUBM(S).H, MADDSUM(S).H, MSUBADM(S).H)	248
11.5 Applications	253
11.5.1 Dot Product	253
11.5.2 Vector Multiplication	255
11.5.3 Matrix Product	256
11.6 Summary Tables	259
11.6.1 Dual MAC Operation Differences	259
11.7 List of Instructions	259
12 Floating-Point Help	262
12.1 Floating Point Packing / Unpacking (PACK, UNPACK)	262
12.1.1 Using Pack and Unpack	262
13 Pipeline Operations	264
13.1 General Rules	265
13.1.1 One Cycle	265
13.1.2 One Cycle - Dual Issue	265
13.1.3 Table of Dual Issue Instructions	267
13.2 16x16-bit and 16x32-bit MAC Instructions	268
13.2.1 One Cycle –Rule 1	268
13.2.2 One Cycle –Rule 2	268
13.2.3 One Cycle - Dual Issue	269
13.2.4 Two Cycles – Rule 1	269
13.2.5 Two Cycles – Rule 2	270
13.2.6 Two Cycles – Rule 3	270
13.3 32x32-bit MAC Instructions	271
13.3.1 Two Cycles – Rule 1	271
13.3.2 Two Cycles – Rule 2	271
13.3.3 Three Cycles – Rule 1	272
13.3.4 Three Cycles – Rule 2	272
13.3.5 Three Cycles – Rule 3	272

Table of Contents	Page
13.4	Dependency Table 273
13.5	Loop 274
13.6	Branch 274
13.6.1	Counting Cycles for Branch Instructions 274
13.7	Applications 276
13.7.1	FIR 276
13.7.2	Vector Quantization 276
13.7.3	Complex Wave Generation 277
13.7.4	FFT - DIT – Butterfly 278
14	Instruction Set Summary 279
14.1	List of Instructions 281
14.1.1	Data Move 281
14.1.2	Add, Sub, Rsub 281
14.1.3	Abs, Absdif 282
14.1.4	Min, Max, Sat 282
14.1.5	Mul (Integer) Multiplication 283
14.1.6	Madd, Msub (Integer) MAC 283
14.1.7	Division 283
14.1.8	Sel, Cmov, Cadd, Csub 284
14.1.9	Logical 284
14.1.10	Compare 285
14.1.11	Shift 287
14.1.12	Extract, Insert 288
14.1.13	Count Leading 289
14.1.14	Bit split; Bit Merge; Parity 289
14.1.15	Boolean Processing 290
14.1.16	Bit Store, Load/Modify/Store 292
14.1.17	Memory Load / Store 293
14.1.18	Address Arithmetic 294
14.1.19	Jump and Loop 295
14.1.20	Call, Return, Context Switch 296
14.1.21	Q-format MUL 297
14.1.22	Q-format MAC 297
14.1.23	Q-format Dual MUL 298
14.1.24	Q-format Dual MAC 299
14.1.25	Floating Point Help 301
14.1.26	Cache Management 301
14.1.27	MMU Management 301
14.1.28	Exceptions (Interrupts, Traps, System Calls, Debug) 302
14.1.29	Core Register and PSW Management 302
14.1.30	Pipeline Control 302
14.1.31	16-bit Opcode Instructions 303

Table of Contents	Page
14.1.32 Load & Store Instructions	304
14.1.33 Branch Instructions	305
14.1.34 Packed Arithmetic Instructions	306
14.1.35 Constants	308
14.1.36 Instructions Modifying a 64-bit Data Register	310
15 TriCore 2	311
15.1 Index Addressing Mode	312
15.2 New Instructions	312
15.2.1 MOV <E register> - Move 64-bit	313
15.2.2 XPOSE.H, XPOSE.B	313
15.2.3 LD.DD, ST.DD - Load/Store 128-bit	314
15.2.4 SWAPMSK - Semaphore Management	315
15.2.5 JNE,JEQ with 5-bit Branch Offset	315
15.2.6 FCALL, FCALLA, FCALLI - Fast Call Context	315

Foreword

The Infineon TriCore Unified Processor was designed with the aim of unifying of 32-bit CPU (Central Processing Unit), MCU (MicroController Unit) and DSP (Digital Signal Processor).

What differentiates one processor type from another are the different hardware mechanisms: MCUs are strong on interrupt, DSPs have very powerful arithmetic units, and CPUs take special care of the system and rely on memory protection.

Another major difference is the instruction set. DSPs for instance have a large number of multiplications and processing instructions. MCUs concentrate on bit manipulation, while CPUs have floating point, cache and memory management instructions. As the first Unified processor, TriCore therefore has a large instruction set.

This Optimization Guide is organized into chapters to describe the different types of instruction. These include:

- Arithmetic objects (byte, short, long, long_long, and signed/unsigned)
- Logical operations
- Bit fields
- Operations on bits (or booleans)
- Pointers
- Program control
- DSP instructions
- Pipeline operations

Each chapter starts with a description of basic operations before going into more detail about those operations. The use of specific instructions and examples are provided to give a better understanding of the applications.

All the assembly examples are implemented with the TASKING Tools for TriCore version 1.4r1. Each chapter (except for chapter 1) has a corresponding source file.

An Instruction Set summary is given at the end of this document.

1 General Information

This chapter provides general background information on the TriCore Architecture, that is useful for all subsequent chapters. Included here is an overview of the TriCore architecture, explanation of the programming models and a presentation of the parallel-issued instructions. Further details on little-endian, overflow, saturation, fraction and packed arithmetic are also provided.

Chapter contents:

- **Block Diagram Overview**
- **Parallel Execution of Instructions**
- **Programming Model**
- **Memory Space**
- **Little-endian**
- **Interrupt Latency and Context-Switch**
- **Fraction**
- **Overflow and Sticky Overflow**
- **Saturation**
- **Advanced Overflow**
- **Packed Arithmetic**
- **16-bit Opcode**
- **Notation**
- **Document Conventions**

1.1 Block Diagram Overview

TriCore integrates DSP and MCU real-time embedded functionality into one RISC-based 32-bit CPU core. The TriCore block diagram of the core consists of an Integer Processing (IP) unit, a Load/Store unit (which is also responsible for address generation), a fetch unit and several system registers (CSFRs). The core does not contain any memory, but includes two local bus sets intended for on-chip program and data memory. An additional bus, the Flexible Peripheral interface (FPI) bus, serves as interface to peripherals and external memory.

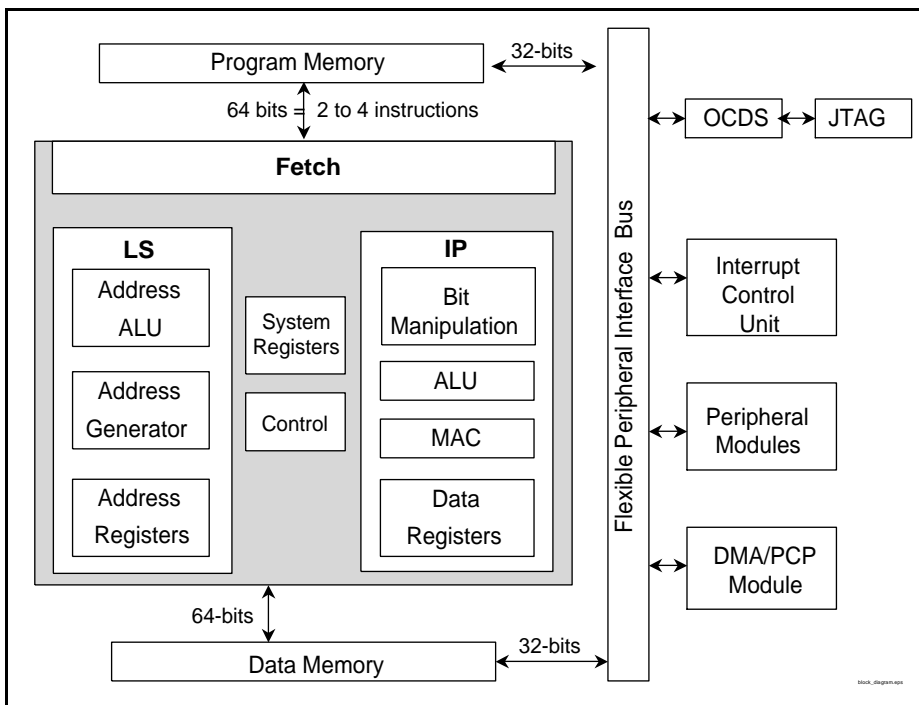


Figure 1 TriCore Block Diagram Overview

1.2 Parallel Execution of Instructions

TriCore is capable of executing two instructions in a single cycle, provided the first instruction is issued to the Integer Processing (IP) unit, followed by an instruction issued to the Load/Store unit. It is therefore referred to as a *dual-issue machine*.

Note: The Alignment Rules for dual-issue fetch and execution are implementation defined. However, in general, alignment on a 64-bit boundary will ensure that both instructions are fetched at the same time.

Dual issue

Arithmetic	Load/store
Execution Slot 1	Execution Slot 2

Examples

Arithmetic	Load/store
<code>add d0,d1,d2</code>	<code>ld.w d9,[a0]</code>
<code>sh d6,d9,#-3</code>	<code>st.w [a5],d6</code>

Note: TriCore can be considered as a triple-issue machine, since a third instruction (LOOP) can be executed in parallel. The general case however is that TriCore can execute two instructions simultaneously. This significant feature offers new opportunities to the assembler programmer.

1.3 Programming Model

The TriCore register set consists of 32 general purpose, 32-bit registers, divided into 16 address registers (A0 through A15) and 16 data registers (D0 through D15).

Three special purpose 32-bit registers are also part of the model:

- **PC (Program Counter)**
Contains the address of the instruction currently executed.
- **PSW (Program Status Word)**
Divided into two parts:
 - The five most-significant bits contain ALU status flags that are set and cleared by arithmetic instructions.
 - The remaining bits control the permission levels, the Protection Register Sets and the Call Depth Counter (CDC).
- **PCXI (Previous Context Information)**
Contains linkage information on the previous execution context, supporting fast interrupts and automatic context switching.

Note: Please refer to the TriCore Architecture Manual for complete descriptions.

Address Registers (32-bit)	Data Registers (32-bit)	System Registers (32-bit)
A15	D15	PCXI
A14	D14	PSW
A13	D13	PC
A12	D12	
A11	D11	
A10	D10	
A9	D9	
A8	D8	
A7	D7	
A6	D6	
A5	D5	
A4	D4	
A3	D3	
A2	D2	
A1	D1	
A0	D0	

Note: Two consecutive, even-odd data registers can be concatenated to form eight extended-size registers (E0, E2, E4..., E14), supporting 64-bit values.

Example:

e0 is the concatenation of d1 and d0

d1	88886666	
	d0	abcd1111
e0	88886666abcd1111	

1.4 Memory Space

The architecture supports a uniform, 32-bit address space, with memory-mapped I/O. All addresses are bytes, therefore giving 4GBytes.

Example:

```
ld.a          a0, 0x12345678
```

a0 points to memory address 0x12345678 (in bytes)

```
ld.w          d0, d0, [a0+]6
```

d0 is loaded with 4 consecutive memory addresses 0x12345678, 9, A, B (in bytes) and a0 is post incremented by 6 bytes.

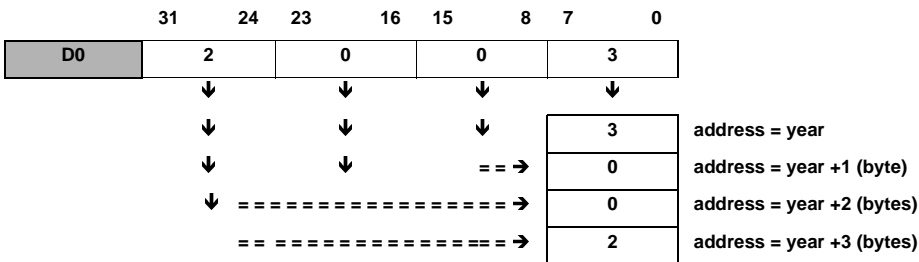
1.5 Little-endian

Little-endian means that a 32-bit number is stored with the *least significant byte* at the lower address. TriCore (as an architecture model), is exclusively little-endian.

Example:

Store register d0 at memory address “year”

```
st.w    year, d0
```



1.6 Interrupt Latency and Context-Switch

Interrupt latency and context-switch time largely determine real-time responsiveness. The high-performance architecture minimizes interrupt latency by avoiding long multi-cycle instructions and by providing a flexible hardware-supported interrupt scheme. Furthermore, the architecture supports fast context switching by automatically saving a subset of the registers on a call, interrupt, or trap, and restoring them on a return.

1.7 Fraction

A fraction is the preferred method for mathematical and algorithm developers to map numbers onto computer arithmetic. The main purpose is to keep the same order of magnitude between input and output signals in multiplication.

For this a binary point is imagined between the Most Significant Bit and the Most Significant Bit-1 bit.

In the following 4-bit example, 0b0100 equals to +0.5 and 0b1100 is equal to -0.5.

Example:

-2^0 $+2^{-1}$ $+2^{-2}$ $+2^{-3}$ **weight of each bit in power of 2 representation**

A	B	C	D
---	---	---	---

-1 $+1/2$ $+1/4$ $+1/8$ **weight of each bit in fractional representation**

A	B	C	D
---	---	---	---

0	1	0	0	$= 0 * (-1) + 1 * (1/2) + 0 * (1/4) + 0 * (1/8) = +0.5$
1	1	0	0	$= 1 * (-1) + 1 * (1/2) + 0 * (1/4) + 0 * (1/8) = -0.5$

1.8 Overflow and Sticky Overflow

An overflow can be seen as the 33rd bit of a 32-bit result (or the 17th bit of a 16-bit result). The overflow is set to 1 if the result of an operation cannot be represented in the data size of the result. This definition is applied for most of the instructions that update the overflow flag. If the overflow occurs because of another case, it will be specified with the instruction.

Sticky overflow is set at the same time as overflow and only the RSTV (Reset Overflow bits) instruction can reset it. A sticky flag is particularly useful on a block of data. The user can check to see if an overflow has occurred, without continuously monitoring the overflow flag.

1.9 Saturation

Saturation of a result is the result of an overflow. TriCore has instructions with and without saturation (add, shift). They only differ in case of overflow. For instructions with built-in saturation, the result will be turned into the maximum value. The maximum value depends on the values being signed or unsigned:

Example: 16-bit

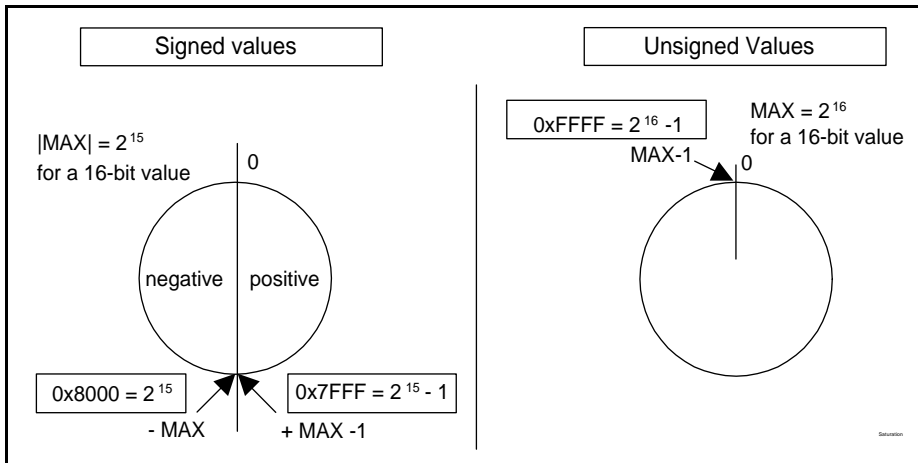


Figure 2 Saturation 16-bit Example

Signed

$MAX = 2^{15}$. The result is set to $(+ MAX - 1)$ or $(-MAX)$

Unsigned

$MAX = 2^{16}$. The result will be $(+ MAX - 1)$ or 0.

1.10 Advanced Overflow

Overflow can be seen as the XOR of bit 32 and bit 31 of a 32-bit register, whereas advanced overflow is the result of an XOR between the two MSBs (bit 31 and bit 30) of the same register. The advanced overflow flag alerts to an overflow, allowing it to be avoided (by using a shift for example).

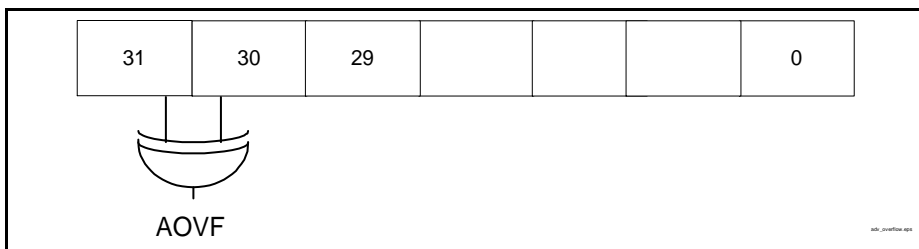


Figure 3 Advanced Overflow

Examples:

For 0x34f245e9, AOVV = 0

For 0x44f245e9, AOVV = 1

For 0x94f245e9, AOVV = 1

The range of advanced overflow is represented (in the diagram below) on this fractional scale:

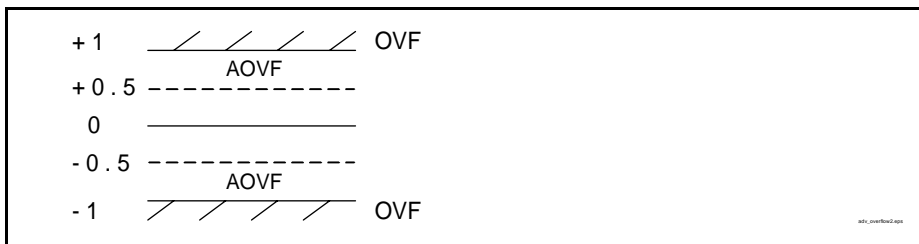


Figure 4 Advanced Overflow Fractional Scale Example

1.11 Packed Arithmetic

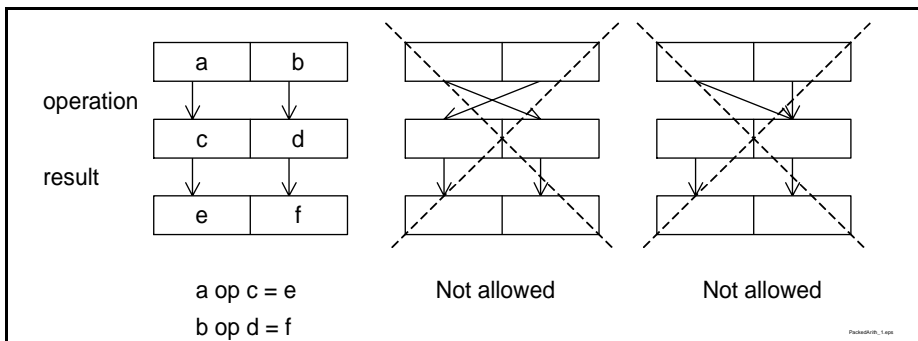
Packed instructions partition a 32-bit word into several identical sub-words, which can then be loaded, operated and stored in parallel. Packed instructions allow for the full exploitation of 32-bit word architecture in signal and data processing applications, since most signal/data is either 16-bit or 8-bit wide.

The TriCore architecture supports both 16 and 8-bit widths. For:

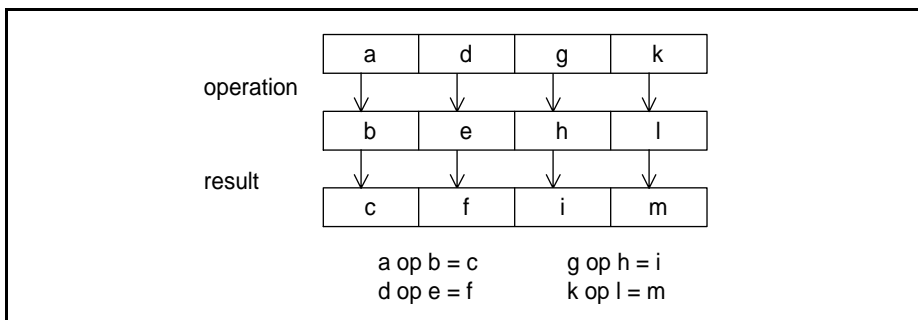
- **.H** (halfword): Allows TriCore to work on 2 values at the same time (Divides the 32-bit word into two 16-bit (half-word) values).
- **.B** (bytes): Allows TriCore to work on 4 values at a time (Divides the 32-bit word into four 8-bit values).

A packed operation will carry out the same operation on multiple data in parallel. This is why it is also called Single Instruction Multiple Data (SIMD). In other words, recombination between parallel data is not allowed.

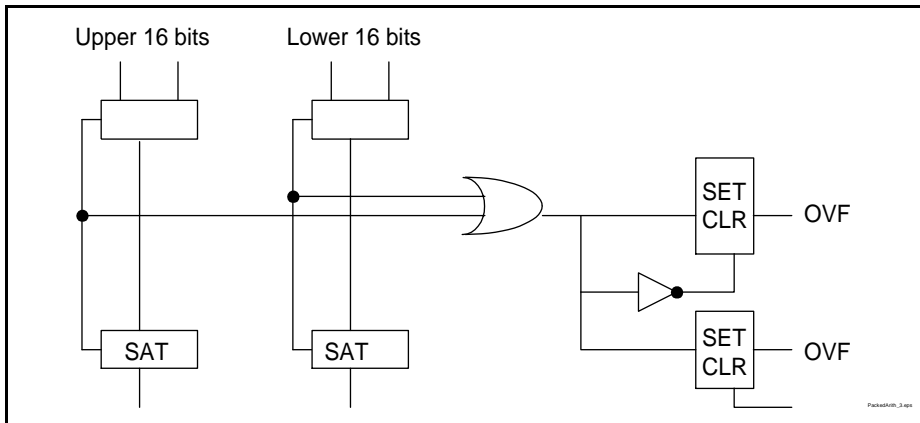
The following example illustrates 16-bit data packed operation:



A byte packed operation is executed as follows:



In packed operations, the resulting flags are an OR of each operation. Overflow and saturation units are activated separately, as shown in the example for halfword below.



1.12 16-bit Opcode

The TriCore instruction set has a mix of 32-bit and 16-bit opcode instructions. The most frequently used instructions are given 16-bit opcode. Some 16-bit opcodes make use of implied operands, where the destination register is fixed. All 16-bit opcode have a 32-bit equivalent.

The reduction of the code size leads to a reduction of instruction code space by an average of one third or more. Code space reduction lowers memory requirements, system cost and power consumption.

1.13 Notation

TriCore instructions have specific mnemonic notations, shown grouped in the following table:

.	IP Instruction	LS Instruction
(S).	Saturation instruction	-
.A	-	Address instruction
.B	Byte packed instruction	Load/store byte
.BU	Unsigned byte packed instruction	Load unsigned byte
.D	-	Load/store double word
.H	Halfword packed instruction	Load/store least significant halfword
.HU	Unsigned halfword packed instruction	Load unsigned halfword
.Q	Q format instruction	Load/store most significant halfword
.T	Bit instruction	-
.U	Unsigned instruction	-
.W	Word instruction	Load/store word
.WU	Unsigned word instruction	-

1.14 Document Conventions

1.14.1 TriCore Instruction Syntax

TriCore syntax convention is for the DESTINATION to appear as the first operand:

`ADD destination, source1, source2...`

For 16-bit opcode it is often the case that the destination and source1 are the same:

`ADD destination/source1, source2`

In this document the formal syntax is only shown for 16-bit opcode. For 32-bit opcode an example is used instead. In the TriCore instruction set any registers can replace any other registers used in the example (Any exceptions to this are indicated).

1.14.2 TriCore Instruction Characteristics

An instruction is referenced by its mnemonic, but there are additional characteristics of interest:

1. On which arithmetic condition is the Overflow flag set ?
2. Is the instruction available with a constant (instead of a variable) ?
3. Is there a 16-bit opcode available ?

Such questions are answered by the appearance of the following keywords in a description:

<i>OVF:</i>	Overflow flag is set by this condition
<i>Constant:</i>	Constant type is available
<i>16-bit opcode syntax</i>	16-bit opcode is available & the syntax is given

1.14.3 Examples

Examples are given with the register result appearing alongside. The format is (in general) hexadecimal:

`mov d0, #1`

do	0000 0001
----	-----------

1.14.4 Symbolic Addresses

Symbolic addresses are used to make examples independent of implementation:

load word from address lx5 (lx5 is a label)

```
ld.w d1, lx5
```

d1	EB5F 7F11
----	-----------

The following types of labels are used:

Address label begins with:	Description (i.e. Address label references...)
eX...	The address of a 64-bit value
IX...	The address of a 32-bit value
sX...	The address of a 16-bit value
bX...	The address for a 8-bit value
lIX...	The address of 2 consecutive 32-bit values
ssX...	The address of 2 consecutive 16-bit values
ssssX...	The address of 4 consecutive 16-bit values

The convention $[-n; +n]$ is used for bit ranges.

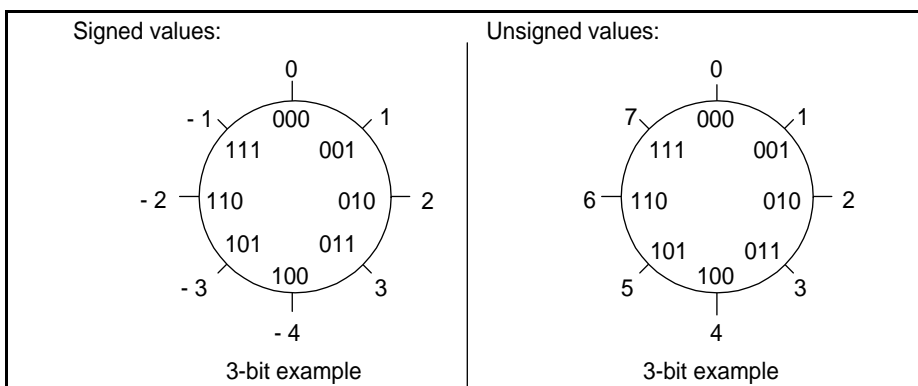
For example, $[-4; +4]$ represents the range -4 to +3, where [is inclusive and) is exclusive.

2 Simple Arithmetic

This chapter covers universal arithmetic functions such as, add, subtract and negate. TriCore has a standard data size of 32-bit and all arithmetic operations refer to 32-bit (word) values. However 16-bit (halfword), 8-bit (byte) and 64-bit (double word) are as important as the standard data size. The chapter covers all basic arithmetic functions operating on the four different data sizes (8, 16, 32 & 64).

Signed & Unsigned Numbers

Numbers are classified in two categories: signed and unsigned. The unsigned numbers are all positive, whereas signed values can be positive or negative.



The range of each category is different for the same number of used bits: In this example the signed values belong to $[-4; +4)$, and the unsigned values belong to $[0; +8)$. In general, all basic arithmetic functions do not differentiate between signed and unsigned variables (a property of 2's complement). However, there are exceptions:

- Operation with saturation
- Operation with constants: in case those constants are smaller than 32-bit, they need extension. They are sign-extended in case of signed constant (respectively zero extended for unsigned constants).
- Extension in sub-word moves.

Note: In this chapter signed and unsigned are only differentiated when required.

Chapter Contents:

- **Addition**
- **Subtraction**
- **Negate**
- **Move**
- **Applications**

2.1 Addition

Several addition types are described for signed or unsigned:

- 32-bit, 32-bit with constant
- 16-bit, packed 16-bit
- 8-bit, packed 8-bit
- 64-bit

2.1.1 32-bit Addition (ADD, ADDS, ADDS.U)

Adding two 32-bit operands can be implemented in three ways.

The following instruction produces a normal result:

```
add d0,d1,d2
```

The following two instructions saturate the result in case of overflow:

```
adds d0,d1,d2 ; signed
adds.u d0,d1,d2 ; unsigned
```

Signed saturation is 0x7FFFFFFF for positive values and 0x80000000 for negative. Unsigned saturation is 0xFFFFFFFF.

OVF: The result of the addition cannot be represented in a 32-bit register.

Constant: An 8-bit constant can replace the second source register (d2 in the example).

16-bit opcode Syntax:

```
add dest/src1,src2
add d15,src1,src2
add dest,d15,src2
add dest/src,k4
add d15,src,k4
```

Example:

```
add d0,d1
add d15,d0,d1
add d0,d15,d2
add d0,#2
add d15,d0,#4
```

2.1.2 32-bit Addition with 8, 16 & 32-bit Signed Constant (ADD, ADDI, ADDIH)

Adding an 8-bit Signed / Unsigned Constant

The syntax of an addition of an 8-bit constant is similar to the addition of 2 variables.

Examples:

```
add  d0,d1,d2           ; normal addition d0 = d1+d2
add  d0,d1,#0x21        ; d0 = d1+33
add  d0,d1,#-0x100      ; d0 = d1-256
add  d0,xd1,#0xd2       ; d0 = d1+0x000000d2
```

Note: In the last example it appears as if the constant is zero-extended, when it would be expected to be sign-extended. The 8-bit constants are in reality 9 bits. Therefore in the example, 0xd2 is interpreted as 0x0d2. The sign will always be zero and the constant will be correctly zero extended.

Note: Some assemblers will only accept 32-bit sign-extended constants. To add an 8-bit negative value, it must be sign extended to 32-bit.

```
add  d0,d1,#0xFFFFF00    add  -256
```

Adding a 16-bit Signed Constant

The ADDI instruction is used for a 16-bit signed constant:

```
addi  d0,d1,#0x9876
```

Adding a 16-bit Unsigned Constant

ADDI can also be used as long as the unsigned constant is smaller than 0x7FFF (i.e. the unsigned constant is a 16-bit number that can not be larger than 15bits in size).

```
addi  d0,d1,#0x6598
```

Because there is no ADDI.U instruction, if the constant is bigger than 0x7FFF, the addition must be synthesized with 2 instructions: a MOV.U instruction followed by a simple ADD.

```
mov.u  d0,#0x875A
add    d1,d0,d2
```

Example:

```
mov d0,#1
```

d0	0000 0001
----	-----------

After Instruction

```
;case 1
addi d0,d0,#0x1110
```

d0	0000 1111
----	-----------

```
;case 2
mov.u d2,#0x8888
add d1,d0,d2
```

d2	0000 8888
d1	0000 9999

Adding a 32-bit Signed / Unsigned Constant

Adding a 32-bit constant is synthesized with ADDI followed by an ADDIH. This will only work if the lower part of the 32-bit value is smaller than 0x7FFF.

If the lower part of the 32-bit value is greater than 0x7FFF, a 3 instruction sequence using MOV.U, ADDIH and ADD is used.

Example:

```
mov d0,#1
```

d0	0000 0001
----	-----------

```
mov d1,#1
```

d1	0000 0001
----	-----------

After Instruction

```
; case 1
addi d0,d0,#0x5677
addih d0,d0,#0x1234
```

d0	0000 5678
d0	1234 5678

```
; case 2
mov.u d2,#0x8641
addih d2,d2,#0x1357
add d1,d1,d2
```

d2	0000 8641
d2	1357 8641
d1	1357 8642

OVF: The result of the addition cannot be represented in a 32-bit register.

2.1.3 16-bit Addition

Adding two 16-bit signed numbers into a 32-bit register can be more involved than adding 2, 32-bit values.

In this example the addition instruction is identical to a 32-bit addition:

```
ld.h d1,sX1      ; load 16bit value into lower half
ld.h d2,sX2      ; load 16bit value into lower half
add d0,d1,d2     ; add the 2 numbers
```

If the result is within range [8000, 7FFF] there is no problem, otherwise we might wish to use saturation:

```
adds d3,d1,d2
```

But the number will never saturate since it has 16 bits of guard bits. These bits are the upper half of the register and prevent overflow detection. If a carry occurs in the 16-bit addition, it will be transferred to the LSB of the upper half (bit 16).

A solution is to use the SAT instruction. A second solution is to load the 16-bit number in the upper part of the register. Both solutions are shown in the following example:

Example:

After Instruction

```
ld.h d1,sX1
ld.h d2,sX2
add d0,d1,d2
adds d3,d1,d2
```

d1	0000 7F11
d2	0000 7F54
d0	0000 FE65
d3	0000 FE65

;solution 1

```
ld.h d1,sX1
ld.h d2,sX2
add d0,d1,d2
sat.h d0,d0
```

d1	0000 7F11
d2	0000 7F54
d0	0000 FE65
d0	0000 7FFF

;solution 2

```
ld.q d1,sX1
ld.q d2,sX2
```

d1	7F11 0000
d2	7F54 0000

```
adds d4,d1,d2
```

d4	7FFF FFFF
----	-----------

```
;by comparison this
is the overflowed
result
```

```
add d5,d1,d2
```

d5	FE65 0000
----	-----------

2.1.4 16-bit Addition using Packed Arithmetic (ADD.H, ADDS.H, ADDS.HU)

The use of packed arithmetic allows the addition of two 16-bit additions in one cycle.

Normal instruction:

```
add.h d5,d1,d2
```

Saturated versions:

```
adds.h d6,d1,d2 ;add with 16-bit signed saturation
adds.hu d5,d1,d2 ;add with 16-bit unsigned saturation
```

Example:

```
ld.w d1,ssX3
```

d1	1234 F678
----	-----------

```
ld.w d2,ssX4
```

d2	7B6F F71F
----	-----------

```
add.h d5,d1,d2
```

d5	8DA3 ED97
----	-----------

```
adds.h d6,d1,d2
```

d6	7FFF ED97
----	-----------

```
adds.hu d6,d1,d2
```

d6	8DA3 FFFF
----	-----------

OVF: The result of the addition cannot be represented in a 16-bit register.

2.1.5 8-bit Addition using Packed Arithmetic (ADD.B)

The use of packed arithmetic allows for the computing of four 8-bit additions in one cycle.

`add.b d0, d1, d2`

This is the only available instruction. There is no saturated case.

Example:

<code>ld.w d1, ssX3</code>	d1	12 34 F6 78
<code>ld.w d2, ssX4</code>	d2	7B 6F F7 1F
After Instruction		
<code>add.b d0, d1, d2</code>	d0	8D A3 ED 97

OVF: The result of the addition *cannot* be represented in an 8-bit register.

2.1.6 64-bit Addition (ADDC, ADDX)

The 64-bit addition is synthesized using two specific 32-bit additions: ADDX (add extended) and ADDC (add with carry). Both instructions modify the carry flag (psw.c). Only the ADDC instruction will take account of it in its addition.

Example:

After Instruction		
	e0	3B5F 004F F01B 5FE0
	e2	0123 4567 89AB C8EF
<code>addx d4, d0, d2</code>	e4	xxxx xxxx 79C7 28CF
		psw.c = 1
<code>addc d5, d1, d3</code>	e4	3C82 45B7 79C7 28CF
		psw.c = 0

The result is in register e4, a short hand notation for d5/d4.

d5/d4 means a concatenation of the two consecutive registers. The even register is the lower 32-bit, the odd register is the upper 32-bit.

OVF: The result of the addition cannot be represented in a 32-bit register.

Constant: An 8-bit constant can replace the second source register (d2 and d3 in the example).

2.1.7 64-bit Addition with 64-bit Constant

64-bit addition with a constant is a combination of multi-precision and 32-bit addition with a constant. If one of the lower 16-bit halves of the 64-bit constant is bigger than 0x7FFF, replace the ADDI instruction with the MOV.U instruction.

Example:

The 64-bit constant is in registers d0 and d1 (0x11111000 98765432). The 64-bit value is in registers d4 and d5 (0x7BFF7F10 EFFF5AE0). The result is in registers d6 and d7.

After Instruction		
<code>mov d9, #0</code>	d9	0000 0000
<code>ld.d e4, eX0</code>	e4	7BFF 7F10 EFFF 5AE0
<code>addi d0, d9, #0x5432</code>	d0	0000 5432
<code>addih d0, d0, #0x9876</code>	d0	9876 5432
<code>addx d6, d4, d0</code>	d6	8875 AF12
<code>addi d1, d9, #0x1000</code>	d1	0000 1000
<code>addih d1, d1, #0x1111</code>	d1	1111 1000
<code>addc d7, d5, d1</code>	d7	8D10 8F11

psw.c = 1

2.1.8 Carry Flag Table

The table which follows shows the 4 TriCore arithmetic instructions which touch / use the carry flag:

Instruction	Use Carry	Touch Carry
<code>add.c</code>	yes	yes
<code>add.x</code>	no	yes
<code>sub.c</code>	yes	yes
<code>sub.x</code>	no	yes

Note: SHA is an exception, which uses the carry flag in a special way.

2.2 Subtraction

Several subtraction types are described for signed or unsigned:

- 64-bit
- 32-bit
- 16-bit
- 8-bit

2.2.1 32-bit Subtraction (SUB, SUBS, SUBS.U)

There is a choice of 3 instructions to subtract two 32-bit operands.

For a normal result use:

```
sub    d0,d1,d2
```

The following two instructions saturate the result in case of overflow:

```
subs   d0,d1,d2 ;signed
```

```
subs.u d0,d1,d2 ;unsigned
```

Signed saturation is 0x7FFFFFFF for positive results and 0x80000000 for negative.

Unsigned saturation is 0x00000000.

OVF: The result of the subtraction cannot be represented in a 32-bit register.

Constant: NA (cf. 32-bit subtraction with constant)

16-bit opcode syntax:

```
sub    dest/src1,src2
```

```
sub    d15,src1,src2
```

```
subs   dest/src1,src2
```

2.2.2 32-bit Subtraction with Constant

Subtracting a constant from a 32-bit register *does not use the SUB instruction*, since the SUB instruction does not accept a constant. The ADD instruction executes both signed and unsigned values.

For an 8-bit constant:

```
add    d2,d1,#0xFFFFFEC    ;add d1 and -20 or subtract 20 from d1
```

For a 16-bit constant:

```
addi   d0,d1,#0xFFFFEDA6    ;add d1 and -4691 or subtract 4691 from d1
```

If the operation is to subtract a 16-bit constant bigger than 0x7FFF, two instructions are required:

```
mov.u   d0,#0x96a0
```

```
sub     d2,d1,d0
```

Example:

```
ld.w d1,1x5
```

d1	EB5F 7F11
----	-----------

After Instruction

```
add d2,d1,#0xFFFFFEC
```

d2	EB5F 7EFD
----	-----------

```
addi d0,d1,#0xFFFFEDA6
```

d0	EB5F 6CB7
----	------------------

```
mov.u d3,#0x96a0
```

d3	0000 96A0
----	-----------

```
sub d2,d1,d3
```

d2	EB5E E871
----	-----------

2.2.3 16-bit Subtraction

Subtracting two 16-bit values uses the 32-bit subtraction.

Example:

```
ld.h d1,sX1
```

```
ld.h d2,sX2
```

```
sub d0,d1,d2
```

To use the saturated subtraction, the two values must be left aligned:

```
ld.q d1,sX1
```

```
ld.q d2,sX2
```

```
subs d4,d1,d2
```

Alternatively, use the SAT instruction:

```
ld.h d1,sX1
```

```
ld.h d2,sX2
```

```
sub d0,d1,d2
```

```
sat.h d0,d0
```

Example:

After Instruction

```
ld.h d1,sX3
```

d1	FFFF F001
----	-----------

```
ld.h d2,sX4
```

d2	0000 7A10
----	-----------

```
sub d0,d1,d2
```

d0	FFFF 75F1
----	-----------

```
subs d3,d1,d2
```

d3	FFFF 75F1
----	-----------

ld.q d1, sX3

ld.q d2, sX4

subs d4, d1, d2

sub d0, d1, d2

d1	F001 0000
d2	7A10 0000
d4	8000 0000
d0	75F1 0000

2.2.4 16-bit Subtraction using Packed Arithmetic (SUB.H, SUBS.H, SUBS.HU)

The packed instructions SUB.H, SUBS.H, SUBS.HU perform two 16-bit subtractions in one cycle.

Normal subtraction:

sub.h d0, d1, d2

With saturation:

subs.h d0, d1, d2

subs.hu d0, d1, d2

OVF: The result of the subtraction cannot be represented in a 16-bit register.

2.2.5 8-bit Subtraction Using Packed Arithmetic (SUB.B)

The packed instruction SUB.B performs four 8-bit subtractions in one cycle:

sub.b d0, d1, d2

OVF: The result of the subtraction cannot be represented in an 8-bit register.

2.2.6 64-bit Subtraction (SUBC, SUBX)

64-bit subtraction uses specific instructions: SUBX (subtract extended) and SUBC (subtract with carry). Borrow is reported in the carry flag **psw.c** so that the next 32-bit subtraction can take account of it.

Borrow is used as: $\text{data} + \text{psw.c} - 1$

Example:

		After Instruction		
ld.d	e0, eX4	e0	0034 0002 7540 0000	
ld.d	e2, eX6	e2	0000 0001 F24D 8A10	
subx	d4, d0, d2	d4	82F2 75F0	psw.c = 0
subc	d5, d1, d3	d5	82F2 75F0	psw.c = 1

The result is in the registers d4 and d5.

OVF: The result of the subtraction cannot be represented in a 32-bit register.

2.2.7 64-bit Subtraction with Constant

64-bit subtraction with a constant is a combination of multi precision subtraction and 32-bit addition, with a signed or unsigned constant. If one of the lower 16-bit halves of the 64-bit constant is bigger than 0x7FFF, replace the ADDI instruction with the MOV.U instruction.

Example:

The 64-bit constant is in registers d0 and d1 (0x11111000 98765432). The 64-bit value is in registers d4 and d5 (0x7BFF7F10 EFFF5AE0). The result is in registers d6 and d7.

After Instruction

```
mov    d9, #0
ld.d   e4, eX1
addi   d0, d9, #0x5432
addih  d0, d0, #0x9876
subx   d6, d4, d0
addi   d1, d9, #0x1000
addih  d1, d1, #0x9999
subc   d7, d5, d1
```

d9	0000 0000
e4	7BFF 7F10 EFFF 5AE0
d0	0000 5432
d0	9876 5432
d6	5789 06AE
d1	0000 1000
d1	9999 1000
d7	E266 6F10

psw.c = 1

2.2.8 Reverse Subtraction (RSUB)

RSUB is used for subtracting a register from an 8-bit constant. There are three different instructions:

- RSUB (normal)
- RSUBS (saturated signed)
- RSUBS.U (saturated unsigned)

Example:

```
ld.w    d1, 1x6
```

d1	7BFF 7F1F
----	-----------

After Instruction

```
rsub    d5, d1, #0x54
```

d5	8400 8135
----	-----------

```
rsubs   d6, d1, #0x54
```

d6	8400 8135
----	-----------

```
rsubs.u d7, d1, #0x54
```

d7	0000 0000
----	-----------

OVF: The result of the subtraction cannot be represented in a 32-bit register.

16-bit opcode syntax:

```
rsub dest/src
```

;subtract the register dest/src from zero and put the result in the same register.

;Tasking's syntax is: rsub dest/src, #0 for this case

2.3 Negate

Negating a value is synthesized with the RSUB instruction, *using zero as a constant*. Negate is equivalent to two's complement. To use 1's complement, use synthesized NOT (See [Logical Instructions](#)).

Example:

```
ld.w    d1, 1x7
```

d1	547F 3E49
----	-----------

After Instruction

```
rsub    d0, d1, #0
```

d0	AB80 C1B7
----	-----------

```
add     d2, d1, d0
```

d2	0000 0000
----	-----------

2.4 Move

This section covers:

- **Moves between Data Registers (MOV)**
- **16-bit Constant into a Data Register (MOV.U, MOV, MOVH)**
- **32-bit Constant into a Data Register**

2.4.1 Moves between Data Registers (MOV)

The MOV instruction moves a register into another register. From the hardware viewpoint a move instruction is identical to an arithmetic operation, hence its inclusion here. Since the move instructions are considered as arithmetic instructions, they can be *dual issued with load and store instructions*.

```
mov    d0,d1
```

16-bit opcode syntax:

```
mov    dest,src
```

2.4.2 16-bit Constant into a Data Register (MOV.U, MOV, MOVH)

The MOV.U instruction moves a 16-bit *unsigned* constant into a register.

```
mov.u   d0,#0x4567
```

The MOV instruction could also be used:

```
mov     d0,#0x1234
```

The difference between the two instructions is that MOV is sign-extended whereas MOV.U is zero-extended. The difference between the MOV and the MOVH instruction:

```
movh    d0,#0x4567
```

MOVH puts the 16-bit constant in the upper half.

Note: Some assembler will not accept a 16-bit negative signed constant. This is due to the assembler always dealing with 32-bit operand and always zero-extending 16-bit value to 32-bit. In the example which follows, it will interpret 0x0000ABCD which cannot be represented in a 16-bit signed format.

```
mov     d0,#0xABCD           ;error message
```

The solution is to manually sign extend to 32-bit.

```
mov     d0,#0xFFFFABCD      ;okay
```

Example:

```
ld.w d1,1x6
```

d1	7BFF 7F1F
----	-----------

After Instruction

```
mov d0,d1
```

d0	7BFF 7F1F
----	-----------

```
mov d2,#0xFFFF8567
```

d2	FFFF 8567
----	-----------

```
mov.u d3,#0x8567
```

d3	0000 8567
----	-----------

```
movh d4,#0x8567
```

d4	8567 0000
----	-----------

Note: The move instructions are considered arithmetic instructions and can be dual issued with load and store instructions.

16-bit opcode syntax:

```
mov dest/src,k4
```

```
mov d15,k8
```

2.4.3 32-bit Constant into a Data Register

The best way to move a 32-bit constant into a data register, is to move the unsigned lower 16-bit constant in the lower part and then to add the upper 16-bit constant to the upper halfword. There is **no restriction** on the values of the constant.

Example:

After Instruction

```
mov.u d0,#0x5678
```

d0	0000 5678
----	-----------

```
addih d0,d0,#0x1234
```

d0	1234 5678
----	-----------

2.5 Applications

Examples of an arithmetic series of applications which use MOV and ADD instructions are provided in this section, as well as a multi-byte operation example which uses packed instructions such as ADD.B.

2.5.1 Arithmetic Series

$\Sigma i (= 1+2+3+...+n)$

This example illustrates the use of simple instructions. There is an accumulation and an increment which takes place inside the loop body.

Example:

```

mov      d4,#1           ; first number in the series
mov      d5,#3           ; last number in the series
sub      d5,d4           ; loop = last - first
mov.a    a0,d5           ; initialization of the loop counter
mov      d2,#0           ; sum = 0
loopg:   add      d2,d4   ; sum += number
         add      d4,#1   ; number++
         loop     a0,loopg ; loop back to loopg label

```

Evolution of D2 and D4 in the loop:

After Instruction				
mov	d4,#1	d4	0000 0001	init.
mov	d2,#0	d2	0000 0000	
add	d2,d4	d2	0000 0001	1st iteration
add	d4,#1	d4	0000 0002	
add	d2,d4	d2	0000 0003	2nd iteration
add	d4,#1	d4	0000 0003	
add	d2,d4	d2	0000 0006	3rd iteration
add	d4,#1	d4	0000 0004	
				etc.

2.5.2 Multi Byte Operations

A, B, C, D are four 8-bit values packed in the same 32-bit register. To compute these two equations in the minimum of cycles:

$$F = A - B + C - D$$

$$S = A + B + C + D$$

The best way is to use packed instructions.

First arrange the lower byte of 4 registers to contain the values.

$$D0 = D \ C \ B \ A$$

$$D1 = A \ D \ C \ B$$

$$D2 = B \ A \ D \ C$$

$$D3 = C \ B \ A \ D$$

Use the DEXTR instruction to do the rotations (See [Bit Field Operations](#)).

Compute F as (A+C) - (B+D)

Compute S as (A+C) + (B+D)

Example:

After Instruction

ld.w	d0, 1x9	d0	1122 3344	
dextr	d1, d0, d0, #24	d1	4411 2233	
dextr	d2, d0, d0, #16	d2	3344 1122	
dextr	d3, d0, d0, #8	d3	2233 4411	
add.b	d4, d0, d2	d4	4466 4466	
add.b	d5, d1, d3	d5	6644 6644	
sub.b	d6, d4, d5	d6	DE22 DE22	; F = (A+C) - (B+D)
add.b	d7, d0, d1	d7	5533 5577	
add.b	d8, d2, d3	d8	5577 5533	
add.b	d9, d7, d8	d9	AAAA AAAA	; S = (A+C) + (B+D)

3 Further Arithmetic

This chapter describes arithmetic instructions that are:

- Either common but not universal (absolute value, multiplication, division)
- Not very common (minimum, maximum)
- Application specific (absolute difference)

Application specific arithmetic instructions are used in Digital Signal Processing. They are included here and not in the specific DSP chapter because they are ALU instructions. The specific DSP chapter deals mainly with MAC instructions.

Chapter contents:

- **Absolute Value**
- **Absolute Difference**
- **Minimum, Maximum**
- **Saturate**
- **Multiplication**
- **Division**
- **Applications**

3.1 Absolute Value

There are 3 data types:

- word
- packed halfword
- packed bytes

Example:

d1	DEAD BEEF
----	-----------

After Instruction

abs d4 , d1

d4	2152 4111
----	-----------

abs.h d5 , d1

d5	2153 4111
----	-----------

abs.b d6 , d1

d6	2253 4211
----	-----------

3.1.1 32-bit (ABS, ABSS)

if (d0 >= 0) d2 = d0 ; else d2 = -d0;

Translates to:

```
abs    d2, d0
```

Saturated case is:

```
abss   d2, d0
```

The following example illustrates the difference with the normal case.

Example:

d0	8000 0002
d1	8000 0000

After Instruction

```
abs    d2, d0
```

```
abss   d3, d0
```

```
abs    d2, d1
```

```
abss   d3, d1
```

d2	7FFF FFFE
d3	7FFF FFFE
d2	8000 0000
d3	7FFF FFFF

OVF: Will be set if 0x8000 0000.

3.1.2 Packed 16-bit (ABS.H, ABSS.H)

Two absolute 16-bit values can be computed in one cycle:

if (d0U >= 0) d2U = d0U ; else d2U = -d0U;

if (d0L >= 0) d2L = d0L ; else d2L = -d0L;

where: U is bits [31:16] & L is bits [15:0]

abs.h d1, d0

The saturated case is:

abss.h d1, d0

Example:

d0	9000 4000
d1	8000 8000
d2	8000 4000
d3	9000 A000

After Instruction

abs.h d4, d0

abss.h d5, d0

abs.h d6, d1

abss.h d7, d1

abs.h d8, d2

abss.h d9, d2

abs.h d10, d3

abss.h d11, d3

d4	7000 4000
d5	7000 4000
d6	8000 8000
d7	7FFF 7FFF
d8	8000 4000
d9	7FFF 4000
d10	7000 6000
d11	7000 6000

OVF: At least one of the two source data is 0x8000.

3.1.3 Packed 8-bit (ABS.B)

Four absolute 8-bit values can be computed in one cycle:

abs.b d1, d0

There is no saturated case for bytes. **OVF:** At least one of the four source data is 0x80.

3.2 Absolute Difference

There are 3 Data types:

- word
- packed halfword
- packed byte

3.2.1 32-bit Value (ABSDIF, ABSDIFS)

if $(d0 > d1)$ $d2 = d0 - d1$; else $d2 = d1 - d0$;

`absdif d2, d0, d1`

The saturated case is:

`absdifs d2, d0, d1`

ABSDIFS behaves the same as ABSDIF, except that it will saturate to 0x7FFFFFFF in case of overflow.

OVF: The result of the subtraction cannot be represented in a 32-bit register OR the result is equal to 0x80000000. In other words, if the equation is rewritten as **$d2 = \text{abs}(d0 - d1)$** , there are two sources of overflow:

- overflow 1 can be generated by the subtraction
- overflow 2 can be generated by the absolute value

Constant: An 8-bit constant can replace the second source register ($d1$ in the examples).

Example:

d0	93FE 85FF
d1	7BFF 7F1F

After Instruction

`absdif d2, d0, d1`

d2	E800 F920
d3	7FFF FFFF

`absdifs d3, d0, d1`

3.2.2 Packed 16-bit (ABSDIF.H, ABSDIFS.H)

Two 16-bit absolute differences can be performed in one cycle.

if ($d0U > d1U$) $d2U = d0U - d1U$; else $d2U = d1U - d0U$;

if ($d0L > d1L$) $d2L = d0L - d1L$; else $d2L = d1L - d0L$;

where:

- U is bits [31:16]
- L is bits [15:0]

The syntax is the same as for a word data type except for the addition of .H (Halfword) suffix:

```
absdif.h    d2,d0,d1
```

For the saturated case:

```
absdifs.h    d2,d0,d1
```

ABSDIFS saturates to 0x7FFF in case of overflow.

OVF: The result of the subtraction cannot be represented in a 16-bit register OR the result is equal to 0x8000. This applies to either the upper or lower packed 16-bit operations.

3.2.3 Packed 8-bit (ABSDIF.B)

Four 8-bit absolute differences can be performed in one cycle:

```
absdif.b    d2,d0,d1
```

There is no saturated case.

OVF: The result of the subtraction cannot be represented in an 8-bit register OR the result is equal to 0x80. This applies to any of the four 8-bit absolute differences.

Example:

d0	93 FE 85 FF
d1	7B FF 7F 1F

After Instruction

absdif.h	d4,d0,d1	d4	E8 01 F9 20
absdifs.h	d5,d0,d1	d5	7F FF 7F FF
absdif.b	d2,d0,d1	d2	E8 01 FA 20

3.3 Minimum, Maximum

Minimum and maximum instruction can be performed on 32-bit, 16-bit, or 8-bit signed or unsigned values.

3.3.1 32-bit Signed/Unsigned (MIN, MIN.U, MAX, MAX.U)

There is a choice of four different instructions: MIN, MAX, MIN.U and MAX.U. The need for signed/unsigned differentiation is due to the different ranges, as shown:

0x000000000x7fffffff ..0x80000000.....0xFFFFFFFF
 unsigned **MIN**===== **MAX**
 signed ===== **MAX...MIN**=====

Example:

d0	FFFF 5AE0
d1	00F4 7F54

After Instruction

min d2, d0, d1

min.u d3, d0, d1

max d4, d0, d1

max.u d5, d0, d1

d2	FFFF 5AE0
d3	00F4 7F54
d4	00F4 7F54
d5	FFFF 5AE0

Constant: An 8-bit constant can replace the second source register (d1 in the examples).

3.3.2 Packed 16-bit Signed/Unsigned (MIN.H, MAX.H, MIN.HU, MAX.HU)

There is a choice of four instructions: MIN.H, MAX.H, MIN.HU and MAX.HU.

Example:

d0	FFFF 5AE0
d1	00F4 7F54

After Instruction

min.h	d2, d0, d1	d2	FFFF 5AE0
min.hu	d3, d0, d1	d3	00F4 5AE0
max.h	d4, d0, d1	d4	00F4 7F54
max.hu	d5, d0, d1	d5	FFFF 7F54

3.3.3 Packed 8-bit Signed/Unsigned (MIN.B, MAX.B, MIN.BU, MAX.BU)

There is a choice of four instructions: MIN.B, MAX.B, MIN.BU, and MAX.BU.

Example:

d0	EF FF 5A E0
d1	00 F4 7F 54

After Instruction

min.b	d2, d0, d1	d2	EF F4 5A E0
min.bu	d3, d0, d1	d3	00 F4 5A 54
max.b	d4, d0, d1	d4	00 FF 7F 54
max.bu	d5, d0, d1	d5	EF FF 7F E0

3.4 Saturate

3.4.1 32-bit Saturation

Saturating a 32-bit operand can be produced by using the suffix 's' on most arithmetic and shift operations.

Example: adds, madds, msubs, abss, shas, etc.

3.4.2 16-bit and 8-bit Saturation (SAT.H, SAT.HU, SAT.B, SAT.BU)

The SAT instruction *does not look at two separated halfwords* (or four separated bytes), but checks if a register value is outside the 16-bit range (8-bit respectively).

SAT.H signed: if (d0 > 0x00007FFF) d1 = 0x00007FFF
 if (d0 < 0xFFFF8000) d1 = 0xFFFF8000

SAT.HU unsigned: if (d0 > 0x0000FFFF) d2 = 0x0000FFFF

SAT.B: signed: if (d0 < 0xFFFF80) d3 = 0xFFFF80
 if (d0 > 0x0000007F) d3 = 0x0000007F

SAT.BU unsigned: if (d0 > 0x000000FF) d4 = 0x000000FF

Examples with positive value:

d0	00F4 3645
----	-----------

After Instruction

sat.h d1, d0

d1	0000 7FFF
----	-----------

sat.hu d2, d0

d2	0000 FFFF
----	-----------

sat.b d3, d0

d3	0000 007F
----	-----------

sat.bu d4, d0

d4	0000 00FF
----	-----------

Examples with negative value:

d0	FFFF 5AE0
----	-----------

After Instruction

sat.h d1, d0

d1	FFFF 8000
----	-----------

sat.b d3, d0

d3	FFFF FF80
----	-----------

16-bit opcode syntax:

```
sat.h      dest/src
sat.hu     dest/src
sat.b      dest/src
sat.bu     dest/src
```

3.5 Multiplication

In this section the 32x32 multiplication instructions are explained. There is a large choice of integer multiplication. They differ by the sign of the operands (signed / unsigned), the size of the result (32-bit, 64-bit) and the overflow behaviour (saturated or not). The multiply-add operation is also introduced.

3.5.1 32 * 32-bit Multiplication (MUL, MULS, MULS.U)

When multiplying a 32-bit value by another 32-bit value in the 'C' language, the upper 32-bit result is discarded and the lower 32-bit result is retained. Consequently there is no need to differentiate between signed and unsigned, and one instruction (MUL) covers both cases.

```
long y, a, b;
y = a * b;
```

is translated:

```
mul    d2, d1, d0
```

Example:

d0	00006000
d1	00004000

After Instruction

```
mul    d2, d1, d0
```

d2	18000000
----	----------

As the numbers get larger, the result will overflow.

Example:

d0	60000000
d1	40000000

After Instruction

```
mul    d2, d1, d0
```

d2	00000000
----	----------

If the user wishes to use saturation, the instructions MULS and MULS.U are available. MULS is for signed numbers and saturates at 0x80000000 for negative values and 0x7FFFFFFF for positive values. MULS.U is for unsigned numbers and saturates at 0xFFFFFFFF.

Example:

d0	FFFF 5AE0
d1	7BFF 7F1F

After Instruction

`mul d2,d1,d0`

`muls d3,d1,d0`

`muls.u d4,d1,d0`

d2	E321 2120
d3	8000 0000
d4	FFFF FFFF

OVF: The result of the multiplication cannot be represented in a 32-bit register.

Constant: An 8-bit constant can only replace the second source register (d0 in the examples) for the MUL instruction.

16-bit opcode syntax:

`mul dest/src1,src2`

3.5.2 32 * 32-bit Multiplication with a 64-bit Result (MUL, MUL.U)

For complete accuracy, TriCore offers 64-bit results. There are two instructions: MUL and MUL.U. Unlike a 32-bit result, a multiplication with a 64-bit result needs to differentiate between signed and unsigned.

In a signed multiplication the partial products of the multiplier are sign-extended, whereas in an unsigned multiplication the partial products are zero-extended.

There is no need for saturated instruction since this operation can never overflow.

The assembler uses the same mnemonic as for a 32-bit result and distinguishes by the register size.

```
mul    d0,d2,d3    ; result on 32-bit
mul    e0,d2,d3    ; signed result on 64-bit
mul.u  e0,d2,d3    ; unsigned result on 64-bit
```

Example:

d2	FFFF 5AE0
d3	7BFF 7F1F

After Instruction

mul e0,d2,d3	e0	F83F B812	E321 2120
mul.u e0,d2,d3	e0	743F 3731	E321 2120

Constant: An 8-bit constant can replace the second source register (d3 in the examples).

3.5.3 16 * 16-bit Multiplication

Multiplying two 16-bit numbers is executed by the MUL instruction:

```
ld.h    d0,sX1
ld.h    d1,sX2
mul     d2,d1,d0
```

A 32-bit result is produced and overflow cannot occur.

3.5.4 8 * 8-bit Multiplication

8*8-bit multiplication is executed as for the 16*16-bit multiplication, except that the values are loaded as bytes.

Example:

```
ld.b    d0, bX1
ld.b    d1, bX2
mul     d2, d1, d0
```

A 16-bit result is produced and overflow cannot occur.

3.5.5 Multiplication by a Power of 2

Multiplying a value by a constant, which is a power of 2, can be achieved in two ways. The MUL instruction with constant can be used:

```
mul     d0, d1, #2    ; d1 is multiplied by 2.
```

As the MUL constant range is limited to 8-bit, *the shift instructions* are useful because they can multiply up to 2^{31} instead of only 255.

```
sh      d0, d1, #12    ; d1 is multiplied by 2 to the power 12.
```

3.5.6 Multiplication/Addition (MADD, MADDS)

TriCore offers a great range of DSP MAC instructions (See chapters 10 and 11). These instructions are introduced in this section as extensions of the MUL instruction. This is the method they most likely to be used in standard code. In addition, they have 2 features not offered by the DSP MAC instructions:

- They work with both signed and unsigned
- They can have an 8-bit constant as one of the multiplier operand

TriCore offers several sub-types which are identical to the MUL instruction: unsigned/signed, normal/saturation and 2 possible accumulator sizes (32 or 64 bits).

In the first example the accumulator source (d1) and destination (d0) are 32-bit.

In the second example the accumulator source (e8) and destination (e0) are 64-bit.

In both examples the multiplier sources (d2, d3) are 32-bit.

Example 1:

```
madd     d0, d1, d2, d3 ; d0 = d1 + (long) (d2*d3)
madds    d0, d1, d2, d3 ; d0 = d1 + (long) (d2*d3)
madds.u  d0, d1, d2, d3 ; d0 = d1 + (long) (d2*d3)
```

Example 2:

```
madd      e0,e8,d2,d3 ; e0 = e8 + (signed_extend_64-bit) (d2*d3)
madds     e0,e8,d2,d3 ; e0 = e8 + (sign_extend_64-bit) (d2*d3)
madds.u   e0,e8,d2,d3 ; e0 = e8 + (zero_extend_64-bit) (d2*d3)
```

OVF: The result of the addition cannot be represented in a 32-bit (or 64-bit) register.

Constant: An 8-bit constant can replace the second source register (d3 in the examples).

3.5.7 Multiplication/Subtraction (MSUB, MSUBS)

For any MADD instruction there is a corresponding MSUB instruction.

In the first example the accumulator source (d1) and destination (d0) are 32-bit.

In the second example the accumulator source (e8), destination (e0) and the product are 64-bit.

In both examples the multiplier sources (d2, d3) are 32-bit.

Example 1:

```
msub      d0,d1,d2,d3 ; d0 = d1 - (long) (d2*d3)
msubs     d0,d1,d2,d3 ; d0 = d1 - (long) (d2*d3)
msubs.u   d0,d1,d2,d3 ; d0 = d1 - (long) (d2*d3)
```

Example 2:

```
msub      e0,e8,d2,d3 ; e0 = e8 - (d2*d3)
msubs     e0,e8,d2,d3 ; e0 = e8 - (d2*d3)
msubs.u   e0,e8,d2,d3 ; e0 = e8 - (d2*d3)
```

OVF: The result of the subtraction cannot be represented in a 32-bit (or 64-bit) register.

Constant: An 8-bit constant can replace the second source register (d3 in the examples).

3.6 Division

The following division cases are presented:

- **32 / 32-bit Division (DVINIT, DVSTEP, DVADJ)**
- **16 / 16-bit Division (DVINIT.H, DVSTEP)**
- **8 / 8-bit Division (DVINIT.B, DVSTEP)**
- **32 / 16-bit Division**
- **Division By a Power of 2**

3.6.1 32 / 32-bit Division (DVINIT, DVSTEP, DVADJ)

TriCore subdivides the division instruction into 3 sub-instructions:

- DVINIT instruction extends the 32-bit dividend into 64 bits and checks if the divisor is different from zero.
- DVSTEP instruction obtains the quotient and the remainder. A DVSTEP performs 8 bits of division at a time. In the current TC1.3 implementation, each DVSTEP takes 4 cycles.
- DVADJ instruction is required to perform a final adjustment of negative values, when the divide operation was signed.

```

dvinit  dest(extended register),dividend,divisor
dvstep  remainder/quotient(extended register),extended dividend,divisor
dvadj   remainder/quotient(extended register),extended dividend,divisor
  
```

The following example is a 32-bit division of 268435456 / 50331648. The divisor corresponds to register d4 and the dividend is register d5. For the 2 results, the quotient is in register d2 = 5 and the remainder is in the register d3 = 16777216.

Example:

		After Instruction	
ld.d	e4, eX8	e4	10000000 03000000
dvinit	e0, d5, d4	e0	0000 0000 1000 0000
dvstep	e2, e0, d4	e2	0000 0010 0000 0000
dvstep	e0, e2, d4	e0	0000 1000 0000 0000
dvstep	e2, e0, d4	e2	0010 0000 0000 0000
dvstep	e0, e2, d4	e0	0100 0000 0000 0005
dvadj	e2, e0, d4	e2	0100 0000 0000 0005

OVF: The divisor is equal to zero.

3.6.2 16 / 16-bit Division (DVINIT.H, DVSTEP)

In 16-bit division, only two divide_steps are required.

Example:

		After Instruction	
ld.h	d4, sX8	d4	0000 0300
ld.h	d5, sX5	d5	0000 1000
dvinit.h	e0, d5, d4	e0	0000 0000 1000 0000
dvstep	e2, e0, d4	e2	0000 0010 0000 0000
dvstep	e0, e2, d4	e0	0000 0100 0000 0005
dvadj	e2, e0, d4	e2	0000 0100 0000 0005

OVF: The divisor is equal to zero.

3.6.3 8 / 8-bit Division (DVINIT.B, DVSTEP)

With an 8-bit division, only one divide_step is required.

Example:

		After Instruction	
ld.b	d4, bX8	d4	0000 0003
ld.b	d5, bX5	d5	0000 0010
dvinit.b	e0, d5, d4	e0	0000 0000 1000 0000
dvstep	e2, e0, d4	e2	0000 0001 0000 0005
dvadj	e0, e2, d4	e0	0000 0001 0000 0005

OVF: The divisor is equal to zero.

3.6.4 32 / 16-bit Division

Generating a 32-bit dividend divided by a 16-bit divisor is executed in a similar way to a 32 / 32-bit division. The difference is in the load of the divisor, because it must be a 16-bit value. The divisor is register d4, the dividend in d5, the quotient is still in d2 and the remainder in d3.

Example:

		After Instruction	
ld.h	d4, sX8	d4	0000 0007
ld.w	d5, lX4	d5	0010 054F
dvinit	e0, d5, d4	e0	0000 0000 0010 054F
dvstep	e2, e0, d4	e2	0000 0000 1005 4F00
dvstep	e0, e2, d4	e0	0000 0002 054F 0002
dvstep	e2, e0, d4	e2	0000 0006 4F00 0249
dvstep	e0, e2, d4	e0	0000 0005 0002 49E6
dvadj	e2, e0, d4	e2	0000 0005 0002 49E6

3.6.5 Division By a Power of 2

Dividing a value by a power of 2 is faster with a shift instruction than with a divide instruction. Refer to the chapter on [Bit Field Operations](#) for a complete description of shift.

```
sh    d0, d1, #-2    ;d1 is divided by 4.
sh    d0, d2, #-22   ;d2 is divided by 2 to the power of 22.
```

Note: Using the bitwise AND instruction to find the remainder from a divide by a power of 2, is faster than using a DIVIDE instruction. For example:

```
AND   d0, d1, #(N-1) ;d0 = remainder of d1/N
```

3.7 Applications

The following applications demonstrate the use of the instructions previously described:

- **Find Maximum of Value in an Unsigned Array**
- **Template Matching**
- **40 * 40-bit Multiplication**
- **Leapyear**

3.7.1 Find Maximum of Value in an Unsigned Array

Example:

```

mov          d1,#0                ; initialize the data reference
lea          a8,Vvalues           ; initialize the pointer
lea          a6,(Nval-1)          ; number of values-1
ld.w         d2,[a8+]4            ; load first value
iloop:       max.u                d1,d1,d2    ; take the max of two values
             ld.w                 d2,[a8+]4    ; load next value
loop         a6,iloop

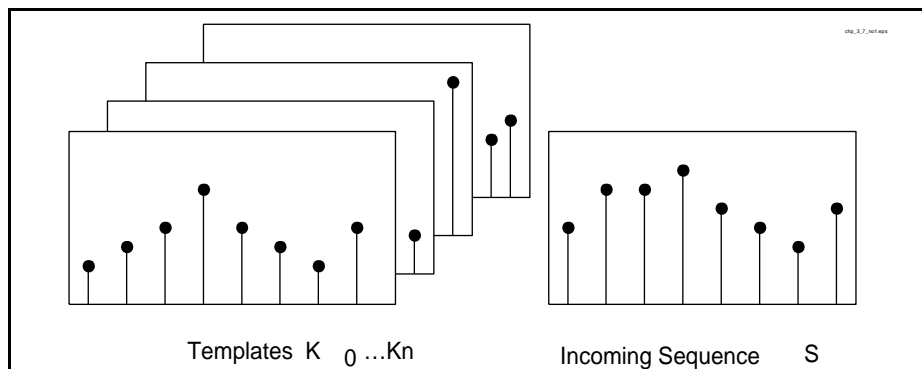
```

To get the minimum value, change the reference value (d1) and replace the MAX instruction with MIN.

3.7.2 Template Matching

The ABSDIF instruction is used to determine the absolute difference between a fixed sequence of values (a template) and an incoming sequence of values. This is useful for template matching.

In template matching, there is one incoming sequence and N templates. The goal is to keep the best match between incoming sequence and the template.



In this instance we need to know if signal S looks like template K_n . Comparing each sample S_t with the correspondent sample K_t will finally give $\sum |S_t - K_t|$.

Example:

This example compares a signal and one template of 16 samples. It begins with the initialization of the sum and the initialization of the loop. Then in a loop, addition of the absolute difference is calculated ($\sum |S_t - K_t|$). Note that only the internal loop is shown.

a2	D000 00B0
a3	D000 00F0

After Instruction

mov.u	d0, #0	d0	0000 0000
-------	--------	----	-----------

lea a4, 0x0f

ld.w	d1, [a2+] 4	d1	FFFF 5AE0	a2	D000 00B4
------	-------------	----	-----------	----	-----------

ld.w	d2, [a3+] 4	d2	0004 F000	a3	D000 00F4
------	-------------	----	-----------	----	-----------

temploop:		d3	1005 9520		
absdif	d3, d1, d2				

ld.w	d1, [a2+] 4	d1	7BFF 7F1F	a2	D000 00B8
------	-------------	----	-----------	----	-----------

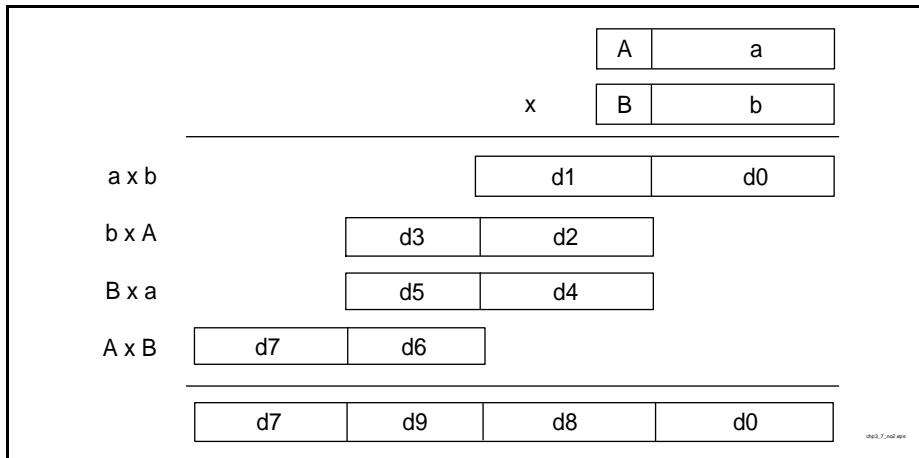
add	d0, d3, d0	d0	1005 9520		
-----	------------	----	-----------	--	--

ld.w	d2, [a3+] 4	d2	FF66 F000	a3	D000 00F8
------	-------------	----	-----------	----	-----------

loop a4, temploop

3.7.3 40 * 40-bit Multiplication

A 40*40 multiplication consists of four multiplications: one 32*32 bits (a x b), two 32*8 bits (b x A and a x B) and one 8*8 bits (A x B).



Example:

After Instruction				
ld.d	e8,eX0	e8	0000 0010 1122 3344	;load A and a
mov.u	d15,#0	d15	0000 0000	
ld.d	e10,eX5	e10	0000 0004 7988 7766	;load B and b
mul	e0,d9,d11	e0	0000 0000 0000 0040	;multiplication axb
mul	e2,d11,d8	e2	0000 0000 4488 CD10	;multiplication bxA
mul	e4,d10,d9	e4	0000 0007 9887 7660	;multiplication Bxa
mul	e6,d8,d10	e6	0822 4C64 1995 0918	;multiplication AxB
addx	d12,d1,d2	d12	4488 CD10	
addc	d8,d12,d4	d8	DD10 4370	
addc	d13,d3,d5	d13	0000 0007	
addc	d9,d13,d6	d9	1995 091F	
addc	d7,d15,d7	d7	0822 4C64	

3.7.4 Leapyear

This routine uses the division to generate the modulo.

Parameters

entry	d4 = year
exit	d2 = yes / no

Pseudo code

```
if      ((year % 400) == 0)  leapyear = YES;
else    if ((year % 100) == 0) leapyear = NO;
else    if ((year % 4)  == 0)  leapyear = YES;
```

Assembly code

```
.define  no      "d6"
.define  yes      "d7"
.define  year     "d4"
.define  mod      "d5"
.define  leapyear "d2"

modulo .MACRO y,x
    dvinit    e0,y,d8          ; (4)
    dvstep    e0,e0,x          ; (8)
    dvstep    e0,e0,x          ; (12)
    dvstep    e0,e0,x          ; (16)
    dvstep    e0,e0,x          ; (20)
                                ; no DVADJ because
                                ; unsigned. See Examples file.
    .ENDM

mov     no,#0                  ; (1)      ; initialization of no=0
lea     a2,0x6e                ; ||      ; initialization of counter
mov     yes,#1                 ; (2)      ; initialization of yes=1
ld.h    year,X3                ; ||      ; year
movh.a  a3,#0xb000             ; (3)      ; initialization of memory
lea     a3,[a3]0x0080          ; ||      ;

lloop:  mov.u    mod,#0x0004     ; (1)      ;
        modulo   year,mod        ; (21)     ; mod 4
        sel      leapyear,d1,no,yes ; (22)     ; if (year mod 4 == 0)
                                ; leapyear=yes, else leapyear=no

        mov.u    mod,#0x0064     ; (23)     ;
        modulo   year,mod        ; (43)     ; mod 100
        sel      leapyear,d1,leapyear,no ; (44)     ;
                                ; if (year mod 100 == 0)
                                ; leapyear=no, else leapyear=leapyear
```

```

mov.u    mod,#0x0190      ; (45)      ;
modulo   year,mod         ; (65)      ; mod 400
sel      leapyear,d1,leapyear,yes ; (66)
                                     ; if(year mod 400 == 0)
                                     ; leapyear=yes, else leapyear=leapyear
add      year,year,#1     ; (67)
                                     ; incrementation of the year
st.b     [a3+]1,leapyear  ; ||       ; store the leapyear
loop     a2,lloop

```

4 Conditional Instructions

This chapter deals with a class of computer instructions called predicate or conditional instructions. A predicate is characterized by the execution of the instruction depending on a condition.

Example: **if (a == 0) b = c; else b = d;**

In TriCore assembly language, this is written as: **sel b, a, c, d**

Chapter contents:

- **'Hidden' Conditional Instructions**
- **Select (SEL, SELN)**
- **Conditional Move (CMOV, CMOVN)**
- **Conditional Add and Sub (CADD, CADDN, CSUB, CSUBN)**
- **Application**

4.1 'Hidden' Conditional Instructions

While not being a predicated machine, TriCore does offer a reasonable number of conditional instructions such as MIN, MAX, SAT, ABS and ABSDIF:

```
min      d0,d1,d2    === if (d1<d2) d0=d1          ; else d0=d2;
max      d0,d1,d2    === if (d1>d2) d0=d1          ; else d0=d2;
sat.hu   d0,d1       === if (d1 > 65536) d0= 65536 ; else d0=d1;
sat.h    d0,d1       === if (d1 <-32768) d0=-32768 ;
                                     else if (d1 > 32767) d0= 32767 ; else d0=d1;
```

These instructions save many cycles as they replace 2 or more instructions by just 1 instruction. However the whole point of conditional instruction is *to avoid branches*. Branches can cause pipeline stalls if they are mispredicted. This not only affects performance, but is also makes the code less deterministic.

In theory, any existing instructions can be made conditional. This is the case with the so-called "predicated machines". In TriCore the number of 1-cycle conditional instructions is limited to conditional move (SEL, CMOV), addition (CADD) and subtraction (CSUB). It is also possible to combine several instructions and CMOV to synthesize any conditional instructions as a multi-cycle instruction.

4.2 Select (SEL, SELN)

The SEL instruction selects a register depending on the result of a *comparison with zero*.

```
sel  d3, d0, d1, d2
sel  d13, d0, d1, #25
```

The control value in the first line of the example is in d0.

If d0 is equal to zero, d2 is moved in d3, otherwise d2 is moved to d1.

In the second example, if d0 is equal to zero, 25 will be moved in d13, otherwise 25 is moved to d1.

This instruction exists with the negate suffix:

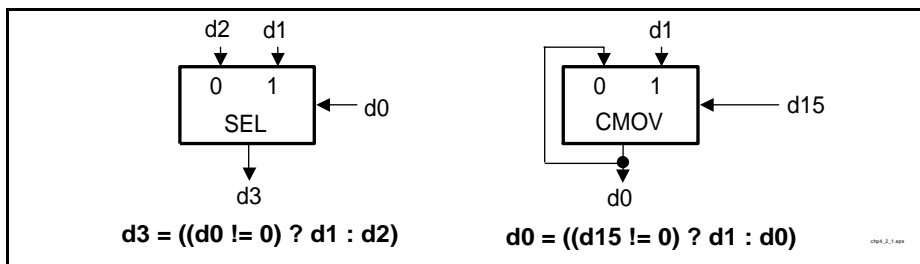
```
seln  d4, d10, d1, d2
```

The control value is in d10. *If d10 is NOT equal to zero*, d2 is moved in d4, otherwise d2 is moved to d1.

Constant: An 8-bit constant can replace the second source register (d2 in the example).

4.3 Conditional Move (CMOV, CMOVN)

As shown in the figure below, CMOV is equivalent to SEL with 1 source tied to the destination.



16-bit opcode syntax:

```
cmov  dest, d15, src
cmov  dest, d15, k4
cmovn dest, d15, src
cmovn dest, d15, k4
```

The advantage of the CMOV instruction is that it is a 16-bit instruction. The drawback is that the control value is always in D15.

```
cmov  d0, d15, d1
```

The instruction also exists with the negate suffix:

```
cmovn d0, d15, d2
```

Example:

d15	0000 0000
d0	1234 5678
d1	7654 3210
d2	7B6F 7F1F

After Instruction

```
sel d3, d0, d1, d2
```

```
cmov d0, d15, d2
```

```
seln d4, d0, d1, d2
```

```
cmovn d5, d15, d2
```

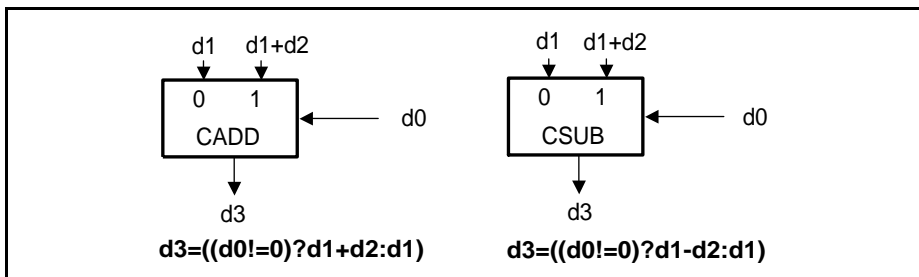
d3	7654 3210
d0	1234 5678
d4	7B6F 7F1F
d5	7B6F 7F1F

4.4 Conditional Add and Sub (CADD, CADDN, CSUB, CSUBN)

```
cadd d3, d0, d1, d2
```

```
csub d4, d0, d1, d2
```

In the first example the select value is in d0. *If d0 is not equal to zero*, d3 will be the result of d1+d2, otherwise only d1 is moved in d3, identically for CSUB (subtraction instead of addition).



These instructions also exist with the negative suffix:

```
caddn d3, d0, d1, d2
```

```
csubn d4, d0, d1, d2
```

Examples:

d0	1234 5678
d1	7654 3210
d2	0000 0003

After Instruction

cadd	d3, d0, d1, d2	d3	7654 3213
csub	d4, d0, d1, d2	d4	7654 320D
caddn	d3, d0, d1, d2	d3	7654 3210
csubn	d4, d0, d1, d2	d4	7654 3210

OVF: The result of the addition/subtraction cannot be represented in a 32-bit register.

Constant: An 8-bit constant can replace the second source register (d2 in the example) for CADD and CADDN only.

Note: CADD and CSUB are destructive operations; the destination register is always written with a new value.

16-bit opcode syntax:

```
cadd    dest/src2, d15, k4
caddn   dest/src2, d15, k4
```

4.5 Application

4.5.1 Multiple If

The following application example is a test of two variables and performing multiple assignments:

Example:

```
if (a > b) x = x + y;   else x = x - y;
if (a != b) x = y;     else x = x + 1;
```

a	d0	0000 000F
b	d1	7540 00FF
x	d2	A000 0490
y	d3	0000 0007

After Instruction

lt	d4, d1, d0	d4	0000 0000
cadd	d2, d4, d2, d3	d2	A000 0490
csubn	d2, d4, d2, d3	d2	A000 0489
ne	d4, d1, d0	d4	0000 0001
caddn	d2, d4, d3, #0x1	d2	0000 0007

5 Logical and Compare

Logic and compare instructions are more universal than arithmetic instructions since they are not linked to a data types. They can be applied to signed integer, unsigned integers, shorts, bit-fields, bits, and so on.

A full range of logical instructions and a series of compare instructions (equal and not equal, greater than and less than, equal any) can work on their own or with shift or logical instruction.

Chapter contents:

- **Logical Instructions**
- **Compare Instructions**
- **Compound Compare Instructions**
- **Application**

5.1 Logical Instructions

Logical instructions include:

- **NOT**
- **AND, OR, XOR**
- **NAND, NOR, ANDN, ORN, XNOR**

5.1.1 NOT

NOT is equivalent to 1's complement.

The NOT instruction is synthesized with the NOR instruction.

Example:

After Instruction		
<code>movh d1, #0x8000</code>	d1	8000 0000
<code>nor d2, d1, #0</code>	d2	7FFF FFFF

More simply, some assemblers accept the alias NOT, which generates a 16-bit opcode.

16-bit opcode syntax:

```
not    src1/dat
```

5.1.2 AND, OR, XOR

The three most common 2-operand logical operations are AND, OR and XOR.

```
and      d2, d0, d1
or       d3, d0, d1
xor      d4, d0, d1
```

Note: C language offers two different types of AND: - && (logic) and &(bit-wise).

The logic AND will give a true or false result (flag) and a bit-wise AND gives a value (for instance 32 bits on a long integer). In a CPU a logical AND is equivalent to the bit wise C & (See the [Boolean Processing](#) chapter for logic C).

Constant: The second source register (d1 in the examples) can be replaced by an 8-bit constant.

16-bit opcode syntax:

```
and      dest/src1, src2
and      d15, k8
or       dest/src1, src2
or       d15, k8
xor      dest/src1, src2
```

5.1.3 NAND, NOR, ANDN, ORN, XNOR

TriCore offers a complete set of 2-operand logical instructions as shown with the truth table below.

The total number of possible combinations of operations of a 2-input logic gate is 16 (2 to the power 2 to the power 2). Among those 16 possibilities, 2 are degenerated 0-operand (0,1), 4 are 1-input gate (a, b, !a, !b) and 3 are the common 2-input cases (and, or, xor). There are therefore 7 remaining operations and TriCore provides 5 of them. The last 2 possible operations ($y = a \text{ AND } !b$), can be implemented by substituting a with b. Similarly for ($y = a \text{ OR } !b$).

Results				function = a op b	TriCore Instruction
a=0 b=0	a=0 b=1	a=1 b=0	a=1 b=1		
0	0	0	0	$y = 0$	turn-off power supply
0	0	0	1	$y = a \text{ and } b$	and
0	0	1	0	$y = a \text{ and } !b$	andn
0	0	1	1	$y = a$	mov
0	1	0	0	$y = !a \text{ and } b$	andn
0	1	0	1	$y = b$	mov
0	1	1	0	$y = a \text{ xor } b$	xor
0	1	1	1	$y = a \text{ or } b$	or
1	0	0	0	$y = a \text{ nor } b$	nor
1	0	0	1	$y = a \text{ xnor } b$	xnor
1	0	1	0	$y = !b$	not
1	0	1	1	$y = a \text{ or } !b$	orn
1	1	0	0	$y = !a$	not
1	1	0	1	$y = !a \text{ or } b$	orn
1	1	1	0	$y = a \text{ nand } b$	nand
1	1	1	1	$y = 1$	create short circuit

Examples:

d0	0101 1010
d1	1001 1001

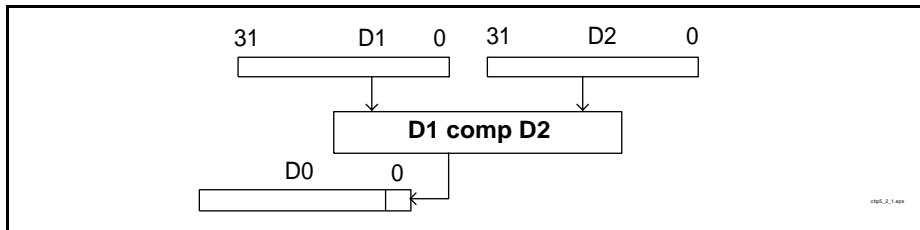
After Instruction

and	d2, d0, d1	d2	0001 1000
or	d3, d0, d1	d3	1101 1011
xor	d4, d0, d1	d4	1100 0011
andn	d5, d0, d1	d5	0100 0010
orn	d6, d0, d1	d6	FFFF FFFE
nand	d7, d0, d1	d7	FFFE EFFF
xnor	d8, d0, d1	d8	EEFF FFEE

Constant: An 8-bit constant can replace the second source register (d1 in the examples).

5.2 Compare Instructions

The general behavior of a compare instruction is shown in the following figure:



5.2.1 Equal and Not Equal (EQ, NE)

The equality comparison:

Example:

After Instruction

eq d2, d0, d1

ne d6, d0, d1

d2	0000 0000
d6	0000 0001

The result is a single flag (bit 0).

Constant: An 8-bit constant can replace the second source register (d1 in the examples).

16-bit opcode syntax:

eq dest, d15, src2

eq dest, d15, k4

5.2.2 Packed Equal (EQ.W, EQ.H, EQ.B)

The difference between a normal equal and packed equal instruction is that, when the comparison is true, EQ.B generates 0xFF for byte, EQ.H generates 0xFFFF for halfword and EQ.W generates 0xFFFFFFFF. It can therefore be used as a mask.

```
eq.w    d3, d1, d2
eq.h    d4, d1, d2
eq.b    d5, d1, d2
```

Packed NE (not equal) instructions do not exist, as they are redundant with equality instructions.

Example:

d1	7BFF 7F1F
d2	7BFF 7F1F

After Instruction

eq	d0, d1, d2	d0	0000 0001
eq.w	d3, d1, d2	d3	FFFF FFFF
ld.w	d7, 1X3	d7	787F 7F4F
eq.h	d4, d1, d7	d4	0000 0000
eq.b	d5, d1, d7	d5	0000 FF00

5.2.3 Equal Any (EQANY.H, EQANY.B)

The EQANY.H and EQANY.B instructions allow a *quick search* for bytes (strings) or half-words (signal processing). If there is at least one equality, the result will be 0x00000001; otherwise it will be 0x00000000.

Example:

d1	7BFF 7F1F
d2	FFFF 5AE0

After Instruction

eqany.h	d3, d1, d2	d3	0000 0000
eqany.b	d4, d1, d2	d4	0000 0001

The result is a single flag (bit 0).

Constant: An 8-bit constant can replace the second source register (d2 in the examples).

5.2.4 Less Than & Greater Than or Equal (LT, LT.U, GE, GE.U)

Comparing two values will lead to different results depending on the values being signed or unsigned. This is why these types of instructions have 2 variants.

Example 1:

d1	D654 3210
d2	4F78 E24F

After Instruction

lt	d3, d1, d2	d3	0000 0001
lt.u	d4, d1, d2	d4	0000 0000

Example 2:

d1	7BFF 7F1F
d2	FFFF 5AE0

After Instruction

ge	d3, d1, d2	d3	0000 0001
ge.u	d4, d1, d2	d4	0000 0000

Constant: An 8-bit constant can replace the second source register (d2 in the examples).

Comparison conditions not explicitly provided in the instruction set can be obtained by either swapping the operands when comparing two registers, or by incrementing the constant by one when comparing a register and a constant.

'Missing' comparison operation	TriCore equivalent comparison operation
LE d0, d1, d2	GE d0, d2, d1
LE d0, d1, k8	LT d0, d1, (k8 + 1)
GT d0, d1, d2	LT d0, d2, d1
GT d0, d1, k8	GE d0, d1, (k8 + 1)

5.2.5 Packed Less Than (LT.B, LT.BU, LT.H, LT.HU, LT.W, LT.WU)

The difference between a normal LT (less than) and a packed LT instruction, is that instead of a flag when the comparison is true, LT.B and LT.BU generate 0xFF for byte, LT.H and LT.HU generate 0xFFFF for halfword and LT.W and LT.WU generate 0xFFFFFFFF for word. It can therefore be used as a *mask*.

Note: The GE.B, GE.BU, GE.H, GE.HU, GE.W, GE.WU instructions do not exist but their results are the opposite of the results of the less than (LT) instruction. If a greater than or equal instruction is needed, use the equivalent less than instruction and interpret the result as follows: The comparison is true when the instruction generates 0x00 (instead of 0xFF).

Example:

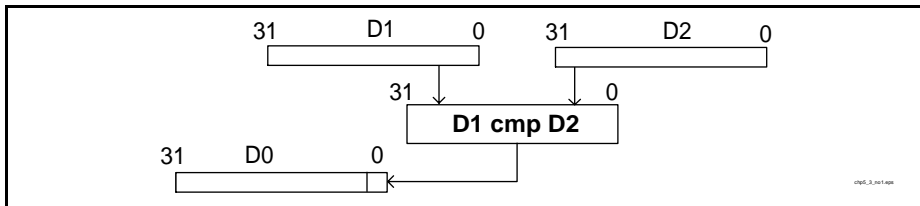
d1	D654 3210
d2	4F78 E24F

After Instruction

lt.b d5, d1, d2	d5	FFFF 00FF
lt.bu d6, d1, d2	d6	00FF FFFF
lt.h d7, d1, d2	d7	FFFF 0000
lt.hu d8, d1, d2	d8	0000 FFFF
lt.w d9, d1, d2	d9	FFFF FFFF
lt.wu d10, d1, d2	d10	0000 0000

5.3 Compound Compare Instructions

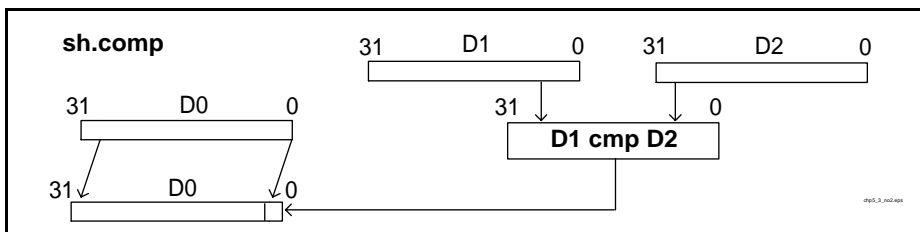
The functionality of the basic compare instructions is shown in the following figure:



It is possible to combine compare instructions with a shift or a logical prefix.

5.3.1 Shift.Compare (SH.EQ, SH.NE, SH.GE, SH.GE.U, SH.LT, SH.LT.U)

The SH prefix is associated with the six comparison instructions: SH.EQ, SH.NE, SH.GE, SH.GE.U, SH.LT, SH.LT.U.



Example:

d0	0000 0010
d1	0001 054F

Before Instruction

After Instruction

sh.eq d2, d0, d1
sh.ge d3, d0, d1
sh.ge.u d4, d0, d1
sh.lt d5, d0, d1
sh.lt.u d6, d0, d1
sh.ne d7, d0, d1

d2	1000 0004	d2	2000 0008
d3	1000 0004	d3	2000 0008
d4	1000 0004	d4	2000 0008
d5	1000 0004	d5	2000 0009
d6	1000 0004	d6	2000 0009
d7	1000 0004	d7	2000 0009

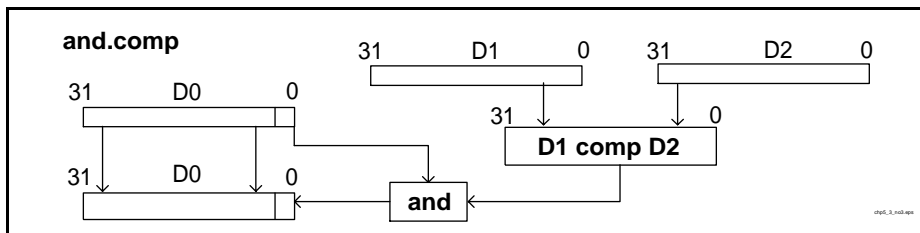
The result is a single flag (bit 0). All other bits are shifted by 1.

Constant: An 8-bit constant can replace the second source register (d1 in examples).

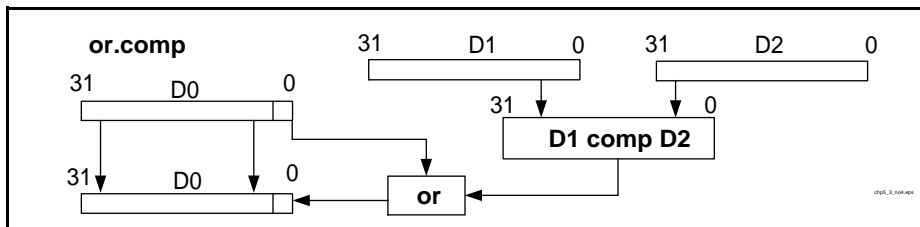
5.3.2 Logical.Compare (AND.comp, OR.comp, XOR.comp)

A logical prefix (AND, OR, XOR) can be associated with any of the six comparison instructions.

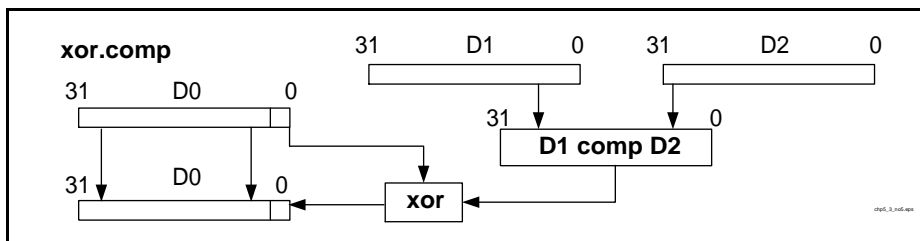
The AND prefix for example, will give six possible instructions: AND.EQ, AND.NE, AND.GE, AND.GE.U, AND.LT, AND.LT.U.



Identically OR



Similarly for XOR



Example:

d0	0000 0010
d1	0001 054F

		Before Instruction		After Instruction	
and.eq	d2, d0, d1	d2	1000 0004	d2	1000 0004
and.ge	d3, d0, d1	d3	1000 0004	d3	1000 0004
and.ge.u	d4, d0, d1	d4	1000 0004	d4	1000 0004
and.lt	d5, d0, d1	d5	1000 0004	d5	1000 0004
and.lt.u	d6, d0, d1	d6	1000 0004	d6	1000 0004
and.ne	d7, d0, d1	d7	1000 0004	d7	1000 0004
or.eq	d2, d0, d1	d2	1000 0004	d2	1000 0004
or.ge	d3, d0, d1	d3	1000 0004	d3	1000 0004
or.ge.u	d4, d0, d1	d4	1000 0004	d4	1000 0004
or.lt	d5, d0, d1	d5	1000 0004	d5	1000 0005
or.lt.u	d6, d0, d1	d6	1000 0004	d6	1000 0005
or.ne	d7, d0, d1	d7	1000 0004	d7	1000 0005
xor.eq	d2, d0, d1	d2	1000 0004	d2	1000 0004
xor.ge	d3, d0, d1	d3	1000 0004	d3	1000 0004
xor.ge.u	d4, d0, d1	d4	1000 0004	d4	1000 0004
xor.lt	d5, d0, d1	d5	1000 0004	d5	1000 0005
xor.lt.u	d6, d0, d1	d6	1000 0004	d6	1000 0005
xor.ne	d7, d0, d1	d7	1000 0004	d7	1000 0005

- Two registers are compared but the logical instruction is applied on bits.
- The result is a single flag (bit 0). All other bits are left unchanged.

Constant: An 8-bit constant can replace the second source register (d1 in the examples).

5.4 Application

This is an example of an “if...else” application which replaces branches by the use of SEL (see [Select \(SEL, SELN\)](#)) and equality.

5.4.1 If... Else

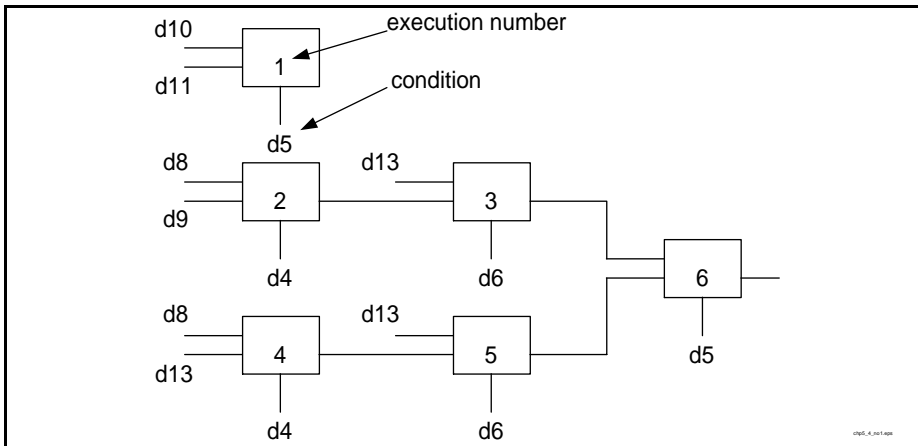
This algorithm comprises of a series of tests before obtaining the final result. It mainly enforces the EQ and SEL instructions. The C code would be:

```
if (cmd.15 == 1) gear = neg(gear);
else gear = gear;
    if (cmd.15 == 1)
        if (TRACEn.0 == 1) TRACEn = TRACEn;
        else if (amb.15 == 1) TRACEn = TRACEn-1;
        else TRACEn = TRACEn+1;
    else if (TRACEn.0 == 1) TRACEn = TRACEn;
        else if (amb.15 == 1) TRACEn = TRACEn-1;
        else TRACEn = TRACEn;
```

It can be re-arranged as:

```
d10 = gear
d11 = neg(gear)
d4 = amd.15
d5 = cmd.15
d6 = TRACEn.0
d8 = TRACEn-1
d9 = TRACEn+1
d13 = TRACEn

1.  if (d5 ==1) d10 = d11    ;else d10 = d10;
2.  if (d4 ==1) d3 = d8      ;else d3 = d9;
3.  if (d6 ==1) d1 = d13     ;else d1 = d3;
4.  if (d4 ==1) d4 = d8      ;else d4 = d13;
5.  if (d6 ==1) d2 = d13     ;else d2 = d4;
6.  if (d5 ==1) d13 = d1     ;else d13 = d2;
```



Example:

		After Instruction	
	ld.d e10,eX0	e10	00000003 0000000F
	ld.d e4,eX1	e4	00000004 00000005
	ld.w d6,lX10	d6	0000 000F
	ld.d e8,eX2	e8	00000005 00000003
	mov.u d15,#1	d15	0000 0001
	ld.w d13,lX11	d13	0000 0004
1.	eq d14,d15,d5	d14	0000 0000
	sel d10,d14,d11,d10	d10	0000 000F
2.	eq d14,d15,d4	d14	0000 0000
	sel d3,d14,d8,d9	d3	0000 0005
3.	eq d14,d15,d6	d14	0000 0000
	sel d1,d14,d13,d3	d1	0000 0005
4.	eq d14,d15,d4	d14	0000 0000
	sel d4,d14,d8,d13	d4	0000 0004
5.	eq d14,d15,d6	d14	0000 0000
	sel d2,d14,d13,d4	d2	0000 0004
6.	eq d14,d15,d5	d14	0000 0000
	sel d13,d14,d1,d2	d13	0000 0004

6 Bit Field Operations

This chapter introduces a new data type: the bit field. While all operations described apply equally to arithmetic data types, the chapter emphasizes the differences:

- A bit-field data type is not a logical, but rather a structural type.
- A bit-field can represent anything: an arithmetic value, a graphics bit map, a printer bit map, a protocol header or a scrambler input.

Powerful bit-field instructions must be able to operate on wide registers and have the flexibility of accessing sub-fields. TriCore meets both criteria.

As a 32-bit CPU, TriCore offers standard 32-bit width for shift operations, but some instructions also have the capability to work on 64-bit fields.

The manipulation of sub-fields is accomplished with three powerful operations: Insertion, extraction and double extraction.

Included in this chapter are the three cases of count leading bit operations (zero, ones and signs). Normalization and converting little-endian to big-endian are shown in applications.

In addition, three application specific instructions that are often used in telecommunications, are covered: bit split, bit merge and parity.

Chapter contents:

- **Logical Shift (SH)**
- **Arithmetic Shift (SHA)**
- **64-bit Shift**
- **Insert (INSERT)**
- **Extract (EXTR , EXTR.U)**
- **Double Extract (DEXTR)**
- **Count Leading bits**
- **Bit Merge & Split (BMERGE, BSPLIT)**
- **Parity (PARITY)**
- **Application Specific**

6.1 Logical Shift (SH)

6.1.1 Left and Right Shifts

TriCore uses the same mnemonic for left and right shifts.

```
sh d0, d1, #3
sh d0, d1, #-3
```

The shift count must be between +31 and -32.

A positive shift count will give a left shift, while a negative count will give a right shift. This is easy to remember since it works like a power of 2.

$\ll 3 \implies$ multiplication by 8 $\implies y = x * 8 \implies y = x * 2^3 \implies \#3$
 $\gg 3 \implies$ division by 8 $\implies y = x * 1/8 \implies y = x * 2^{-3} \implies \#-3$

Example:

d1	F000 0011	
After Instruction		
sh d0,d1,#4	d0 0000 0110	$\ll 4$
sh d0,d1,#-4	d0 0F00 0001	$\gg 4$

16-bit opcode syntax:

```
sh dest/src, shift count.
```

The shift count must be between -8 and +7.

6.1.2 Shift by a Variable Number

The shift count can be a variable number contained in a 32-bit register.

Example:

d0	0101 1010
d1	0000 0004
After Instruction	
sh d2,d0,d1	d2 1011 0100

6.1.3 Packed Halfword Shift (SH.H)

The SH.H instruction is identical to the SH instruction, except that it works *simultaneously* on two 16-bit values instead of one 32-bit value.

Example:

d0	0101 1010
d1	0000 0004

After Instruction

sh.h d3, d0, d1

d3	1010 0100
----	-----------

The shift count must be between –16 and +15. There is only one shift count for two elements.

Constant: An 8-bit constant can replace the second source register (d1 in the example).

6.2 Arithmetic Shift (SHA)

6.2.1 Left and Right Shifts

The rules are identical to logical shift:

- A positive shift count will give a left shift
- A negative count will give a right shift

```
sha d0, d1, #4      ; ; <<4
sha d0, d1, #-4     ; ; >>4
```

6.2.1.1 3 Differences between logical and arithmetic shifts

1. On right shift:

- SH fills the value with zero's
- SHA fills the value with signs.

Example:

(binary used)

d1	1011 0111 0101 0101 1010 1010 1000 0110
----	---

After Instruction

sh d0, d1, #-6

d0	0000 0010 1101 1101 0101 0110 1010 1010
d2	1111 1110 1101 1101 0101 0110 1010 1010

sha d2, d1, #-6

2. The SHA instruction *modifies the carry flag*: psw.c = (0 OR shifted out bits).
This is used mainly in Floating Point Library.

Example:

d1	1999 2001
----	-----------

After Instruction

sha d0, d1, #4

d0	9992 0010
----	-----------

psw.c = 1

sha d0, d1, #-4

d0	0199 9200
----	-----------

psw.c = 1

3. The **OVF** flag is meaningful:

OVF: In case of left shift, if the bits shifted out are not all the same.

16-bit opcode syntax:

sha dest/src, k4

6.2.2 Shift by a Variable Number

Arithmetic shift can be used with a register as a positive or negative shift-count.

Example:

d0	E180 91D1
d1	FFFF FFFC

After Instruction

sha d2, d0, d1

d2	FE18 091D
----	-----------

psw.c = 1

6.2.3 Packed Halfword Arithmetic Shifts (SHA.H)

The SHA.H instruction is identical to the SHA instruction, except that it works *simultaneously* on two 16-bit values instead of one 32-bit value.

The carry is untouched, but the overflow flag is still valid as the OR of any overflows.

Example:

d0	E180 91D1
d1	FFFF FFFC

After Instruction

sha.h d3, d0, d1

d3	FE18 F91D
----	-----------

sha.h d4, d0, #-4

d4	FE18 F91D
----	-----------

The 2, 16-bit data are right shifted by 4.

6.2.4 Arithmetic Shift with Saturation (SHAS)

The SHAS instruction saturates the result if its sign bit differs from the sign bits that are shifted out.

Example:

d1	F490 0000
d0	1001 1001
d2	0000 0004

After Instruction

shas d3, d1, #4

d3	8000 0000
----	-----------

shas d4, d0, d2

d4	7FFF FFFF
----	-----------

Note: SHAS is equivalent to a multiplication by a power of 2 with saturation. But contrary to a Multiplier, it also allows division by a power of 2. This is the case when the shift value is negative. It is equivalent to a normal right arithmetic shift. There cannot be any saturation.

OVF: The bits shifted out are not all the same.

6.3 64-bit Shift

This section describes the procedure for computing the different 64-bit shifts.

6.3.1 64-bit Left Shift

A 64-bit left shift is the combination of a left shift on the lower 32-bit value and a double extraction of both registers to get the upper value.

Example:

In this example d2 is the lower destination register and d3 is the upper destination register.

After Instruction

sh d2, d0, #8 dextr d3, d1, d0, #8	e0	1122 3344 5566 7788
	d2	6677 8800
	d3	2233 4455

6.3.2 64-bit Right Shift

A 64-bit right shift is the combination of a right shift on the upper 32-bit value and a double extraction of both registers to obtain the lower value.

Example:

In this example d2 is the lower destination register and d3 is the upper destination register.

After Instruction

dextr d2, d1, d0, # 24 sh d3, d1, #-8	e0	1122 3344 5566 7788
	d2	4455 6677
	d3	0011 2233

6.3.3 64-bit Arithmetic Left Shift

A 64-bit arithmetic shift left is executed the same way as for a 64-bit left shift, but uses SHA for the lower register.

Example:

		After Instruction	
sha d2,d0,#8 dextr d3,d1,d0,#8	e0	1122 3344 5566 7788	psw.c = 1
	d3	6677 8800	
	d3	2233 4455	

6.3.4 64-bit Arithmetic Right Shift

A 64-bit arithmetic shift right is executed the same way as for a 64-bit right shift, but uses SHA for the upper register.

Example:

		After Instruction	
dextr d2,d1,d0,# 24 sha d3,d1,#-8	e0	1122 3344 5566 7788	
	d3	0011 2233	
	d2	4455 6677	

6.3.5 64-bit Arithmetic Left Shift with Saturation

The 64-bit arithmetic shift left with saturation involves more operations than the normal arithmetic shift, since it corrects results with overflow. If the upper register overflows it must be saturated, and the lower register must be saturated.

Example:

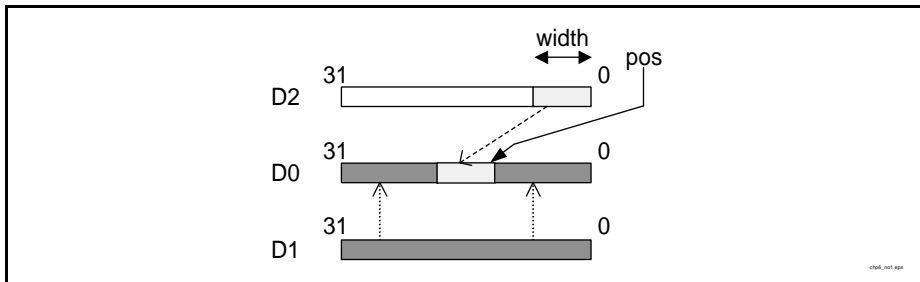
After Instruction			
	e0	1122 3344 5566 7788	
movh d7, #0x8000	d7	8000 0000	
nor d7, d7, #0	d7	7FFF FFFF	;positive saturation
shas d2, d1, #8	d2	7FFF FFFF	;shift the upper value
dextr d3, d1, d0, #8	d3	2233 4455	;shift left of the 64-bit value
eq d4, d2, d7	d4	0000 0001	;check the saturation
movh d5, #0	d5	0000 0000	;allow obtaining 0xFFFFFFFF
nor d5, d5, #0	d5	FFFF FFFF	;if saturation
sel d3, d4, d5, d3	d3	FFFF FFFF	;saturate lower register

6.4 Insert (INSERT)

The INSERT instruction inserts an n -bit field on any bit boundary, into a register at a defined position, *without changing the other bits*.

```
insert    d0,d1,d2,position,width
```

Insert <width> bits of <d2> starting at <position> into <d1> and puts result into <d0>.



Constant: The bits inserted (4-bit constant).

Example:

In the following examples, four bits are extracted at the 24th position and placed into the register d0. There are 6 different syntax. The examples are chosen to give the same result.

position = 24

width = 4

d0	7540 0000
d1	F430 0009
d2	0000 0018
d3	0000 0004

After Instruction

```
insert d4,d0,d1,#24,#4
insert d5,d0,d1,d2,#4
insert d6,d0,#9,#24,#4
insert d7,d0,#9,d2,#4
insert d8,d0,d1,e2
insert d9,d0,#9,e2
```

d4	7940 0000
d5	7940 0000
d6	7940 0000
d7	7940 0000
d8	7940 0000
d9	7940 0000

6.4.1 Corner Cases

There are two corner cases:

- (position + width) is bigger than 32
- The width is equal to 0

Note: In both cases the result is architecturally undefined.

Example:

d0	1122 3344
d1	0000 0055

After Instruction

```
insert d5,d0,d1,#16,#24
```

d5	0055 3344
d6	1122 3344

```
insert d6,d0,d1,#16,#0
```

Note: Sometimes a developer will use the simulator to discover the value of an undefined result! As shown in the example above, the results seem to have some kind of logic and it is then tempting to use. This could happen for instance when the width is given by a variable and is just outside the allowed range. This avoids 1 or 2 cycles (checking the width).

As mentioned, this result is architecturally undefined. It means that the result can vary from one silicon implementation to the next. The crucial point is that software should NEVER rely on an UNDEFINED result.

6.4.2 Duplicate

The INSERT instruction can be used to duplicate a 16-bit value in a 32-bit register. The source operands are identical; the value must be in the lower half of the register, with the position and width equal to 16.

Example: Duplicate 0x9876

d0	0000 9876
----	------------------

After Instruction

```
insert d1,d0,d0,#16,#16
```

d1	9876 9876
----	------------------

6.4.3 Clear or Set a Bit Field

The INSERT instruction can be used to clear or set a bit field:

Example: Clear or set bits 16 through 21 of d0

d0	1122 3344
----	-----------

After Instruction

```
mov.u d2,#0xFFF
```

d2	0000 0FFF
----	-----------

```
insert d3,d0,#0,#16,#6
```

d3	1100 3344
----	-----------

;clear bits 16 through 21 of d0

```
insert d4,d0,d2,#16,#6
```

d4	113F 3344
----	-----------

;set bits 16 through 21 of d0

Note: The instruction extends the constant only if it is zero. Accordingly, the value 0xFFF is in a register instead of 0x1.

6.4.4 32-bit Shuffle

Of all possible combinations of halfword shuffling in a 32-bit register, four cases exist.

If d0 is the destination and d1 and d2 are the sources:

	JoinLL	JoinLH	JoinHL	JoinHH
d2	abcd	abcd	abcd	abcd
d1	ABCD	ABCD	ABCD	ABCD
d0	cdCD	ABcd	abCD	abAB

The first three cases are implemented with a simple INSERT instruction:

```
insert d0,d1,d2,#16,#16 ; JoinLL
insert d0,d1,d2,#0,#16  ; JoinLH
insert d0,d2,d1,#0,#16  ; JoinHL
```

The fourth case requires a shift before the insertion:

```
sh d3,d1,#-16
insert d0,d2,d3,#0,#16 ; JoinHH
```


6.4.5 64-bit Interleave

There are 6 possible combinations of interleaving in a 64-bit register. Left or right bytes can be interleaved, or left / right halfwords, or left / right words.

	MixL8	MixR8	MixL16	MixR16	MixL32	MixR32
e4	abcdefgh	abcdefgh	abcdefgh	abcdefgh	abcdefgh	abcdefgh
e2	ABCDEFGHGH	ABCDEFGHGH	ABCDEFGHGH	ABCDEFGHGH	ABCDEFGHGH	ABCDEFGHGH
e0	aAcCeEgG	bBdDfFhH	abABefEF	cdCDghGH	abcdABCD	efghEFGH

The first case is computed as follows:

Note: This example could be performed quicker using SH, AND and OR's)

Example MixL8

```

mov      d6, #0
sh       d7, d2, #-8
insert   d0, d4, d7, d6, #8      ; d6 = 0
add      d6, d6, #16
sh       d7, d7, #-16
insert   d0, d0, d7, d6, #8      ; d6 = 16
sub      d6, d6, d6
sh       d7, d3, #-8
insert   d1, d5, d7, d6, #8      ; d6 = 0
add      d6, d6, #16
sh       d7, d7, #-16
insert   d1, d1, d7, d6, #8      ; d6 = 16

```

The second case is computed as follows:

Example MixR8

```

mov      D6, #8
insert   d0, d2, d4, d6, #8      ; d6 = 8
sh       d7, d4, #-16
add      d6, d6, #16
insert   d0, d0, d7, d6, #8      ; d6 = 24
add      d6, d6, #-16
insert   d1, d3, d5, d6, #8      ; d6 = 8
sh       d7, d5, #-16
add      d6, d6, #16
insert   d1, d1, d7, d6, #8      ; d6 = 24

```

The third case is computed as follows:

Example MixL16

```
sh          d6, d2, #-16
insert     d0, d4, d6, #0, #16 ; JoinHH
sh          d6, d3, #-16
insert     d1, d5, d6, #0, #16 ; JoinHH
```

The fourth case is computed as follows:

Example MixR16

```
insert     d0, d2, d4, #16, #16 ; JoinLL
insert     d1, d3, d5, #16, #16 ; JoinLL
```

The fifth and sixth cases are computed with a move:

Example MixL32

```
mov        d1, d5
mov        d0, d3 ; MixL32
```

Example MixR32

```
mov        d1, d4
mov        d0, d2 ; MixR32
```

These interleave operations can be extended to a pair of 64-bit. The following table shows the 3 possible combinations: Left and right bytes, left and right halfwords, left and right words.

	MixPair8		MixPair16		MixPair32	
e4	abcdefgh		abcdefgh		abcdefgh	
e2	ABCDEFGHGH		ABCDEFGHGH		ABCDEFGHGH	
e8/e6	aAcCeEgG	bBdDfFhH	abABefEF	cdCDghGH	abcdABCD	efghEFGH

The first case is a combination of MixR8 and MixL8:

Example MixPair8

```

mov          d0, #8
insert       d6, d2, d4, d0, #8      ; d0 = 8
sh           d1, d4, #-16
add          d0, d0, #16
insert       d6, d6, d1, d0, #8      ; d0 = 24
add          d0, d0, #-16
insert       d7, d3, d5, d0, #8      ; d0 = 8
sh           d1, d5, #-16
add          d0, d0, #16
insert       d7, d7, d1, d0, #8      ; d0 = 24
mov          d0, #0
sh           d1, d2, #-8
insert       d8, d4, d1, d0, #8      ; d0 = 0
add          d0, d0, #16
sh           d1, d1, #-16
insert       d8, d8, d1, d0, #8      ; d0 = 16
sub          d0, d0, d0
sh           d1, d3, #-8
insert       d9, d5, d1, d0, #8      ; d0 = 0
add          d0, d0, #16
sh           d1, d1, #-16
insert       d9, d9, d1, d0, #8      ; d0 = 16

```

The second case is a combination of MixR16 and MixL16:

Example MixPair16

```

insert       d6, d2, d4, #16, #16    ; JoinLL
insert       d7, d3, d5, #16, #16    ; JoinLL
sh           d1, d2, #-16
insert       d8, d4, d1, #0, #16     ; JoinHH
sh           d1, d3, #-16
insert       d9, d5, d1, #0, #16     ; JoinHH

```

The third case is a combination of MixR32 and MixL32:

Example MixPair32

```

mov          d9, d5
mov          d8, d3                  ; MixR32
mov          d7, d4
mov          d6, d2                  ; MixL32

```

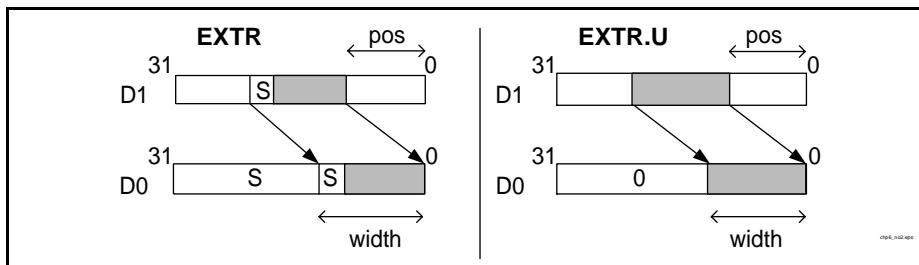
6.5 Extract (EXTR , EXTR.U)

The EXTR instruction extracts and aligns n-bit field on any bit boundary.

```
extr    d0,d1,position,width
```

```
extr.u  d0,d1,position,width
```

Extract <width> bits of <d1> starting at <position> from <d1>, put the result in <d0>, and sign-extend (or zero-extend) the result.



Example:

In the following example, four bits are extracted from the 20th bit of register d0. Note that there are 3 different syntax for each. The examples have been chosen to give the same result.

d0	F490 0000
d2	0000 0014
d3	0000 0004

After Instruction

```
extr    d1,d0,#20,#4
```

```
extr    d4,d0,d2,#4
```

```
extr    d4,d0,e2
```

```
extr.u  d5,d0,#20,#4
```

```
extr.u  d6,d0,d2,#4
```

```
extr.u  d6,d0,e2
```

d1	FFFF FFF9
d4	FFFF FFF9
d4	FFFF FFF9
d5	0000 0009
d6	0000 0009
d6	0000 0009

; position in d2
; width in d3

; position in d2
; width in d3

6.5.1 Corner Cases

There are two corner cases:

- (position + width) is bigger than 32
- Width is equal to 0

Note: In both of these cases, the result is architecturally undefined.

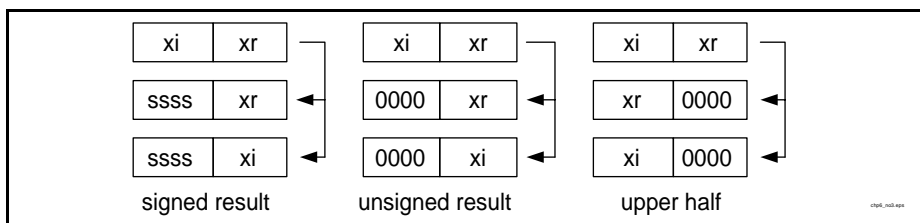
Example:

	d0	1122 3344
After Instruction		
extr d1,d0,#24,#16	d1	0000 0011
extr d2,d0,#24,#0	d2	0000 0000
extr d3,d0,#0,#31	d3	1122 3344

Note: As mentioned, this result is architecturally undefined. It means that the result can vary from one silicon implementation to the next. The crucial point therefore is that software should NEVER rely on an UNDEFINED result.

6.5.2 Unpack 16-bit Values

Sometimes two 16-bit values regrouped in a 32-bit register need to be unpacked:



If the values are signed:

```
extr      d1,d0,#0,#16      ; extract xr and sign-extend it
extr      d2,d0,#16,#16     ; extract xi and sign-extend it
```

If the values are unsigned:

```
extr.u    d3,d0,#0,#16      ; extract xr
extr.u    d4,d0,#16,#16     ; extract xi
```

If the values are in the upper half, the sequence is:

```
sh        d5,d0,#16         ; extract xr
insert    d6,d0,#0,#0,#16   ; extract xi
```

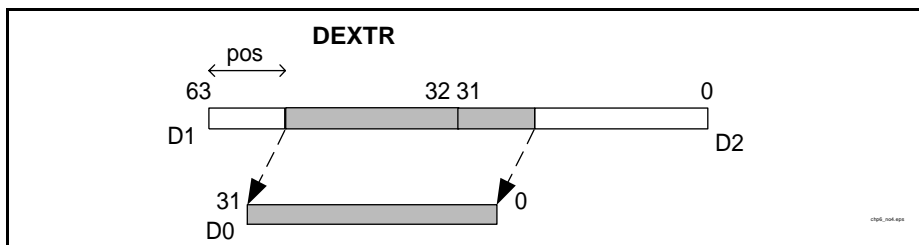
6.6 Double Extract (DEXTR)

The DEXTR instruction extracts 32 bits from 64 bits on any bit boundary.

- This is the only ALU instruction which has a 64-bit field as a source.
- The two 32-bit registers constituting the 64-bit field *DO NOT NEED* to be a register pair.

`dextr d0,d1,d2,position`

Extract 32 bits starting at <position> of <d1> (through <d2>) and put the result in <d0>.



Example:

d1 and d9 are source registers. Only the 28 least significant bits of d1 and the 4 most significant bits of d9, are kept.

- The first example uses a constant position.
- The second example uses a variable position.

d9	7540 0000
d1	F490 0000
d3	0000 0004

After Instruction

`dextr d2,d1,d9,#4`

d2	4900 0007
d4	4900 0007

`dextr d4,d1,d9,d3`

6.6.1 Corner Cases

There are 2 corner cases:

- Position is > 31
- Position is = 0

The first of these cases is architecturally undefined. In the second case the result will be a complete register (which is equivalent to a move instruction).

Example:

d1	0000 0011
d0	1122 3344

After Instruction

dextr d7, d1, d0, #0

d7	0000 0011
----	-----------

6.6.2 Rotate

The DEXTR instruction can be used to rotate a 32-bit register.

It is important to have 2 identical source operands.

Example:

This example shows 2 ways to rotate d0 by 12.

d0	1234 5678
d1	0000 000C

After Instruction

dextr d2, d0, d0, #12

d2	4567 8123
----	------------------

dextr d2, d0, d0, d1

d2	4567 8123
----	------------------

6.6.3 Swap

The DEXTR instruction is used to swap two 16-bit halves of a 32-bit register. The two source registers are identical and the width must be 16.

Example:

d0	1234 5678
----	------------------

After Instruction

dextr d1, d0, d0, #16

d1	5678 1234
----	------------------

6.6.4 Normalization

The DEXTR instruction is used to normalize the result of a DSP filter accumulation, in which a 64-bit accumulator is used with several guard bits. The value of the position is determined by using the CLS instruction (See [Application Specific](#)).

6.7 Count Leading bits

There are three different count-leading cases:

- **Counting Zeros (CLZ, CLZ.H)**
- **Counting Ones (CLO, CLO.H)**
- **Counting Signs (CLS, CLS.H)**

6.7.1 Counting Zeros (CLZ, CLZ.H)

Counting zeros is performed with the CLZ instruction. The instruction also exists with packed halfword.

Example:

d0	054E 6F01 <u>00000</u> 10110001110 <u>0</u> 110111100000001
----	--

After Instruction

clz d2, d0

d2	0000 0005
----	-----------

clz.h d4, d0

d4	0005 0001
----	-----------

6.7.2 Counting Ones (CLO, CLO.H)

Counting ones is performed with the CLO instruction. The instruction also exists with packed halfword.

Example:

d1	FE85 A781 <u>1111111</u> 1010000101 <u>1</u> 010011110000001
----	---

After Instruction

clo d3, d1

d3	0000 0007
----	-----------

clo.h d4, d1

d4	0007 0001
----	-----------

6.7.3 Counting Signs (CLS, CLS.H)

Counting signs is performed with the CLS instruction. The instruction also exists with packed halfword.

Note: The result is a 1 subtracted from the result of a CLO or CLZ.

Example:

d0	054E 6F01
d1	FE15 C331

After Instruction

cls d2, d0

d2	0000 0004
----	-----------

cls d3, d1

d3	0000 0006
----	-----------

cls.h d4, d1

d4	0006 0001
----	-----------

6.7.4 Corner Cases

Each count-leading instruction has two corner cases:

- when the value is all zeros
- when the value is all ones

Both cases give meaningful results with the 3 instructions.

Example:

d0	FFFF FFFF
d1	0000 0000

After Instruction

clz d3, d1

d3	0000 0020
----	-----------

clz d4, d0

d4	0000 0000
----	-----------

clo d5, d0

d5	0000 0020
----	-----------

clo d6, d1

d6	0000 0000
----	-----------

cls d8, d1

d8	0000 0000
----	-----------

cls d9, d0

d9	0000 001F
----	-----------

6.8 Application Specific

6.8.1 Bit Merge & Split (BMERGE, BSPLIT)

The BMERGE instruction is used to interleave 2 bitstreams.

`bmerge d0, d1, d2`

Even bits of d0 = bits [0,1,2,3...15] of d1

Odd bits of d0 = bits [0,1,2,3...15] of d2

BSPLIT is the reverse operation to BMERGE.

`bsplit d0/d1, d2`

bits [0,1,2,3...15] of d0 = even bits of d2

bits [0.1.2.3...15] of d1 = odd bits of d2



Note: The even destination register receives the even bits. The odd destination register receives the odd. The destination is always a register pair. The 2 unused upper 16-bits of the destination register pair are cleared.

Example:

d6	0005 6123
----	-----------

After Instruction

`bsplit e4, d6`

e4	00000045 00000391
d7	0005 6123

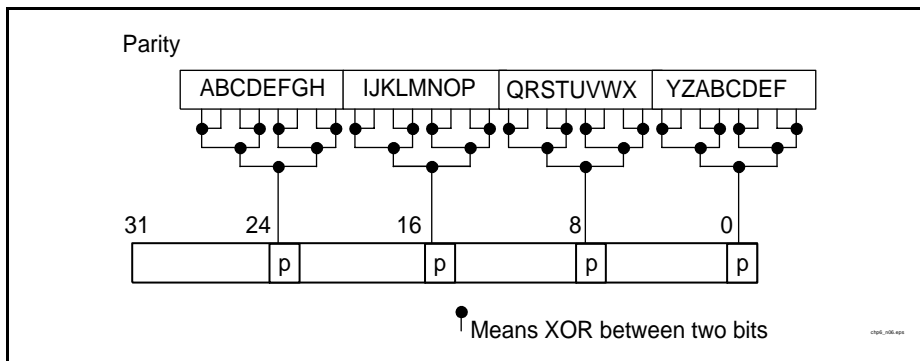
`bmerge d7, d5, d4`

6.8.2 Parity (PARITY)

For each byte, the PARITY instruction will set a parity bit if there is an odd number of '1'.

`parity dest, src`

Compute a XOR on each bit of the four bytes and give four results.



Example:

d6	93FE 85FF
----	-----------

After Instruction

`parity d8, d6`

d8	0001 0100
----	-----------

6.9 Applications

Normalization, left shift filled with 1's, converting little-endian to big-endian and counting high bits, all demonstrate the application of previous instructions.

6.9.1 Normalization

Normalization allows for the maximum precision on values. The count leading sign counts the number of non magnitude bits (this is equivalent to the number of sign bits minus 1). The value is then left aligned.

Example:

		After Instruction	
<code>ld.w</code>	<code>d0, lX21</code>	d0	0001 054F
<code>cls</code>	<code>d1, d0</code>	d1	0000 000E
<code>sh</code>	<code>d2, d0, d1</code>	d2	4153 C000

6.9.2 Left Shift Filled with 1s

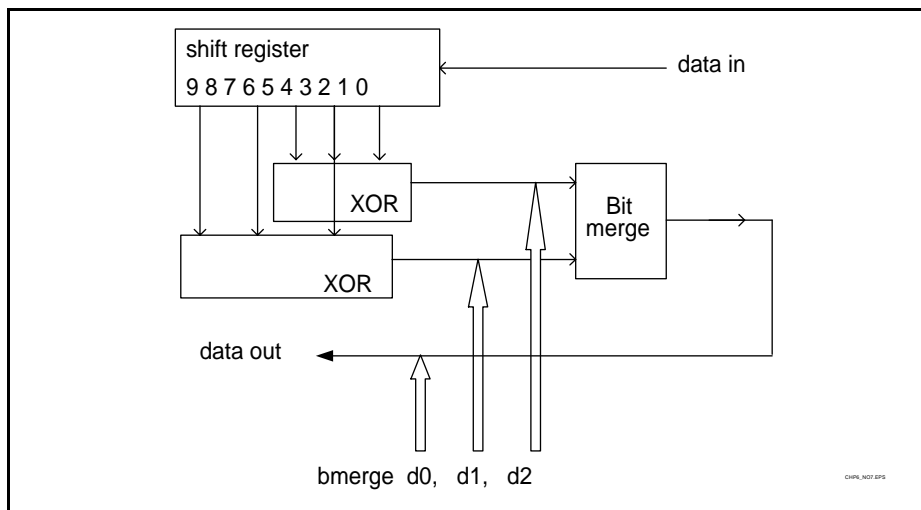
Left shift is performed by the INSERT instruction. The aim is to fill the least significant bits with a different value other than 0. This is used to give greater precision than filling with zeros.

Example:

		After Instruction	
<code>ld.w</code>	<code>d0, lX16</code>	d0	1234 5678
<code>mov.u</code>	<code>d1, #15</code>	d1	0000 000F
<code>insert</code>	<code>d2, d0, d1, #4, #28</code>	d2	2345 678F

6.9.3 Convolutional coder

Most convolutional coders and LFSRs need to merge two streams of bits. This operation is often (incorrectly) represented as an AND gate in the literature. It is actually an interleave (1/1) of 2 bit streams. This operation could typically represent 70-100 cycles in software (on 32-bit block), while it is only 2 cycles by using BMERGE.



6.9.4 Converting Little-endian to Big-endian

d1	0000 0000
----	-----------

After Instruction

```
ld.w    d0,1X16
```

```
mov.u    d2, #24
```

```
lea    a2,3
```

lazy:

```
insert
d1,d1,d0,d2,#8
```

```
sh    d0,d0,#-8
```

```
add    d2,d2,#-8
```

```
loop    a2, lazy
```

d0	1234 5678
d2	0000 0018
a2	0000 0003

1st pass

2nd pass

3rd pass

4th pass

d1	7800 0000	7856 0000	7856 3400	7856 3412
d0	0012 3456	0000 1234	0000 0012	0000 0000
d2	0000 0010	0000 0008	0000 0000	FFFF FFF8
a2	0000 0002	0000 0001	0000 0000	0000 0000

6.9.5 Counting High Bits

The calculation is performed on each byte using packed instructions. To count the number of 1s in 8 bits:

- The original word is ANDed with 0x55.
- The 8-bit result consists of four 2-bit words. Each of these 2-bit words contains the count of the number of 1s in the least significant bit of the original word's corresponding 2-bit field. The word is shifted right by 1 bit and again ANDed with 0x55.
- The resultant group of 2-bit words now contains the count of the number of 1s in the most significant bit of the corresponding 2-bit field in the original word.

Adding the result of these two AND operations produces four 2-bit words which each contains the count of the number of 1s in the corresponding 2-bit field of the original word.

Using this sum, the above process is repeated except the sum is ANDed with 0x33 and shifted by two, instead of by one. The result is an 8-bit sum of two four bit words, where each 4-bit word contains the count of the number of 1s in the corresponding 4-bit field in the original word.

Use this second sum and repeat the process, except AND with 0x0F and shift by four instead of two. The final result, the third 8-bit sum, contains the number of 1s in the entire 8-bit original word.

After Instruction

<code>lea</code>	<code>a2, coef</code>	<code>a2</code>	0000 0038
<code>ld.w</code>	<code>d0, lX13</code>	<code>d0</code>	1122 3344
<code>ld.w</code>	<code>d1, [a2+]</code>	<code>d1</code>	5555 5555
<code>and</code>	<code>d2, d0, d1</code>	<code>d2</code>	1100 1144
<code>sh.h</code>	<code>d3, d0, #-1</code>	<code>d3</code>	0891 19A2
<code>and</code>	<code>d3, d3, d1</code>	<code>d3</code>	0011 1100
<code>add.h</code>	<code>d0, d2, d3</code>	<code>d0</code>	1111 2244
<code>ld.w</code>	<code>d1, [a2+]</code>	<code>d1</code>	3333 3333
<code>and</code>	<code>d2, d0, d1</code>	<code>d2</code>	1111 2200
<code>sh.h</code>	<code>d3, d0, #-2</code>	<code>d3</code>	0404 0891
<code>and</code>	<code>d3, d3, d1</code>	<code>d3</code>	0000 0011

<code>add.h</code>	<code>d0, d2, d3</code>	<code>d0</code>	<code>1111 2211</code>
<code>ld.w</code>	<code>d1, [a2+]</code>	<code>d1</code>	<code>0F0F 0F0F</code>
<code>and</code>	<code>d2, d0, d1</code>	<code>d2</code>	<code>0101 0201</code>
<code>sh.h</code>	<code>d3, d0, #-4</code>	<code>d3</code>	<code>0111 0221</code>
<code>and</code>	<code>d3, d3, d1</code>	<code>d3</code>	<code>0101 0201</code>
<code>add.h</code>	<code>d0, d2, d3</code>	<code>d0</code>	<code>0202 0402</code>
<code>extr.u</code>	<code>d5, d0, #0, #8</code>	<code>d5</code>	<code>0000 0002</code>
<code>extr.u</code>	<code>d6, d0, #8, #8</code>	<code>d6</code>	<code>0000 0004</code>
<code>extr.u</code>	<code>d7, d0, #16, #8</code>	<code>d7</code>	<code>0000 0002</code>
<code>extr.u</code>	<code>d8, d0, #24, #8</code>	<code>d8</code>	<code>0000 0002</code>
<code>add</code>	<code>d5, d5, d6</code>	<code>d5</code>	<code>0000 0006</code>
<code>add</code>	<code>d7, d7, d8</code>	<code>d7</code>	<code>0000 0004</code>
<code>add</code>	<code>d6, d5, d7</code>	<code>d6</code>	<code>0000 000A</code>

7 Boolean Processing

This chapter covers the most elementary of all data types: the BIT, also called BOOLEAN in high level language.

Desktop CPUs have very little or no boolean processing. It is only amongst controllers (also called MCUs), that this class of processing is found. TriCore is a unified CPU/DSP/MCU.

A bit can only be processed in four ways: pass, invert, set, and clear. This is true for the processing of one bit, but for the processing of two bits, the processing of two bits with accumulation, and the processing of two bits in a series of bits; the ways become virtually endless.

This chapter introduces bit insert instructions, single operand Boolean functions, followed by two operand functions. Finally, the more sophisticated features offered by TriCore, such as accumulated bit processing and shifted bit processing, are covered. Direct applications of these instructions, logic gates and state machine are also included.

Chapter contents:

- **Bit Insert**
- **Single-operand Boolean Functions**
- **Bit Set**
- **2-Operand Boolean Functions**
- **2-Operand Boolean Functions (With Shift)**
- **3-Operand Boolean Functions**
- **Summary Table: Five Instances of ANDing Bits**
- **Applications**

7.1 Bit Insert

TriCore offers 2 variants of the function: a simple insertion (INS.T) and an insertion preceded by a negation of the bit (INSN.T).

7.1.1 Bit Insert (INS.T)

Used to insert any single bit in any position.

Example:

d0	F2505678
d1	DEAD BEEF

After Instruction

`ins.t`
`d3,d1:10,d0:18`

d3	DEAD BEAF
----	-----------

7.1.2 Bit Negate and Insert (INSN.T)

This is a more powerful instruction which negates the selected bit before re-insertion. This instruction can also be used to negate a bit (cf.later).

Example:

Bit 3 of register d0 is negated and inserted in place of bit 16 of register d1. The result is in register d3.

d0	F2505678
d1	55555555

After Instruction

`insn.t`
`d3,d1:16,d0:3`

d3	55545555
----	----------

7.2 Single-operand Boolean Functions

There are only four operations which can be applied to a bit:

- Pass (do not touch)
- Invert
- Set
- Clear

These bit operations are not directly available on TriCore and must be synthesized.

7.2.1 Bit Invert

This is achieved by implementing XOR with 1, and masking all other bits.

Example: Invert bit 4 of d0

```
xor      d0,d0,#0x10
```

TriCore constants are limited to 8 bits and only bits 0..7 can be inverted.

To invert bits in the range 0..15, two instructions are needed:

Example: Invert bit 10 of d0

```
mov.u    d15,#0x0400
xor      d0,d0,d15
```

Similarly, for the bits 16 to 31:

Example: Invert bit 20 of d0

```
movh     d15,#0x0010
xor      d0,d0,d15
```

A general method is to use XOR.T followed by insert. This requires two instructions:

Example: Invert bit 27 of d0

```
nand.t   d15,d0:27,d0:27 ;bitneg bit27 of d0
insert   d0,d0,d15,#27,#1 ;and reinsert in d0
```

If the position is unknown, it will take 3 instructions:

Example: Invert bit 27 of d0

```
extr     d2,d15,d5,#1      ;extract the bit at position d5 of d15
nand.t   d2,d2:0,d2:0      ;invert it
insert   d15,d15,d2,d5,#1  ;reinsert it at position d5 in d15
```

Otherwise, use INSN.T:

Example: Invert bit 27 of d0

```
insn.t   d0,d0:27,d0:27 ;
```

7.2.2 Bit Set

This requires one insert instruction.

Examples:

```
insert    d0,d0,#1,#0,#1    ;bitset bit 0 of d0
insert    d0,d0,#1,#31,#1   ;bitset bit 31 of d0
insert    d4,d4,#1,#15,#1   ;bitset bit 15 of d4
insert    d15,d15,#1,#4,#1  ;bitset bit 4 of d15
```

7.2.3 Bit Clear

This requires one insert instruction.

Examples:

```
insert    d0,d0,#0,#0,#1    ;bitclear bit 0 of d0
insert    d0,d0,#0,#31,#1   ;bitclear bit 31 of d0
insert    d0,d0,#0,#16,#1   ;bitclear bit 16 of d0
```

Note: The last field being 1 implies that 1 bit is cleared (or set). To clear N contiguous bits, change this field N.

Example: Clear bits 15,16,17 of d0

```
insert    d0,d0,#0,#15,#3
```

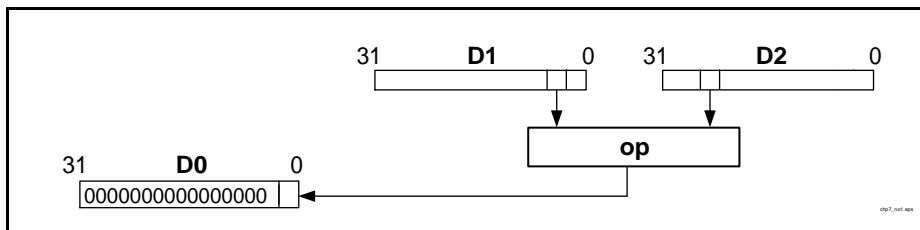
Note: Strictly speaking this is not a boolean operation but is part of bit-field operations (See [Bit Field Operations - Chapter 6](#)).

7.3 2-Operand Boolean Functions

Important 2-operand operations are add, sub and multiply. The truth table for boolean add, boolean sub and boolean multiply show that these are identical to XOR, XOR and AND.

2-operand boolean functions are not the ANDing of 2 bits in the same place between 2 registers. As shown below, the operations are performed on *2 independent bits* and the result put in bit 0 of a third register:

```
<op>/t    d0, d1, d2
```



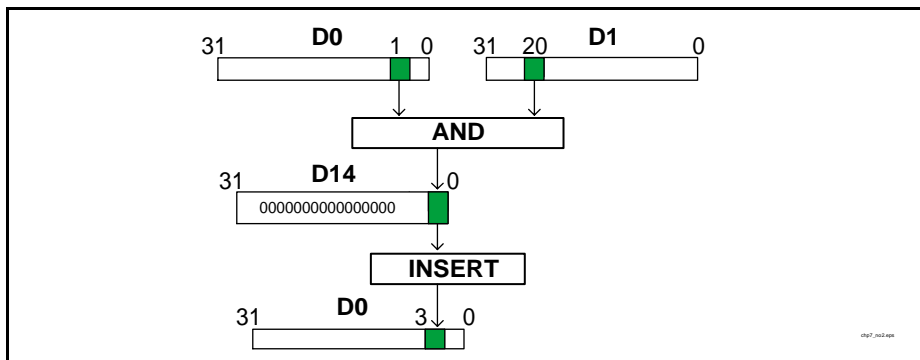
7.3.1 Bit and (AND.T)

For the ANDing of 2 bits, use AND.T.

```
and.t      d14, d0:20, d1:1    ;
```

Bit 1 of register d1 is ANDed with bit 20 of register d0. The result is a boolean (or flag) in bit 0 of register 14. All other bits are cleared. To put the result into a third register at position 3 for example, use an insert:

```
and.t      d14, d0:20, d1:1    ;
insert     d0, d0, d14, #3, #1 ;
```



7.3.2 Bit or (OR.T)

For the ORing of 2 bits, use OR.T.

```
or.t      d14,d0:20,d1:1    ;
```

7.3.3 Bit nand, nor (NAND.T, NOR.T)

The **nand** function is equivalent to not (and (a, b))

The **nor** function is equivalent to not (or (a, b)).

```
nand.t    d14,d0:20,d1:1    ;
```

```
nor.t     d14,d0:20,d1:1    ;
```

7.3.4 Bit andn, orn (ANDN.T, ORN.T)

The **andn** function is equivalent to and (a, (not) b).

The **orn** function is equivalent to or (a, (not) b).

```
andn.t    d14,d0:20,d1:1    ;
```

```
orn.t     d14,d0:20,d1:1    ;
```

7.3.5 Bit xor, xnor (XOR.T, XNOR.T)

The **xorn** function is equivalent to: $\text{xor}(a, (\text{not})b)$

which is equivalent to: $\text{or}((\text{not})a,b)$

which is also equivalent to: $\text{not}(\text{xor}(a,b))$

```
xor.t     d14,d0:20,d1:1    ;
```

```
xorn.t    d14,d0:20,d1:1    ;
```

7.3.6 Summary of Examples

d0	0000 0010
d1	0001 054F

After Instruction

and.t	d2,d0:2,d1:6	d2	0000 0000
or.t	d3,d0:2,d1:6	d3	0000 0001
nand.t	d7,d0:2,d1:6	d7	0000 0001
nor.t	d9,d0:2,d1:6	d9	0000 0000
andn.t	d5,d0:2,d1:6	d5	0000 0000
orn.t	d6,d0:2,d1:6	d6	0000 0000
xnor.t	d8,d0:2,d1:6	d8	0000 0000
xor.t	d4,d0:2,d1:6	d4	0000 0001

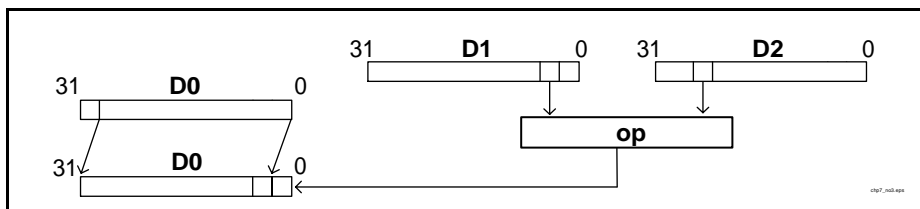
7.4 2-Operand Boolean Functions (With Shift)

This section demonstrates how to accumulate bit results with 2-operand functions.

7.4.1 SH.op.T

The diagram below shows a 2-operand boolean function redrawn with the addition of the shift parameter. The operation and resulting bit are the same. The difference is that the resulting bit is now part of a 'history register' consisting of a series of former boolean operations. The left-most bit (the oldest), has been shifted out to make space for the most recent result.

sh.<op>.t d0, d1, d2



As shown in the example which follows, all eight 2-operand boolean functions can have the shift prefix.

Example:

d0	0000 0010
d1	0001 054F

Before Instruction

After Instruction

sh.and.t	d2, d0:2, d1:6	d2	0000 0001	d2	0000 0002
sh.andn.t	d3, d0:2, d1:6	d3	0000 0001	d3	0000 0002
sh.or.t	d4, d0:2, d1:6	d4	0000 0001	d4	0000 0003
sh.orn.t	d5, d0:2, d1:6	d5	0000 0001	d5	0000 0002
sh.nor.t	d6, d0:2, d1:6	d6	0000 0001	d6	0000 0002
sh.nand.t	d7, d0:2, d1:6	d7	0000 0001	d7	0000 0003
sh.xor.t	d8, d0:2, d1:6	d8	0000 0001	d8	0000 0003
sh.xnor.t	d9, d0:2, d1:6	d9	0000 0001	d9	0000 0002

7.5 3-Operand Boolean Functions

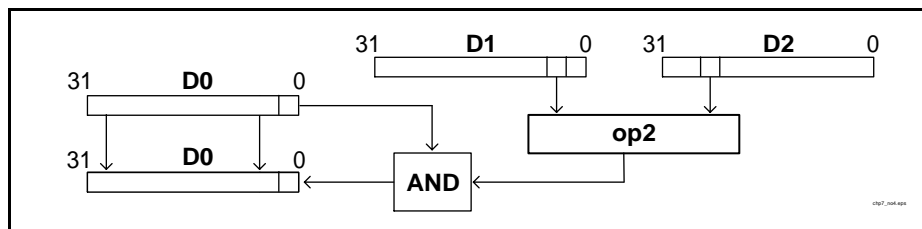
To accelerate the computation of complex conditional expressions, 'accumulating' versions of the boolean functions are supported.

7.5.1 AND.op2.T

The second operation (op2) is applied to the bits, as indicated by the syntax. Only the least significant bit of the destination register is ANDed with the result of the op2 operation.

op2 can be: AND, ANDN, OR, NOR

AND.<op2>.t d0, d1, d2

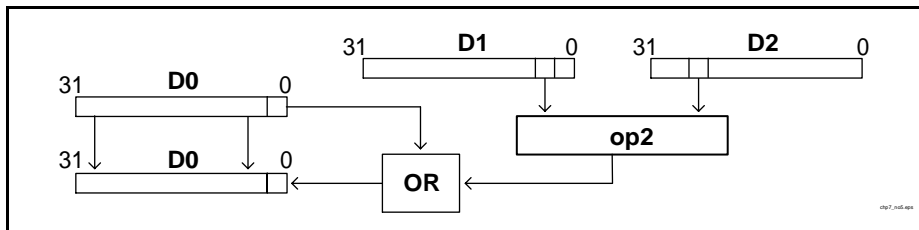


7.5.2 OR.op2.T

The Least Significant Bit (LSB) of the destination register is ORed with the result of the op2 operation.

op2 can be: AND, ANDN, OR, NOR

OR.<op2>.t d0,d1,d2



The examples show the eight possible forms of accumulating instructions.

Example:

d0	0000 0010
d1	0001 054F

Before Instruction

After Instruction

and.and.t d2,d0:2,d1:6

d2	1234 5679
----	-----------

d2	1234 5678
----	-----------

and.andn.t d3,d0:2,d1:6

d3	1234 5679
----	-----------

d3	1234 5678
----	-----------

and.or.t d4,d0:2,d1:6

d4	1234 5679
----	-----------

d4	1234 5679
----	-----------

and.nor.t d5,d0:2,d1:6

d5	1234 5679
----	-----------

d5	1234 5678
----	-----------

or.and.t d6,d0:2,d1:6

d6	1234 5679
----	-----------

d6	1234 5679
----	-----------

or.andn.t d7,d0:2,d1:6

d7	1234 5679
----	-----------

d7	1234 5679
----	-----------

or.or.t d8,d0:2,d1:6

d8	1234 5679
----	-----------

d8	1234 5679
----	-----------

or.nor.t d9,d0:2,d1:6

d9	1234 5679
----	-----------

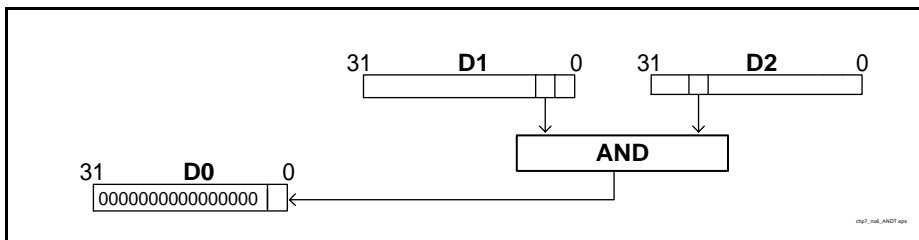
d9	1234 5679
----	-----------

7.6 Summary Table: Five Instances of ANDing Bits

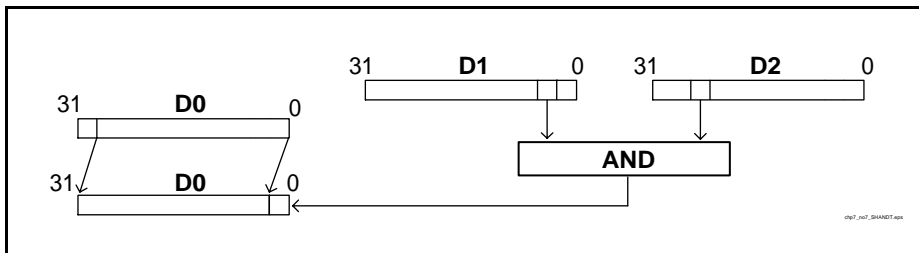
TriCore offers five possible AND processing of boolean variables:

- **AND.T**
- **SH.AND.T**
- **OR.AND.T**
- **AND.op.T**
- **AND.comp**

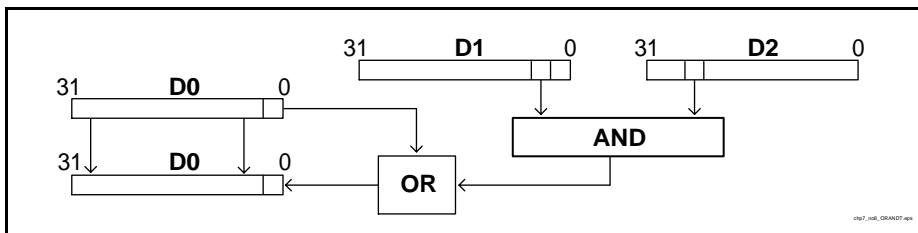
7.6.1 AND.T



7.6.2 SH.AND.T

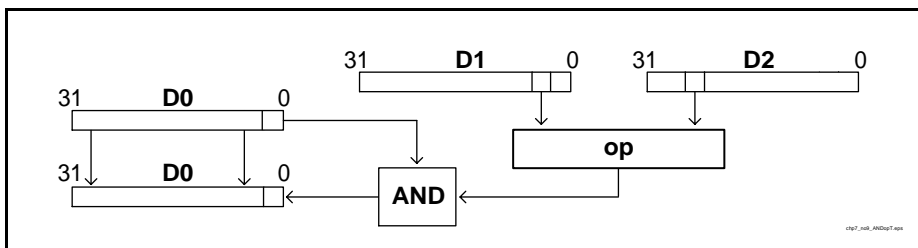


7.6.3 OR.AND.T



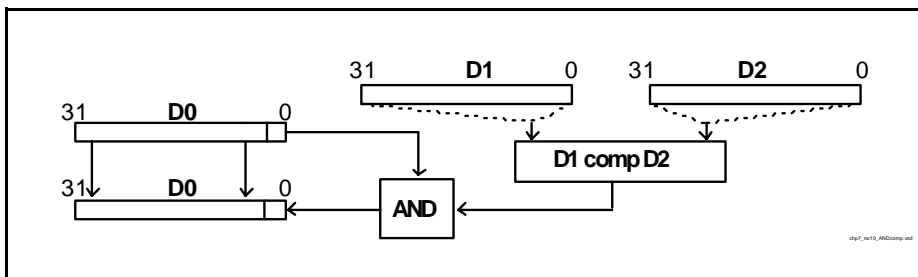
7.6.4 AND.op.T

Where <op> can be AND, ANDN, OR, NOR



7.6.5 AND.comp

Where <comp> can be eq, ne, ge, ge.u, lt, lt.u



(See also the [Logical and Compare](#) chapter for this instruction)

Similar tables can be drawn for OR, XOR.

7.7 Applications

This section provides examples which demonstrate the use of the TriCore bit instruction to develop logic gates and typical state machine. This section also illustrates bit testing.

7.7.1 Logic Gates

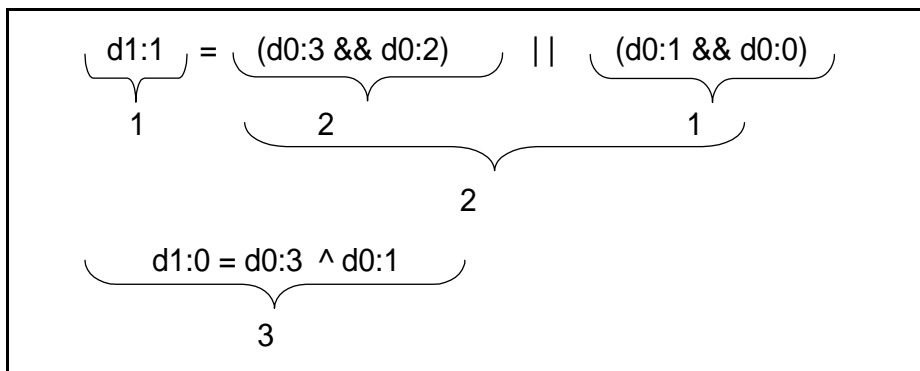
The power of the TriCore bit instruction to develop logic gates is easily demonstrated in the following example:

2 equations:

$$d1:1 = (d0:3 \&\& d0:2) \ || \ (d0:1 \&\& d0:0)$$

$$d1:0 = d0:3 \wedge d0:1$$

These will take only 3 cycles:



The assembly code will be:

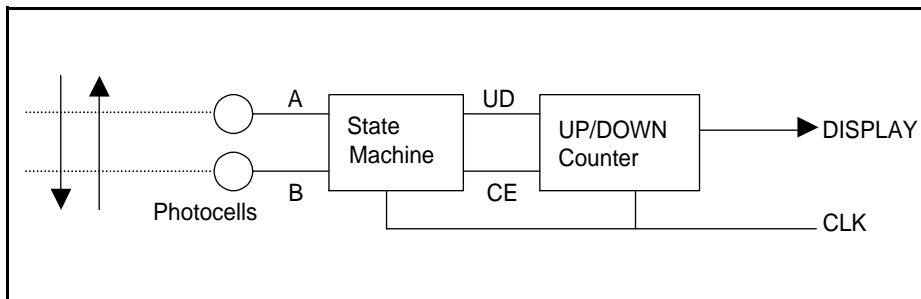
d0	FFFF BEAF
----	-----------

After Instruction

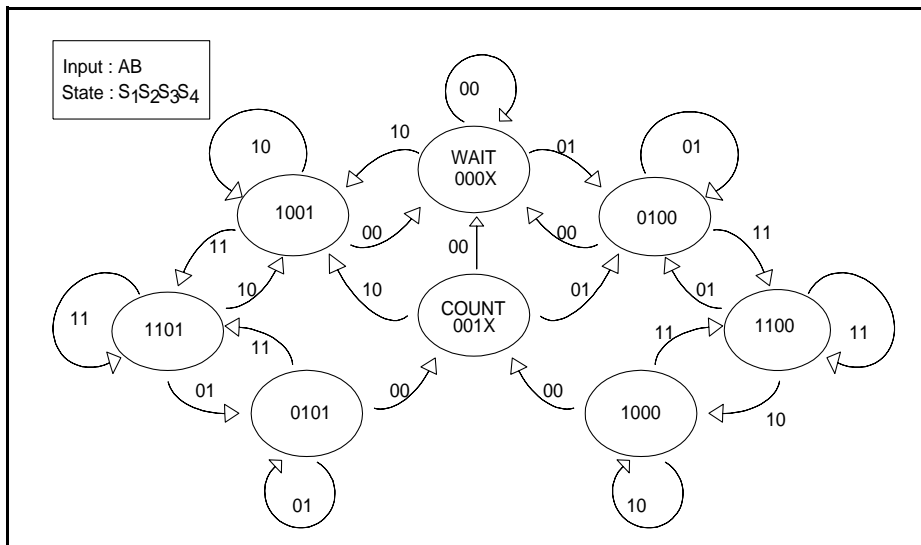
and.t d1,d0:0,d0:1	d1	0000 0001
or.and.t d1,d0:2,d0:3	d1	0000 0001
sh.xor.t d1,d0:1,d0:3	d1	0000 0002

7.7.2 Typical State Machine

This simple state machine is used to adapt the output of two photocells controlling an up/down counter.



State Diagram:



S_1 : Past of A

S_2 : Past of B

S_3 : Counter Enable

S_4 : Counter UP or DOWN

Equations: $S_1^+ = A$

$S_2^+ = B$

$S_3^+ = /A./B./(S1.S2.S4+/S1./S2./S4)$

$S_4^+ = /S1./S2.A+/(S1.S2).S4$

Example:

d0	0000 0003
d1	0000 0005

After Instruction

<code>nor.t d4,d0:3,d0:1</code>	d4	0000 0000	;d4:0 = /(0+A)=/A
<code>andn.t d6,d4:0,d0:0</code>	d6	0000 0000	;d6:0 = /A./B
<code>nor.t d4,d0:3,d1:0</code>	d4	0000 0000	;d4:0 = /S4
<code>and.andn.t d4,d1:3,d1:2</code>	d4	0000 0000	;d4:0 = /S4.S1./S2
<code>or.t d8,d0:3,d1:0</code>	d8	0000 0001	;d8:0 = S4
<code>and.andn.t d8,d1:2,d1:3</code>	d8	0000 0001	;d8:0 = S4.S2./S1
<code>or.t d7,d4:0,d8:0</code>	d7	0000 0001	;d7:0 = /S4.S1./S2 + S4.S2./S1
<code>sh.and.t d3,d6:0,d7:0</code>	d3	0000 0006	;d3:0 = /A./B.(/S4.S1./S2 + S4.S2./S1)
<code>nor.t d4,d0:3,d1:3</code>	d4	0000 0001	;d4:0 = /S1
<code>andn.t d5,d4:0,d1:2</code>	d5	0000 0000	;d5:0 = /S1./S2
<code>and.t d4,d5:0,d0:1</code>	d4	0000 0000	;d4:0 = /S1./S2.A
<code>andn.t d6,d1:0,d5:0</code>	d6	0000 0001	;d6:0 = S4./(/S1./S2)
<code>sh.or.t d3,d6:0,d4:0</code>	d3	0000 000d	;d3:0 = /S1./S2.A + /(S1./S2).S4

7.7.3 Bit Testing

It is a loading bit followed by a test, depending on what you want to do with the value of the bit. In the following example, the value of the bit determines the value of register d1.

Example:

d0	0000 FFFF
d8	0000 0280

After Instruction

```
ld.w    d9, lx9
extr.u   d0, d9, #3, #1
sel      d1, d0, d8, d9
```

d9	0000 010B
d0	0000 0001
d1	0000 0280

8 Pointer Arithmetic and Addressing Modes

The TriCore architecture has two very different execution units, each corresponding to a separate pipeline:

- IP (Integer Processing)
- LS (Load/Store Unit)

This chapter deals with operations taking place within the LS unit.

The two main functions of the LS unit are to:

- Move data from the data register file to memory and vice-versa using different addressing modes.
- Generate the memory effective address and update it for the next cycle.

This chapter describes the different addressing modes, followed by the load operations and the store operations on data registers. All the possible operations on pointers (address registers) are presented in the following order:

- load
- store
- add
- subtract
- compare
- move.

Finally, the cases where address registers do not contain pointers but a general value, is explained.

Chapter contents:

- **Addressing Modes**
- **Data Load and Store**
- **Bit Load and Store**
- **Address Register Load and Store**
- **Address Arithmetic and Initialization**
- **Address Register as a GP Register**
- **Application**

8.1 Addressing Modes

Different addressing modes are described, including:

- **Absolute Addressing** and **Extended Absolute Addressing**
- **Base + Offset Addressing**
- **Pre-Increment / Decrement Addressing**
- **Post-Increment/Decrement Addressing**
- **Circular Addressing**
- **Bit-Reverse Addressing**
- **Indexed Addressing (ADDSC.A)**
- **PC-Relative Addressing**
- **Bit Addressing**

Note: In TriCore, all addresses are byte addresses.

8.1.1 Absolute Addressing

Due to its simplicity, absolute addressing mode is (initially) the most employed memory mode.

Example:

d0	0000 0000
----	-----------

After Instruction

```
ld.w d0, 0xD0000000
```

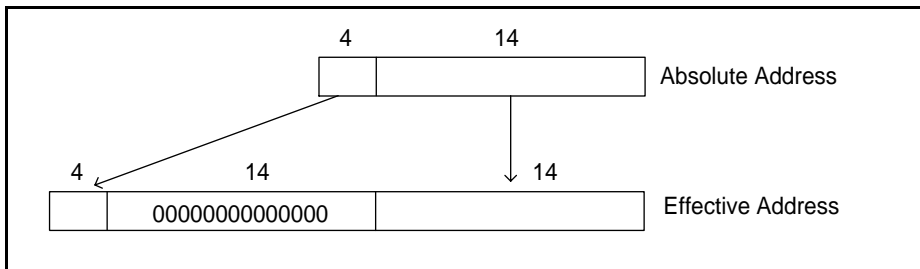
d0	1234 5678
----	-----------

; internal scratch
will be truncated to 18bit

```
ld.w d0, (0xD0000000+0x20)
```

d0	0007 0006
----	-----------

The instruction specifies an 18-bit constant as the memory address. As shown below, the full 32-bit address results from moving the 4 most-significant bits of the 18-bit constant to the 4 most-significant bits of the 32-bit register. The other bits are zero-filled.



The upper 4 bits of the 18-bit immediate operand are not a direct part of the address but are used to select one of the sixteen 256-Mbyte segments. Once a 256-Mbyte segment has been selected, the lower 14 bits of the immediate operand are then used to address *the first 16 Kbytes within the selected segment*. The reason for this special translation is to allow absolute addressing to be used in the first 16 Kbytes of each address segment, a necessary feature for referencing I/O peripheral registers and static data.

Note: Writing an 18-bit address in Hexadecimal - With Tasking assembler it should be written as a 32-bit word with zero between the 28th and the 15th bit.

8.1.2 Extended Absolute Addressing

Extended absolute addressing is synthesized using two instructions:

- MOVH.A
- LEA

The LEA instruction loads a 32-bit address into an address register. After execution of the MOVH.A instruction, a base + 16-bit offset is used to address data in order to establish a base register.

Example:

After Instruction			
<code>movh.a a2, #((word_address) + 0x8000 >> 16)</code>	<table border="1"> <tr> <td>a2</td><td>9876 0000</td></tr> </table>	a2	9876 0000
a2	9876 0000		
<code>lea a2, [a2] (((word_address) + 0x8000) & 0xffff) - 0x8000)</code>	<table border="1"> <tr> <td>a2</td><td>9876 1234</td></tr> </table>	a2	9876 1234
a2	9876 1234		

Note: ((const + 0x8000) >> 16 is used for rounding)

Note: The same example is used later but with constants. Here it is loaded from the memory, and the address is not known. With constants the address register is initialized at a known address. In the example above the address is not known as it is loaded from memory.

8.1.3 Base + Offset Addressing

Base + offset addressing is used for referencing record elements, local variables (using the stack pointer SP as the base) and static data (using an address register pointing to the static data area).

- Effective address = address register + sign extended offset

Example:

	d0	0000 0000	
	After Instruction		
lea a2, 0xD0000010	a2	D000 0010	
ld.w d0, [a2]+0x48	d0	0000 0004	word at location 0xD0000058
ld.w d0, [a2]-0x10	d0	12345678	word at location 0xD0000000

The offset value is always given in bytes.

The offset has a maximum value of [-512..+512] bytes for all instructions except load word (ld.w) and load address (ld.a), which provide a range of [-32768..+32767] bytes.

8.1.4 Pre-Increment / Decrement Addressing

Pre-incrementing and pre-decrementing addressing are used for linear array and stack access, to respectively push data onto an upward or downward growing stack.

- Effective address = address register + sign-extended 10-bit increment
- Updated address register = address register + sign extended 10-bit increment

Note: Since the increment is a signed value, the decrement is an increment with negative value.

Example:

	d0	0000 0000	
	After Instruction		
lea a2, 0xD0000010	a2	D000 0010	
ld.w d0, [+a2] 6	d0	0002 0001	word at location 0xD0000016
ld.h d0, [+a2] 4	d0	0000 0003	halfword at location 0xD000001a

The increment value is always given in bytes. In the example above a load word is incremented by 6 bytes and the load halfword by 4 bytes.

The increment has a maximum value of [-512..+512] bytes.

8.1.5 Post-Increment/Decrement Addressing

Post-incrementing and post-decrementing addressing provides forward and backward sequential access of arrays. This mode can also be used to pop down (post-increment) or pop up (post-decrement) a growing stack.

- Effective address = address register
- Updated address register = address register + sign-extended 10-bit increment

Note: A decrement is a negative increment (increment is signed).

Example:

	d0	0000 0000	
	After Instruction		
lea a2, 0xD0000010	a2	D000 0010	
ld.w d0, [a2+] 4	d0	0403 0201	;a2 = 0xD0000014
ld.d e4, [a2+] 8	d4 d5	0001 0000 0003 0002	;a2 = 0xD000001c
ld.b d10, [a2-] 1	d10	0000 0004	;a2 = 0xD000001b
ld.b d1, [a2+] +0x43	d1	0000 0000	;a2 = 0xD000005e

The increment value is always given in *bytes*. In the examples above, a load double word is incremented by 8 bytes and the load byte is decremented by one byte. The last example shows that increment is absolutely independent of the loaded data size.

The increment has a maximum size of [-512..+512] bytes.

8.1.6 Circular Addressing

Circular addressing is mainly used for accessing data values in circular buffers while performing filter calculations or convolution.

TriCore combines two consecutive address registers to define a circular buffer. The even address register contains the base (start) address, the upper 16 bits of the odd address register contains the length of the circular buffer, and the lower 16 bits of the odd address register contains an index into the circular buffer. When the circular buffer is accessed, only the index is updated; the base and length are unchanged.

odd	Length L	Index I
even	Base B	

Circular addressing mode

Update algorithm:

if $(I + \text{offset} < 0)$ $I = I + \text{offset} + L;$

else if $(I + \text{offset} \geq L)$ $I = I + \text{offset} - L;$

else $I = I + \text{offset};$

Consider a circular buffer consisting of 24, 16-bit values (so the *length* is 48 bytes).

If the current *index* is 46 and the *offset* is 2 (two bytes per value), the new value of the *index* wraps around to 0 ($I + \text{offset} \geq L$ in this case).

If the *index* is 46 and the *offset* is 4 (two entries per step), the new value of the *index* would be 2 ($I + \text{offset} - L = (46 + 4) - 48$).

If the current *index* is 4 and the *offset* is -8, then the next *index* would be 44 ($I + \text{offset} + L = 4 - 8 + 48$).

- Since circular buffers are implemented using address register pairs, up to eight circular buffers can be active simultaneously.
- The circular buffer can be any size up to 65 535 bytes. Size is always in *bytes*.
- The offset is a signed 10-bit offset.
- A circular buffer must start at a base address that is aligned to a *multiple of eight bytes*.
- The length of the buffer must be a *multiple* of the size of the data accesses made into the buffer. For example, if a 32-bit data access is made into the circular buffer, the length of the buffer must be a multiple of four bytes.

Example:

	a3	0000 0000	
After Instruction			
movh.a a3,#4	a3	0004 0000	;length = 4, index = 0
lea a2,base_address	a2	D000 0010	;base address = base_address
ld.h d0,[a2/a3+c]2	d0	0000 0201	a3 0004 0002
ld.h d0,[a2/a3+c]2	d0	0000 0403	a3 0004 0000
ld.h d0,[a2/a3+c]2	d0	0000 0201	a3 0004 0002

8.1.7 Bit-Reverse Addressing

Bit-reverse addressing is used to access arrays used in FFT algorithms. A common implementation of FFT ends with results stored in bit-reversed order. As shown in the Figure below, a common FFT implementation ends with results stored in bit-reversed order. The Bit-Reverse addressing mode corrects this problem. By using Bit-Reverse, the data are stored in the correct linear order.

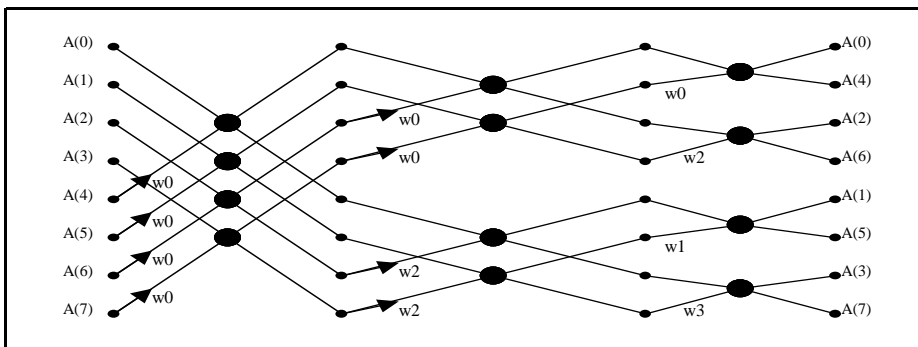


Figure 5 Bit-Reverse Addressing

Bit-reverse addressing is implemented using two consecutive address registers in a similar fashion as for circular buffers. The even register holds the base address; the beginning of a buffer within which bit reverse addressing is employed. The odd register holds a size and an index in the upper and lower 16-bit of the register.

odd	Modifier M	Index I
even	Base B	

Register pair for bit-reverse addressing

The size of the buffer is always in *halfwords* and it must be a *power of 2*.

There is *no offset*. This means the format of the value in memory must be the same as the size that is loaded or stored.

Example:

Initialization of the bit-reverse pointer:

	a3	0000 0000	
	After Instruction		
movh.a a3,#8	a3	0008 0000	;modifier = 8, index = 0
lea a2,val	a2	D000 0014	;base address = val
ld.h d0,[a2/a3+r]2	d0	0000 0000	a3 0008 0008
ld.h d0,[a2/a3+r]2	d0	0000 0004	a3 0008 0004
ld.h d0,[a2/a3+r]2	d0	0000 0002	a3 0008 000c
ld.h d0,[a2/a3+r]2	d0	0000 0006	a3 0008 0002
ld.h d0,[a2/a3+r]2	d0	0000 0001	a3 0008 000a
ld.h d0,[a2/a3+r]2	d0	0000 0005	a3 0008 0006
ld.h d0,[a2/a3+r]2	d0	0000 0003	a3 0008 000e
ld.h d0,[a2/a3+r]2	d0	0000 0007	a3 0008 0001

8.1.8 Indexed Addressing (ADDSC.A)

Indexed addressing is used for accessing data values in two-dimensional arrays and structures. There is no indexed addressing on TriCore and the two examples of addressing modes which follow, do not exist. **However**, there are several ways to create indexed addressing on TriCore.

Incorrect Examples:

```
ld.w    d0, [a2] + [a6]    ;non existent
ld.h    d0, [a2+a6] + [a8] ;non existent
```

One way to create indexed addressing on TriCore, is to use the ADDSC.A instruction. In a scaled index addition and subtraction, the contents of the data are left-shifted by zero, one, two or three bits (for addressing indexed arrays of bytes, halfwords, words or double-words), before adding the contents to an address register.

Example:

a2	D000 0038
a3	0000 0004
d0	0000 0004

;a3 and d0 represent the offset

After Instruction

```
add.a   a4, a2, a3
```

```
ld.w    d1, [a4]
```

```
addsc.a a4, a2, d0, #0
```

```
ld.w    d1, [a4]
```

a4	D000 003C
d1	0000 000A
a4	D000 003C
d1	0000 000A

16-bit opcode syntax:

addsc.a dest/src1(address),src2(address),d15,n

ADD.A and SUB.A instructions can be used.

Circular addressing can also be used. Look at this two-dimension array:

A[j][i]

		x	

To access A[j][i], only the size of the value to define the offset and increment the index, is required.

Going through the row and accessing x, which is a halfword, is executed as:

Example:

;a2=j, a3=0x00080004 (size=8, index(i)=4), offset=2 (2 bytes = halfword)

lea a2,A2

mov.ha a3,#8

add.a a3,#4

ld.h d0,[a2/a3+3+c]

a2	D000 0038
a3	0008 0000
a3	0008 0004
d0	0000 000A

To change the row, the base of the circular buffer (a2 in this example) must also be changed. Note that the *offset* is *fixed*.

8.1.9 PC-Relative Addressing

TriCore architecture does not support the direct PC-relative addressing of data. The main reason for this is that the separate on-chip instruction and data memories make data access to the program memory expensive. It typically adds two cycles of added access time.

When PC-relative addressing of data is required, the address of a nearby code label is placed into an address register and used as a base register in base + 16-bit offset mode to access the data. Once the base register is loaded, it can be used to address other nearby PC-relative data items.

A code address can be loaded into an address register in various ways. If the code is statically linked (as it almost always is for embedded systems), then the absolute address of the code label is known, and can be loaded using the LEA instruction, or with a sequence to load an extended absolute address. The absolute address of the PC-relative data is also known, and there is no need to synthesize PC-relative addressing.

For code that is dynamically loaded, or assembled into a binary image from position independent pieces without the benefit of the relocating linker, the appropriate way to load a code address for use in PC-relative data addressing is to use the JI instruction (Jump & Link). A jump and link to the next instruction is executed, placing the address of that instruction into the return address register (A11). Before doing so, copy the actual return address of the current function to another register.

8.1.10 Bit Addressing

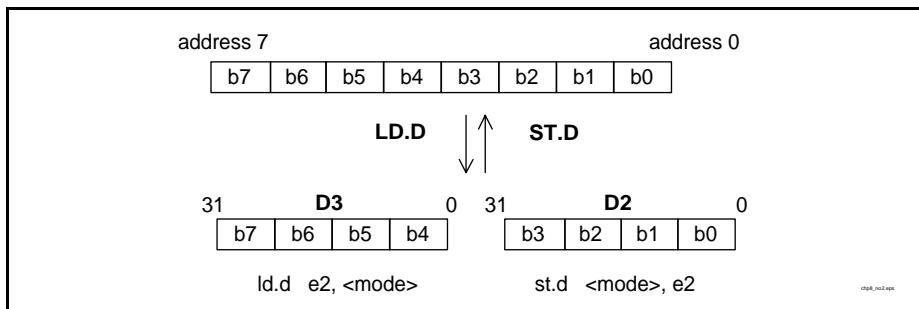
For support of addressing of indexed bit arrays, the ADDSC.AT instruction scales the index value by one eighth (shifts right three bits) and adds it to the address register. The two low-order bits of the resulting byte address are cleared to give the address of the word containing the indexed bit. To extract the bit, the word containing it is loaded, and the bit index is used in an EXTR instruction. A bit field, beginning at the indexed bit position, can also be extracted. To store a bit or a bit field at an indexed bit position, ADDSC.AT is used in conjunction with the LDMST instruction (See the [Bit Load and Store](#) section).

8.2 Data Load and Store

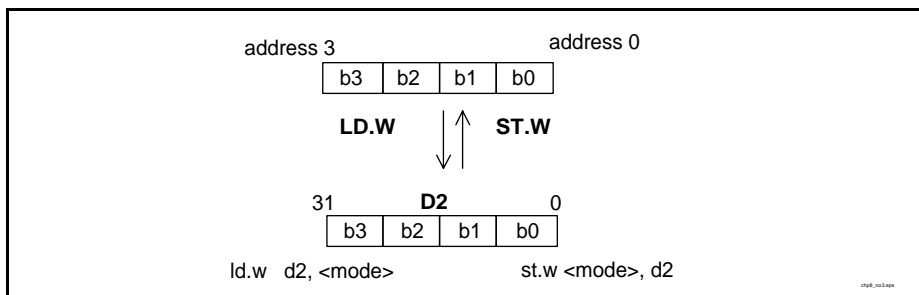
The following table summarizes the characteristics of the 12 data load and store instructions. The alignment rules apply to the memory address. TriCore has different alignment rules depending on internal memory (scratchpad memory) or external memory.

Instruction name	Data Size	Extension to 32-bit	Alignment Rules		16-bit opcode available ?
			Internal	External	
ld.d	double	no need	halfword	word	no
ld.w	word	no need			yes
ld.h	halfword	sign		halfword	yes
ld.hu	halfword	zero			no
ld.q	halfword	zero in lower half			no
ld.b	byte	sign	byte	byte	no
ld.bu	byte	zero			yes
st.d	double	no need	halfword	word	no
st.w	word	no need			yes
st.h	halfword	no need		halfword	yes
st.q	halfword	no need			no
st.b	byte	no need	byte	byte	yes

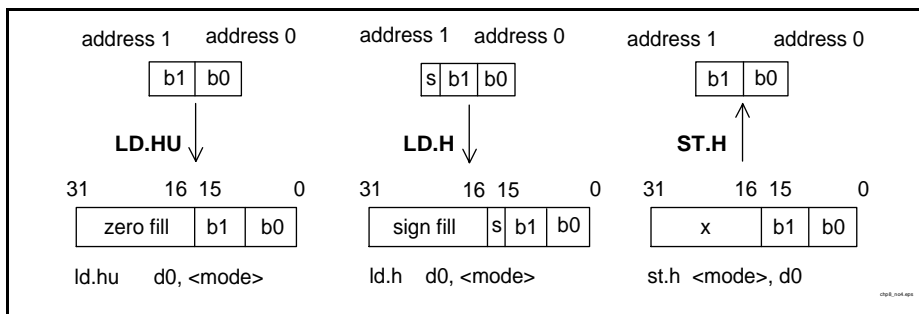
8.2.1 Load/Store Double Word (LD.D, ST.D)



8.2.2 Load/Store Word (LD.W, ST.W)

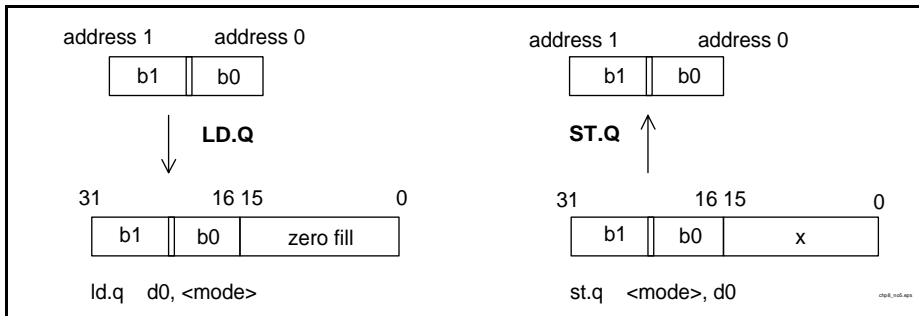


8.2.3 Load/Store Halfword (LD.H, LD.HU, ST.H)



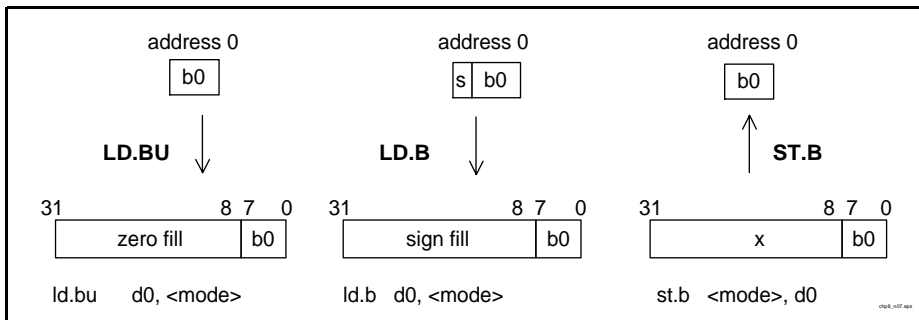
8.2.4 Load/Store Halfword (LD.Q, ST.Q)

RISC machines classically align their data on the right side, whereas DSPs use a left-alignment. Left alignment is generally associated with fractional (or Q) format, hence the name.

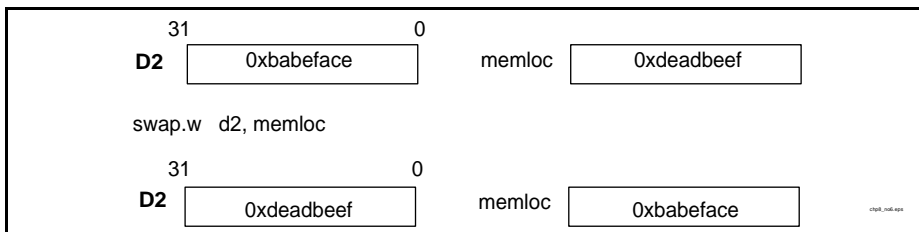


Note: LD.Q fills the lower part with zeros. To leave the lower part untouched, two instructions (LD followed by INSERT) and a temporary register must be used.

8.2.5 Load/Store Byte (LD.B, LD.BU, ST.B)



8.2.6 Swap (SWAP.W)



8.3 Bit Load and Store

8.3.1 Storing a bit (ST.T)

The ST.T instruction directly clears or sets a bit in memory:

`st.t <memory mode>, position, action (clear or set)`

Example:

IX0	1234 5678
IXI	1122 3344

After Instruction

`st.t lX0,#3,#0`

IX0	1234 567 0
IXI	1122 33 64

;clear the 4th bit of IX0

`st.t lX0,#5,#1`

;set the 6th bit of IX0

The combination of the IMASK and LDMST instructions can also be used to store a bit. The difference between them is the manner in which memory is accessed. For ST.T, the first 16 Kbytes of each of the 16 memory segments can be reached since they are used only with the absolute addressing mode.

The IMASK / LDMST combination offers all the addressing modes due to the LDMST instruction.

8.3.2 Storing a Bit Field (IMASK, LDMST)

IMASK and LDMST can also be used for storing a bit field into a word in memory. This is useful when dealing with (memory mapped) peripherals. The memory word location is specified using any addressing modes.

The IMASK instruction is very similar to the INSERT instruction, but instead generates a data register pair, which contains a mask and a value. The mask is used by the LDMST instruction to indicate which portion of the word to modify.

`imask dest (extended register), value, position, width`

Create a mask containing *<width>* bits, starting at *<position>* and put the result in e0 (upper). Then left-shift *<value>* by *<position>* and put the result in e0 (lower).

The IMASK instruction offers four possibilities of syntax:

d2	0000 0004
d3	0000 0005

After Instruction

`imask e0, d2, d3, #3`

d0	00000080
d1	000000E0

`imask e0, d2, #5, #3`

d0	00000080
d1	000000E0

`imask e0, #4, d3, #3`

d0	00000080
d1	000000E0

`imask e0, #4, #5, #3`

d0	00000080
d1	000000E0

LDMST

`ldmst <mode>, source (extended register with new value and mask)`

Load a value from memory, modify the bits at the position *<mask>* with the bits of *<new value>* and store the modified value.

Example:

IX9	0001 054F
-----	-----------

After Instruction

`imask e8, #5, #7, #3`

d8	0000 0280
d9	0000 0380

Value at position 7
Mask at position 7

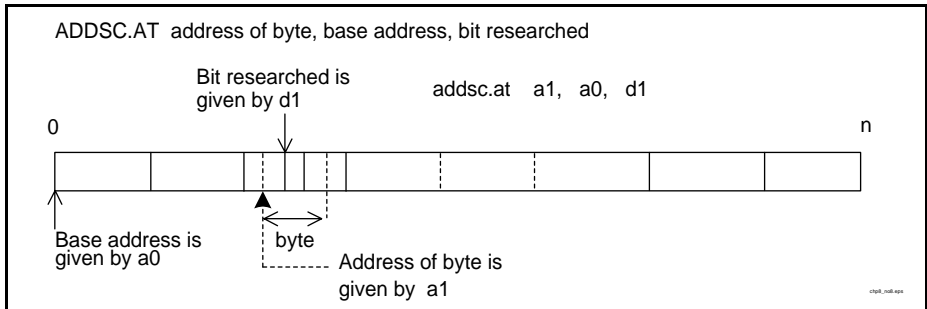
`ldmst lx9, e8`

IX9	0001 06CF
-----	-----------

Note: For all the examples the bit count begins at 0 (implying the first bit is bit 0).

8.3.3 Storing a Single Bit (ADDSC.AT)

Storing a single bit with a variable bit offset from a word aligned byte pointer starts out by figuring out the word address with a special scaled offset instruction that shifts the bit offset to the right by three position, so producing a byte offset. It then proceeds to add it to the byte pointer above, and finally zeros out the lower bits, aligning the access on a word boundary. This allows an IMASK and LDMST to store the bit into the correct position in the word. The ADDSC.AT instruction returns the address of the byte containing the required bit.



Example:

Before Instruction

After Instruction

```
lea a9,lsbitex
```

```
mov d8, #0
```

```
addsc.at  
a8,a9,d8
```

```
imask  
e10,#1,d8,#1
```

```
ldmst [a8],e10
```

a9	d000 0d48
d8	0000 0000
a8	0000 0080
d10	0000 0001
d11	0000 0001
a8	0000 010B

Value
Mask

[a8]	0000 010A
------	-----------

8.3.4 Loading a Single Bit

The approaches for loading individual bits depend on whether the bit within the word (or byte) is given statically or dynamically.

Static Position

Loading a single bit with a fixed bit offset from a byte pointer can be accomplished with an ordinary load instruction, but the bit does not generally end up in position 0 of a data register as required. However, since it is possible to insert, logically operate on, or jump on any bit in a register (see [Chapter 9](#)), this is not a drawback.

Dynamic Position

Loading a single bit with a variable bit offset from a word aligned byte pointer can be performed in the same way as for store a bit. Again the special scaled offset instruction shifts the bit offset to the right by three positions, producing a byte offset, then adds it to the byte pointer above, and finally zeros out the lower bits, aligning the access on a word boundary. This allows a word load to access the word that contains the bit, which can then be extracted with an extract instruction, which only uses the lower five bits of the bit pointer, i.e. the bits that were shifted out.

Example:

a9	D000 0048
d8	0000 0003

After Instruction

```
addsc.at    a8, a9, d8
ld.w        d9, [a8]
extr.u      d0, d9, d8, #1
```

a8	d000 0048
d9	0000 010B
d0	0000 0001

8.4 Address Register Load and Store

In the TriCore programming model there is a significant difference between data and address registers (pointers). All pointers are 32-bit wide and are word aligned. There is no need for halfword or byte access.

Existing instructions and their differences are shown in the following table:

Name	Pointer Size	Alignment rules	16-bit opcode available ?
ld.da	double	word	no
ld.a	word	word	yes
st.da	double	word	no
st.a	word	word	yes

8.4.1 Load/Store Address Register (LD.A, ST.A)

Storing/restoring address registers can be accomplished with LD.A and ST.A. Any addressing mode can be used.

```
ld.a    a5, context_module
ld.a    a5, [a6]+offset
st.a    [a6]+offset-1, a5
ld.a    a5, [a14+]2
st.a    [a6]-0xa, a5
```

The case below is also valid, as long as *a5* points to a valid memory region before *and after* the load:

```
ld.a    a5, [a5]
st.a    [a5], a5
```

8.4.2 Load/Store Double Address Registers (LD.DA, ST.DA)

The 64-bit bus bandwidth can also be used to perform a double load of two registers in one cycle.

```
ld.da    Aa/Aa+1, <mode>
st.da    <mode>, Aa/Aa+1
```

This is especially useful when reloading a circular buffer pointer, since it requires initializing two registers.

Example:

```
ld.da    a4/a5, circ_buffer_save    ;retrieve circular pointer
;... init fir algorithm would be here
ld.q     d0, [a4/a5+c]+4            ;ld first data
;... fir algorithm would be here
st.q     [a13], d0                  ;save 16bit result
st.da    circ_buff_save, a4/a5      ;save circular pointer
```

For more than 2 pointers TriCore provides a powerful mechanism called context switching. 8 data and address registers can be loaded in one instruction (See [Program Control and Context Switch - Chapter 9](#)).

8.5 Address Arithmetic and Initialization

Note: All address instructions are part of the Load/Store unit, which has its own ALU. So all of them can be dual issued with IP instructions.

8.5.1 LEA (Initializing to a Segmented 18-bit Address)

Initialization of base pointers requires *loading a constant into an address register*. When the base pointer is in the first 16 Kbytes (which corresponds to 0x3FFF) of each segment, this is done using the LEA (Load Effective Address) instruction, via absolute addressing mode.

LEA also uses the base + offset addressing mode:

Example:

After Instruction

```
lea    a2, 0x50
```

```
lea    a3, [a2]+4
```

a2	0000 0050
a3	0000 0054

The LEA instruction can perform a move of a 16-bit constant into an address register. The difference between the LEA and the LD instruction is that the LD instruction accesses the value directly, whereas the LEA instruction accesses the address of this value.

Example:

After Instruction		
ld.a a2,0x50	a2	0000 0050
lea a3,[a2]+4	a3	0000 0054

The LEA instruction can perform a move of a 16-bit constant into an address register. The difference between LEA and LD instruction is that the LD instruction accesses the value directly, while the LEA instruction accesses the address of this value.

Example:

@lx2 = D000 0064	lx2	1122 3344
After Instruction		
ld.a a2, lx2	a2	1122 3344
lea a3,lx2	a3	D000 0064

8.5.2 Initializing to a 32-bit Address

The most common pointer operation is initialization (with constants). In this respect TriCore encounters the basic law of opcode physics, that a 32-bit opcode cannot contain a 32-bit constant address.

Loading a 32-bit address is synthesized by a formula, which can be made simpler with the use of a macro:

```
CONSTA .macro reg,addr
    movh.a reg,#(((addr) + 0x8000 >> 16)
    lea reg,[reg](((( addr) + 0x8000) & 0xffff) - 0x8000)
.endm
```

Instruction	Becomes	
CONST.A a2,0xd4007fff	movh.a a2,#0xd400	lea a2,[a2]0x7fff
CONST.A a2,0xd4008000	movh.a a2,#0xd401	lea a2,[a2]-0x8000
CONST.A a2,0xd4008001	movh.a a2,#0xd401	lea a2,[a2]-0x7fff

In case a macro-assembler is not available, this expression can be simplified in many situations. If the address for instance, is known at compile time and the lower half of the address is smaller than 0x7FFF, loading a 32-bit address can be written as:

After Instruction

<code>movh.a a2, #0xC000</code>	a2	C000 0000
<code>lea a2, [a2] 0x4000</code>	a2	C000 4000

Note: This example is NOT universal !

The same method can be applied to the lower half of the address if it is greater than 0x7FFF, by loading the upper bound and using a negative offset.

After Instruction

<code>movh.a a2, #0xD000</code>	a2	D000 0000
<code>lea a2, [a2] -0x4000</code>	a2	C000 C000

8.5.3 Addition (ADD.A, ADDIH.A)

Adding two address registers is implemented with the ADD.A instruction:

```
add.a a2, a3, a4
```

To add a 4-bit constant to an address register, use the 16-bit instruction:

```
add.a a2, #4
```

ADDIH.A will add a 16-bit constant to the upper part:

```
addih.a a2, a3, #0x1234
```

Example:

a3	0000 0010
a4	0000 0A00

After Instruction

```
add.a a2, a3, a4
```

```
add.a a3, #4
```

```
addih.a a2, a4, #0x1234
```

a2	0000 0A10
a3	0000 0014
a2	1234 0A00

To add a 16-bit constant to the lower part, use a dummy load with auto-increment

```
ld.w d0, [a4+] #0x1234
```

Note: In a typical situation the pointer of interest would have been used previously and correctly auto-incremented.

16-bit opcode Syntax:

```
add.a dest/src1(address), src2(address)
```

```
add.a dest/src1(address), k4
```


8.5.4 Subtraction (SUB.A)

Subtracting two address registers is also possible:

```
sub.a a2, a3, a4
```

The same instruction is used to subtract an 8-bit constant. In the example, the stack pointer a10 is auto-decremented by 20 bytes.

```
sub.a a10, #0x20
```

Example:

a3	0000 0A70
a10	0000 0070

After Instruction

```
sub.a a2, a3, a10
```

```
sub.a a10, #0x20
```

a2	0000 0A00
a10	0000 0050

Note: *RESTRICTION* - The subtraction with 8-bit constant is always used with register a10 and it is a 16-bit opcode.

8.5.5 Comparison (EQ.A, EQZ.A, GE.A, LT.A, NE.A, NEZ.A)

The result of a comparison is put in the least significant bit of the destination data register and clears the remaining register bits to zeros.

The EQZ.A and NEZ.A instructions are used to test for null pointers, a frequent operation when dealing with linked lists and complex data structures.

Example:

a2	0000 0010
a3	0000 0010

After Instruction

eq.a	d0, a2, a3	d0	0000 0001
eqz.a	d1, a2	d1	0000 0000
ge.a	d2, a2, a3	d2	0000 0001
lt.a	d3, a2, a3	d3	0000 0000
ne.a	d4, a2, a3	d4	0000 0000
nez.a	d5, a2	d5	0000 0001

Note: Comparison conditions not explicitly provided in the instruction set can be obtained by swapping the two operands registers.

'Missing' comparison operation		TriCore equivalent comparison operation	
LT.A	d0, a2 , a3	GE.A	d0, a3, a2
GT.A	d0, a2 , a3	LT.A	d0, a3, a2

8.5.6 Move (MOV.AA)

The MOV.AA instruction moves an address register into another address register:

Example:

a2	1234 0000
----	-----------

After Instruction

`mov.aa a3,a2`

a3	1234 0000
----	-----------

16-bit opcode Syntax:

```
mov.aa    dest (address),src (address)
```

8.5.7 Mixed Registers Moves (MOV.D, MOV.A)

There are instances when the pointer manipulation cannot be performed in the LS unit. A typical case is multiplication by N, or division by N, or division by a power of 2. This is achieved by moving an address register into a data register (MOV.D) and back (MOV.A):

Example:

```
mov.d      d0,a2          ; move address value to data value to be changed
sh         d0,d0,#2       ; modify value
mov.a      a2,d0          ; move back to address register
```

16-bit opcode Syntax:

```
mov.a      dest (address),src (data)
```

8.6 Address Register as a GP Register

Conditions exist where address registers can contain values which are not pointers: length of circular buffer, and arithmetic values. The Load/Store(LS) unit is a small, separate Arithmetic and Logic Unit (ALU). This is useful where there is no memory activity. The LS could be used to perform calculations.

8.6.1 Loading to a 16-bit Value (MOVH.A)

The MOVH.A instruction is used to move a 16-bit constant into the upper part of an address register.

Example:

```
movh.a  a3, #(Circ_buffer_length/2); divide by 2 because data is 16-bit
                                           ; and all memory is in bytes
```

To move a 16-bit constant into the lower part of an address register, write:

```
lea      a2, 0x1234
```

Note: The constant can not be more than 0x3FFF. There is no need for a hash symbol (#) in front of the constant with the Tasking syntax, as it is viewed as an effective address.

Examples:

After Instruction

```
movh.a  a2, #0x0078
```

```
lea      a3, 0x1234
```

a2	0078 0000
a3	0000 1234

8.7 Application

This example illustrates the use of some address arithmetic instructions.

8.7.1 Find Index of Maximum Value in Array

All the values of the array are checked to find the index. If the value is greater than the reference, take its index.

Example:

```
mov      d1,#0                ;initialize the data reference
lea      a8,values            ;initialize the pointer
lea      a6,5                 ;number of values
lea      a5,1
sub.a    a6,a6,a5             ;adjust loop counter for post-decr on loop
ld.h     d2,[a8]              ;load first value
iloop:   max      d1,d1,d2     ;take the max of two values
         jlt      d2,d1,lab    ;check if we need the address of the max
         mov.aa   a4,a8        ;take the address of the new max
lab:     ld.w     d2,[+a8]4     ;load next value
loop     a6,iloop
```

Note: If the values are 16-bit, this function can be implemented with the IXMAX instruction.

9 Program Control and Context Switch

Program control is usually under the control of the programmer, whereas context switching only occurs during exception processing (interrupts and traps). With TriCore, both program control and context switching are under the programmer's control in the case of subroutine calls and routine.

This chapter describes unconditional and conditional branches, their displacements and other features. Loops and calls are then explained, followed by an explanation of context switch instructions.

Chapter contents:

- **Unconditional Branches**
- **Conditional Branches**
- **Loops**
- **Subroutine Calls**
- **Context Switch**
- **Applications**
- **Summary Table: Displacement Field Width**

9.1 Unconditional Branches

Name	Definition	Displacement field width	
		32-bit opcode	16-bit opcode
JA	Jump absolute	24	
J	Jump	24	8
JI	Jump indirect		x

The displacement is signed in the 32-bit opcode and unsigned in the 16-bit opcode. A label can replace the constant for the jump instructions.

9.1.1 Jump Absolute (JA)

The JA instruction uses 24 bits of displacement as an absolute address.

Example:

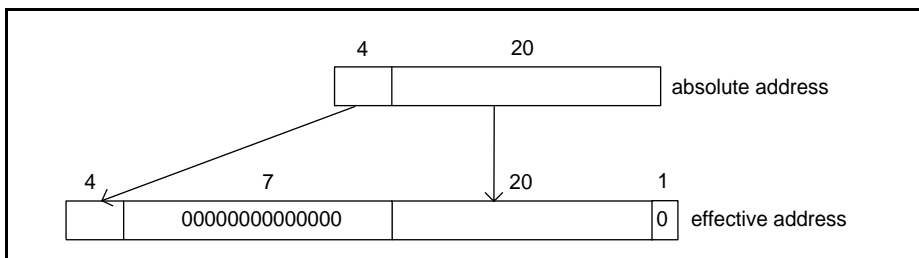
PC	d400 0000
----	-----------

After Instruction

`ja external_memory`

PC	a000 1234
----	-----------

This is the same principle as absolute addressing, using 24-bit to build the address. Last bit is always 0, since opcodes are 16 or 32-bit.



9.1.2 Jump PC Relative (J)

The instruction J uses a relative 24-bit signed offset.

Example:

PC	d400 0004
----	-----------

After Instruction

j32 two

PC	d400 0008
----	-----------

In this example the assembler converts this to 4-byte, to add to the Program Counter.

16-bit Opcode Example:

PC	d400 0008
----	-----------

After Instruction

j16 three

PC	d400 000a
----	-----------

In this example the assembler converts this to 2-byte to add to the Program Counter.

9.1.3 Jump Indirect (JI)

The JI instruction uses the address contained in an address register.

Example:

```
movh.a    a2,#0xa000
lea       a2,[a2]0x016c
ji        a2
four:     ;(address 0xa000016c)
```

Implementation Example:

```
if (a > 0) (x = x * 5; y += x; )
if (b != 0) (y = y * 5; x += y; )
z = x * y;
```

After Instruction

mov.u d2,#32

d2	0000 0020	; a
d3	1122 3344	; b
d4	0000 0010	; x
d5	9988 7766	; y

ld.w d3,lx0

mov.u d4,#16

ld.w d5,lx1

jgtz d2,cond

first: jnz d3,condi

j end0

cond: mul d4,d4,#5

add d5,d5,d4

j first

condi: mul d5,d5,#5

add d4,d4,d5

end: mul d6,d4,d5

d4	0000 0050	; x=x*5
d5	9988 77B6	; y=y+x

d5	FFAA 568E	; y=y*5
d4	FFAA 56DE	; x=x+y
d6	DB16 C324	; z= z*y

9.2 Conditional Branches

This section covers:

- **Jump If (eq, ne, lt, le, ge, gt) Zero**
- **Jump If (eq, ne, lt, lt.u, ge, ge.u) Small Constant**
- **Jump If (eq, ne, lt, lt.u, ge, ge.u) Any Value**
- **Jump On Bit (JZ.T, JNZ.T)**
- **Jump On Address (JZ.A, JNZ.A, JEQ.A, JNE.A)**

9.2.1 Jump If (eq, ne, lt, le, ge, gt) Zero

Name	Definition	Displacement field width (bits) 16-bit opcode
JZ	Jump if equal to zero	8 or 4 (see Notes 1, 2 & 3)
JNZ	Jump if not equal to zero	8 or 4 (see Notes 1, 2 & 3)
JLTZ	Jump if less than zero	4 (see Note 2)
JLEZ	Jump if less than or equal to zero	4 (see Note 2)
JGEZ	Jump if greater than or equal to zero	4 (see Note 2)
JGTZ	Jump if greater than zero	4 (see Note 2)

Note: 1) The displacement is 8-bit when d15 is used, 4-bit in all other cases.

Note: 2) If the displacement cannot fit in 4-bits (or 8-bits), the assembler automatically generates the 32-bit opcode version of the instructions; JEQ, JNE, JLT, JLT.U, JGE, JGE.U.

Note: 3) The assembler will refuse the JZ,JNZ because the 32-bit version is JEQ/JBE K (with K = 0)

```
jz      d2, LABEL
jnz     d2, LABEL
jltz    d2, LABEL
jlez    d2, LABEL
jgez    d2, LABEL
jgtz    d2, LABEL
```

Example:

```
if (a == 0) ( b = b + 5; a = 1; )
b +=a;
```

After Instruction

mov.u d15, #0

ld.w d14, lX1

jnz d15, LBX

add d14, #5

mov d15, #1

LBX: add d14, d14, d15

d15	0000 0000
d14	9988 7766

d14	9988 776B
d15	0000 0001
d14	9988 776C

9.2.2 Jump If (eq, ne, lt, lt.u, ge, ge.u) Small Constant

```

jeq    d0, k4, LABEL
jne    d0, k4, LABEL
jlt    d0, k4, LABEL
jlt.u  d0, k4, LABEL
jge    d0, k4, LABEL
jge.u  d0, k4, LABEL

```

The displacement is signed in the 32-bit opcode and unsigned in the 16-bit opcode.

Name	Definition	Displacement field width	
		32-bit opcode	16-bit opcode
JEQ	Jump if equal to	15	4
JNE	Jump if not equal to	15	4
JLT	Jump if less than	15	
JLT.U	Jump if less than (unsigned)	15	
JGE	Jump if greater than or equal to	15	
JGE.U	Jump if greater than or equal to (unsigned)	15	

Constant: The k4 constant is a 4-bit signed constant and can be between –8 and +7.

Example:

```
if (y >= -6) {y = y * 5;} y += 3;
```

d4	0000 0010
----	-----------

After Instruction

```
jlt    d4, #-6, end
```

```
mul    d4, d4, #5
```

```
end:   add d4, d4, #3
```

d4	0000 0050
d4	0000 0053

9.2.3 Jump If (eq, ne, lt, lt.u, ge, ge.u) Any Value

Name	Definition	Displacement field width	
		32-bit opcode	16-bit opcode
JEQ	Jump if equal to	15	4
JNE	Jump if not equal to	15	4
JLT	Jump if less than	15	
JLT.U	Jump if less than (unsigned)	15	
JGE	Jump if greater than or equal to	15	
JGE.U	Jump if greater than or equal to (unsigned)	15	

The displacement is signed in the 32-bit opcode and unsigned in the 16-bit opcode.

```
jeq    d0,d1,LABEL
jne    d0,d1,LABEL
jlt    d0,d1,LABEL
jlt.u  d0,d1,LABEL
jge    d0,d1,LABEL
jge.u  d0,d1,LABEL
```

Example:

```
if (y >= x)
{ y = y * 5; x = x * 6; z = x * y; }
z ++;
```

```
ld.w  d5, lx1
```

```
jlt   d4,d5 end
```

```
mul   d4,d4,#5
```

```
mul   d5,d5,#6
```

```
mul   d6,d5,d4
```

```
end:  add,d6,d6,#1
```

d4	0000 0010
d5	9988 7766

d4	0000 0050
d5	9932 CC64
d6	DFDF DF40
d6	DFDF DF41

9.2.4 Jump On Bit (JZ.T, JNZ.T)

Name	Definition	Displacement field width	
		32-bit opcode	16-bit opcode
JZ.T	Jump if bit equal to zero	15	4
JNZ.T	Jump if bit not equal to zero	15	4

The displacement is signed in the 32-bit opcode and unsigned in the 16-bit opcode.

```
jz.t    d0:3,LABEL ;if bit 3 of register d0=0, jump to LABEL
jnz.t   d0:3,LABEL
```

Example:

```
boolean flag1;
if (flag1 == 0) { y = y * 5; x += y; }
x ++;
```

d4	0000 0010
----	-----------

After Instruction

```
ld.w    d5,lx1
```

d5	9988 7766
----	-----------

```
jnz.t   d2:6,end
```

```
mul     d4,d4,#5
```

d4	0000 0050
----	-----------

```
add     d5,d5,d4
```

d5	9988 77B6
----	-----------

```
end:    add d5,d5,#1
```

d5	9988 77B7
----	-----------

9.2.5 Jump On Address (JZ.A, JNZ.A, JEQ.A, JNE.A)

Name	Definition	Displacement field width	
		32-bit opcode	16-bit opcode
JZ.A	Jump if address equal to zero	15	4
JNZ.A	Jump if address not equal to zero	15	4
JEQ.A	Jump if address equal to address	15	-
JNE.A	Jump if address not equal to address	15	-

The displacement is signed in the 32-bit opcode and unsigned in the 16-bit opcode.

Jump if address is equal to zero:

```
jz.a      a2, LABEL
jnz.a     a2, LABEL
```

Jump if address is equal to a small constant:

```
lea      a3, 4
jeq.a    a2, a3, LABEL
jne.a    a2, a3, LABEL
```

Jump if address is equal to any value:

```
jeq.a    a2, a3, LABEL
jne.a    a2, a3, LABEL
```

Example:

```
if (p != q) { x += y; }
x ++;
```

a2	0000 0050
a3	0000 0060
d4	0000 0010

After Instruction

```
ld.w     d5, lx1
```

d5	9988 7766
----	-----------

```
jeq.a    a2, a3, end
```

```
add      d5, d5, d4
```

d5	9988 7776
d5	9988 7777

```
end:     add d5, d5, #1
```

9.3 Loops

A programmer can implement the Loop construct in four different ways:

1. Using three instructions (this is offered by all CPUs and is not discussed further here):
 - decrement loop
 - test loop if equal to 0
 - jump if 0
2. Using JNEI, JNED (jump if not equal and increment(respectively decrement))
3. Using zero overhead loop (LOOP instruction)
4. Using the LOOPU instruction

9.3.1 Loop Using Jump Non Equal Decrement/Increment JNED/JNEI

This instruction compares two registers or a register and a constant; then increments (or decrements) the register automatically.

Name	Definition	Displacement field width 32-bit opcode (bits)
JNED	Jump if not equal to and decrement	15
JNEI	Jump if not equal to and increment	15

Constant: Can be used with a 4-bit signed constant and must be between –8 and +7.

Jump on a small constant:

```
jned    d0,k4,LABEL
jnei    d0,k4,LABEL
```

Jump on any value:

```
jned    d0,d1,LABEL
jnei    d0,d1,LABEL
```

With the JNEI instruction, if the input d0 is equal to 0xFFFFFFFF, the output will be 0x00000000.

With the JNED instruction, if the input d0 is equal to 0x00000000, the output will be 0xFFFFFFFF.

Example - Small Constant:

```
mov    d0,#8
```

```
loop1: jned d0,#-2,loop1
```

d0	0000 0008
d0	FFFF FFFE

Example - Any value:

```
mov d0, #450
```

```
mov d1, #460
```

```
loop2: jnei d0, d1, loop2
```

d0	0000 01c2
d1	0000 01cc
d0	0000 01cc

9.3.2 Zero Overhead Loop (LOOP)

The main difference between the LOOP instruction and the JNEI / JNED instruction, is that the LOOP is much faster. If M instructions are executed N times in the loop, it will take (N*M) cycles for LOOP, and (N*(M+1)) for JNEI / JNED (However, it could take longer, depending on the implementation).

Compare the following three implementations of the same algorithm, where d2, d3, a2 are the different counters.

Example 1:

```
mov.u d2, #16
L: add d1, d1, d0
  ld.w d0, [a3+]
  add d2, d2, #-1
  jne d2, #0, L
```

cycle count	$2 * (\text{counter} - 1) + 3 + 2 * \text{counter}$
-------------	---

Example 2:

```
mov d3, #15
LB: add d0, d1, d0
  ld.w d0, [a3+]
  jned d3, #0, LBL
```

cycle count	$2 * (\text{counter} - 1) + 3 + \text{counter}$
-------------	---

Example 3:

```
lea a2, 0xf
LBL: add d0, d1, d0
  ld.w d0, [a3+]
  loop a2, LBL
```

cycle count	$(\text{counter} - 1) + 2$
-------------	----------------------------

In the example 1, counter d2 is decremented in the algorithm itself with the second addition (which is actually a subtraction). The result is compared to zero to execute the jump or not.

In the example 2, counter d3 is automatically decremented and tested with the JNED instruction. There is therefore one less instruction in the loop.

In the third example, counter a2 is also automatically decremented and tested, but once the loop is started the LOOP instruction becomes invisible. The added 2 cycles are only an example. The actual number of cycles is implementation dependent.

9.3.3 Infinite Loop (LOOPU)

The LOOPU instruction is an infinite loop. It is also a zero overhead loop and is used to implement a (do – while) loop in C language.

Example:

```
do y = y*4; while (y < 100);
```

```
mov d0,#1
```

d0	0000 0001
d1	0000 0100

```
mov d1,#256
```

```
LABEL:
```

```
mul d0,d0,#4
```

d0	0000 0004
----	-----------

```
jlt d1,d0,outside
```

```
loopu LABEL
```

```
outside
```

Note: As there is no condition with the instruction, the condition is inside the loop with the jump instruction.

9.4 Subroutine Calls

- **Jump and Link (JL, JLA, JLI)**
- **Call, Return (CALL, CALLA, CALLI, RET)**

9.4.1 Jump and Link (JL, JLA, JLI)

A Jump and Link instruction stores the address of the next instruction in A11 and jumps to the indicated address. If a return is required, the JI instruction (ji a11) must be used.

Name	Definition	Displacement field width 32-bit opcode
JLA	Jump and link absolute	24
JL	Jump and link	24
JLI	Jump and link indirect	

```

jla    ABS_LABEL
jl     LABEL
jli    a2

```

The difference between the JL and JLA instructions is that the JL instruction uses a relative 24-bit signed offset, whereas the JLA instruction uses the 24 bits of displacement as an absolute address.

Jump and Link offers a very simple method of calling sub-routines, although it is also primitive in the sense that it does not save any registers and does not manage a stack. Therefore for function calls, it is better to use the TriCore hardware context switch mechanism. This is achieved with the CALL instructions.

9.4.2 Call, Return (CALL, CALLA, CALLI, RET)

Unlike a Jump and Link instruction, Call instructions save the caller's non-volatile registers (known as the upper context), in a dynamically allocated save area, known as the CSA - Context Save Area.

The function Return instruction RET, is used to return from a function that was invoked via a CALL instruction. RET restores the upper context of the calling function, and branches to the return address contained in register A11 (before the context restore operation).

Name	Definition	Displacement field width	
		32-bit opcode	16-bit opcode
CALL	Call	24	8
CALLA	Call absolute	24	
CALLI	Call indirect		
RET	Return from a call		

call new
calla new
calli a2
ret

Example - Jump & Link:

mov.u d13, #32

d13	0000 0020
d0	1122 3344
d14	0000 0010
d1	6677 8899
d2	7799 BBDD

ld.w d0, lX0

mov.u d14, #16

ld.w d1, lX1

add d2, d1, d0

j1 star

sub d12, d14, d2

j exit1

star: add d14, d13, d1

mov.u d2, #0x1e

d12	6677 889B
d14	6677 88B9
d2	0000 001E

```
ji a11
exit1;
```

Example - Call:

```
mov.u d13,#32
```

d13	0000 0020
d0	1122 3344
d14	0000 0010
d1	6677 8899
d2	7799 BBDD

```
ld.w d0,lx0
```

```
mov.u d14,#16
```

```
ld.w d1,lx2
```

```
add d2,d1,d0
```

```
call star
```

```
sub d12,d14,d2
```

d12	FFFF FFF2
-----	-----------

```
j exit2
```

```
star: add d14,d13,d1
```

d14	6677 88B9
d2	0000 001E

```
mov.u d2,#0x1e
```

```
ret
```

```
exit2:
```

In example 1, the register d12 is equal to 0x6677889B. The subtraction takes account of the modification of register d14. However, in example 2, register d14 was saved so that after the execution of the RET instruction, it restores the previous d14 value.

The lower context can be used to pass arguments to a called function. Similarly, since the arguments are not automatically restored as part of the RET sequence, they can be used to pass return values from called functions back to their callers.

See the example which follows.

Example:

Solve $y = (0.5 \cdot x - 1) / 2$ by calling subroutine so that it becomes:

$y = \text{Lshr}(\text{Ladd}(\text{Lmult}(0x4000, x), 0x8000), 1)$

```
movh      d5, #0x4000      ;load 0.5
ld.w      d4, lX6          ;load x
call      Lmult            ;0.5 * x will return
mov       d11, d2          ;keep the result
mov       d10, d2          ;keep the result
movh      d4, #0x8000      ;load -1
mov       d5, d11         ;load previous result
call      Ladd             ;0.5*x - 1 will return
mov       d4, d2           ;load previous result
mov.u     d5, #1           ;load 1
call      Lshr             ;(0.5*x - 1)/2 will return
```

At the end of the last call the upper context will be restored and registers d8 through d15 will have their previous value, so d11 and d10 are unchanged.

Registers d4 and d5 are used as parameters and the result is always in d2.

9.5 Context Switch

In addition to instructions that implicitly save and restore contexts, such as Calls and Returns (CALLs & RETs), the TriCore instruction set also includes instructions that allow a task's context to be explicitly saved, restored, loaded and stored. These instructions are detailed in the sections which follow.

9.5.1 Context Saving and Restoring (SVLCX, RSLCX)

A task's upper context is always automatically saved on a call, interrupt or a trap, and is automatically restored on a return. However, the lower context of a task must be explicitly saved or restored.

The SVLCX instruction (Save Lower Context) saves registers A2 through A7 and D0 through D7, together with the return address in the register A11 from the saved PC field. This operation is performed when using the FCX and PCX pointers to manage the CSA (Context Save Area) lists.

RSLCX (Restore Lower Context) restores the lower context. The RSLCX instruction loads registers A2 through A7 and D0 through D7 from CSA. It also loads A11 from the saved PC field. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

Example:

To solve the equation $(x + 0.5)^2$

```
movh    d5, #0x4000    ;move 0.5
ld.w    d4, 1X6        ;load the data
svlcx                   ;save lower context
adds    d4, d4, d5      ;x + 0.5
mul      d10, d4, d4     ; (x + 0.5)2
rslcx                   ;restore lower context
```

In this example, the lower context is saved in the Context Save Area. Register *d4* will not be changed after restoring the lower context; i.e. *d4* = 1X6 at the end, not (1X6 + *d5*). The result will be in register *d10*.

9.5.2 Context Loading and Storing (STUCX, LDUCX, STLCX, LDLCX)

The effective address of the memory area where the context is stored to, or loaded from, is part of the load and store instruction. The effective address must address a memory location aligned on a 16-word (512-bit) boundary, otherwise an address alignment trap is generated.

The STUCX instruction (Store Upper Context) stores the same context information that is saved with an implicit upper context save operation: Registers A10 to A15 and D8 to D15, and the current PSW and PCXI.

The LDUCX instruction (Load Upper Context) loads registers A10 to A15 and D8 to D15. The PSW and link word fields in the saved context in memory are ignored. The PSW, FCX and PCXI are unaffected.

The STLCX instruction (Store Lower Context) stores the same context information that is saved with an explicit lower context save operation: Registers A2 to A7 and D0 to D7, and the current PSW and PCXI.

The LDLCX instruction (Load Lower Context) loads registers A2 through A7 and D0 through D7. The saved return address and the link word fields in the context stored in memory are ignored. Registers A11, FCX and PCXI are not affected.

```
stucx    <mode>      ;Store upper context
lducx    <mode>      ;Load upper context
stlxc    <mode>      ;Store lower context
ldlxc    <mode>      ;Load lower context
```

<mode> is absolute addressing mode and base+offset addressing mode.

Both of the following examples solve the equation $(x + 0.5)^2 + 1$

Example 1:

```
movh     d5,#0x4000 ;move 0.5
ld.w     d4,lx6     ;load x
lea      a2,0x80    ;initialization of the lower context area address
adds     d2,d4,d5   ;x + 0.5
mov      d4,d2      ;load the result
movh     d5,#0x8000 ;load -1
stlxc    [a2]       ;store lower context at the specified address
mul      d2,d4,d4    ;(x + 0.5)2
subs     d12,d2,d5   ;(x + 0.5)2 + 1
ldlxc    [a2]       ;load lower context
```

In example 1 above, the lower context is saved at the address specified by a2, so register d2 will not be changed after the load of the lower context. The result will be in register d12.

Example 2:

```
movh    d5,#0x4000 ;move 0.5
ld.w    d4,lX6      ;load x
lea      a3,0xc0     ;initialization of the upper context area address
adds     d2,d4,d5    ;x + 0.5
mov      d11,d2      ;save the result
mov      d4,d11      ;load the data
movh     d5,#0x8000 ;load -1
stucx    [a3]        ;store upper context at the specified address
mul      d11,d4,d4    ;(x + 0.5)2
subs     d2,d11,d5    ;(x + 0.5)2 + 1
lducx    [a3]        ;load upper context
```

In this second example, the upper context is saved at the address specified by *a3*, so register *d11* will not be changed after the load of the upper context. The result will be in register *d2*.

This example points out the address values. They *must be aligned on the 16 registers* that are stored or loaded. This means that the address must be a multiple of 0x40.

9.6 Applications

The applications covered here are:

- **Arithmetic Equation Using Subroutine Call**
- **Sweet64**

9.6.1 Arithmetic Equation Using Subroutine Call

Example:

The following example is to solve the simple equation $y = (0.5 * x - 1) / 2$

After Instruction		
movh d5, #0x4000	d5	4000 0000 ;load 0.5
ld.w d4, 1x6	d4	080F 15C8 ;load the data
call Lmult		;0.5*x will return
mov d11, d2	d11	0407 8AE4 ;keep the result
mov d10, d2	d10	0407 8AE4 ;keep the result
movh d4, #0x8000	d4	8000 0000 ;load -1
mov d5, d11	d5	0407 8AE4 ;load the previous result
call Ladd		;0.5*x - 1 will return
mov d4, d2	d4	8407 8AE4 ;load the previous result
mov.u d5, #1	d5	0000 0001 ;load 1
call Lshr		; (0.5*x - 1)/2 will return
Ladd: adds d2, d4, d5	d2	8407 8AE4 ;addition with saturation
ret16		
Lmult: mul.q d2, d4, d5, #1	d2	0407 8AE4 ;multiplication of two fractions
ret16		
Lshr: rsub d6, d5, #0	d6	FFFF FFFF ;negate the shift count
mov d8, #0x1f	d8	0000 001F ;move 31

rsub	d9, d8, #0	d9	FFFF FFE1	;move -31
lt	d14, d6, d8	d14	0000 0001	;compare the shift count with 31
ge	d15, d6, d9	d15	0000 0001	;compare the shift count with -31
sel	d10, d14, d9, d8	d10	FFFF FFE1	;select the good shift count
and	d15, d15, d14	d15	0000 0001	;check if it is in the range
cmovn	d6, d15, d10	d6	15.3 FFFF FFFF	;select the good shift
shas	d2, d4, d6	d2	C203 C572	;right shift the value
retl6				

9.6.2 Sweet64

This application executes 64-bit operations. All the calls refer to the functions created in previous chapters.

Example:

Equation $(x + y) * 256 - \text{constant}$; x, y constants are all in 64-bit format.

```
ld.d    e4, eX4           ;load data x
ld.d    e6, eX6           ;load data y
call    add64             ;x + y will return
mov64   d4, d5, d2, d3    ;load result
mov     d6, #0x08         ;load shift count
call    shl64             ;(x + y)*256 will return
mov64   d8, d9, d2, d3    ;load result
call    sub64c            ;(x + y) * 256 - constant
```

After Instruction

ld.d	e4, eX4	e4	00000003 00000010	;load x
ld.d	e6, eX5	e6	00000004 00000005	;load y
call	add64			;x + y will return
mov64	d4, d5, d2, d3	d4 d5	0000 0015 0000 0007	;load the result

mov d6, #8	d6	0000 0008	;load shift count
call shl64			;(x + y)*256 will return
mov64 d8, d9, d2, d3	d8 d9	0000 1500 0000 0700	;load the result
call sub64c j the_end add64:			;(x + y) * 256 – constant
addx d2, d4, d6	d2	0000 0015	;add the lower registers
addc d3, d5, d7	d3	0000 0007	;add the upper registers
ret16 shl64:			
sh d2, d4, d6	d2	0000 1500	;shift lower register
dextr d3, d5, d4, d6	d3	0000 0700	;shift upper register
ret16 sub64c:			
mov d1, #0	d1	0000 0000	;initialize the register
addi d0, d1, #0x5432	d0	0000 5432	;add lower part of the constant
addih d0, d0, #0x9876	d0	9876 5432	;add upper part of the constant
subx d2, d8, d0	d2	6789 C0CE	;subtract the lower registers
addi d4, d1, #0	d4	0000 0000	;add lower part of the constant
addih d4, d4, #0x9999	d4	9999 0000	;add upper part of the constant
subc d3, d9, d4	d3	6667 06FF	;subtract the upper registers
ret16 the_end:			

9.7 Summary Table: Displacement Field Width

Name	Definition	Displacement field width (bits)		Syntax Register	Syntax Constant
		32-bit opcode	16-bit opcode		
j	Jump	24	8	j LABEL	
ja	Jump absolute	24		ja LABEL	
ji	Jump indirect		X	ji a2	
jz	Jump if equal to zero		8 or 4	jz d2.LABEL	
jnz	Jump if not equal to zero		8 or 4	jnz d2.LABEL	
jltz	Jump if less than zero		4	jltz d2.LABEL	
jlez	Jump if less than or equal to zero		4	jlez d2.LABEL	
jgez	Jump if greater than or equal to zero		4	jgez d2.LABEL	
jgtz	Jump if greater than zero		4	jgtz d2.LABEL	
jeq	Jump if equal to	15	4	jeq d0,d1,LABEL	jeq d0,k4,LABEL
jne	Jump if not equal to	15	4	jne d0,d1,LABEL	jne d0,k4,LABEL
jlt	Jump if less than	15		jlt d0,d1.LABEL	jlt d0,k4,LABEL
jlt.u	Jump if less than (unsigned)	15		jlt.u d0,d1,LABEL	jlt.u d0,k4,LABEL
jge	Jump if greater than or equal to	15		jge d0,d1,LABEL	jge d0,k4,LABEL
jge.u	Jump if greater than or equal to (unsigned)	15		jge.u d0,d1,LABEL	jge.u d0,k4,LABEL

jz.t	Jump if equal to zero bit	15	4	jz.t d0:3, LABEL	
jnz.t	Jump if not equal to zero bit	15	4	jnz.t d0:3, LABEL	
jeq.a	Jump if equal to address	15		jeq.a a2, a3, LABEL	
jne.a	Jump if not equal to address	15		jne.a a2, a3, LABEL	
jz.a	Jump if equal to zero address	15	4	jz.a a2, LABEL	
jnz.a	Jump if not equal to zero	15	4	jnz.a a2, LABEL	
jned	Jump if not equal to and decrement	15		jned d0, d1, LABEL	jned d0, k4, LABEL
jnei	Jump if not equal to and increment	15		jnei d0, d1, LABEL	jnei d0, k4, LABEL
jl	Jump and link	24		jl LABEL	
jla	Jump and link absolute	24		jla LABEL	
jli	Jump and link indirect			jli a2	
call	Call	24	8	call LABEL	
calla	Call absolute	24		calla LABEL	
calli	Call indirect			call LABEL	
ret	Return from a call			ret	

10 DSP and MAC

DSP is a science that cannot be covered in a single chapter of an optimization guide. Or can it?

It is often the case that a large number of DSP programmers do not develop new algorithms, but implement standard algorithms, usually in assembly language. The biggest part of the knowledge is therefore in mapping DSP equations onto fixed-point Digital Signal Processors. This mapping is more easily accomplished if the DSP has a rich set of instructions.

TriCore has such a rich set of instructions. MAC instructions were developed with DSP applications such as standard speech coders, specifically in mind.

This chapter is divided into 2 parts. Part 1 gives an introduction to DSP arithmetic. Part 2 gives a detailed description of instructions, with an emphasis on all possible subtypes.

Reading this chapter will provide a very good insight into the flexibility available in all of the TriCore instructions.

Chapter contents:

- **Introduction to DSP Arithmetic**
- **Multiplication**
- **Multiply-Add**
- **Multiply-Subtract (MSUB.Q, MSUBR.Q)**
- **Multiply-Accumulate**
- **Applications**
- **Summary Tables**

10.1 Introduction to DSP Arithmetic

The most common DSP arithmetic challenges are:

- Mapping of a fractional number to a CPU register
- Misunderstanding of the multiplication operation
- Confusion between register width and data width

For example, multiplying 2 16-bit signed numbers produces a 31-bit result in a 32-bit register. Where is the missing bit?

Multiplying 2 16-bit signed numbers gives a 31-bit result, but only 16 bits are stored. The question then becomes which part is stored, upper or lower?

10.1.1 Fractional Arithmetic

DSP algorithms often use the fractional representation of numbers.

- Generation of sine and cosine waves
- Filter coefficients used in FIR and IIR (generated by filter development tools)
- Sine and cosine coefficients used in FFT

Representing a binary number as a fraction simplifies the equation.

Note: The Table rows in the following examples are (top to bottom):

Register bits

Hexa

Binary

Fractional weight

The value of 16-bit register = 0x56B7 is 0.6774597025:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
5				6				B				7			
0	1	0	1	0	1	1	0	1	0	1	1	0	1	1	1
-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

To represent 0.6774597025 in a 32-bit register, represent it as a 16-bit with the added least significant half word filled with zero:

31	30	29	28	27	26	25	24	23	22	21	20																				
5				6				B																							
0	1	0	1	0	1	1	0	1	0	1	1																				
2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}																				
												19	18	17	16	15:0															
7				0000																											
0	1	1	1	000000000000000000																											
2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}	$\dots\dots\dots 2^{-31}$																										

Example - The minimum and maximum values in a fractional format:

A fraction is a number that belongs to range: [-1; +1). Note the exclusion of +1.

Maximum negative value (-1) in a 16-bit and 32-bit register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8				0				0				0			
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
8				0				0				0			
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}
15:0															
0000															
0000000000000000															
2^{-16}				$\dots\dots\dots 2^{-31}$											

Maximum positive value (very near to +1) in a 16-bit and 32-bit register:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7				F				F				F			
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
7				F				F				F			
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

15:0

FFFF
1111111111111111
2^{-16} 2^{-31}

One advantage of fractions is that the number value remains quasi identical, independently of the register length.

Therefore 0x7FFF, or 0x7FFFFFFF, or 0x7FFF0000 can be considered to have the same values:

- 0x7FFF = 0.9999694824218750
- 0x7FFFFFFF = 0.9999999953433871269226074218750
- 0x7FFF0000 = 0.99996948242187500000000000000000

It is a matter of precision, and precision is variable. For instance, a DSP application working with 16-bit precision will consider the three numbers shown above, as equivalent.

For practical purposes, the maximum fractional positive number is considered to be 0.99997. With the next two numbers being 0.99994 and 0.99991, it is evident that five decimal digits are sufficient to represent signed 16-bit numbers.

10.1.2 Multiplication Problem: Word Growth

The initial problem with multiplication is that the inputs and the result do not have the same data size. The result will have twice the number of bits.

input 1	ABCDEFGH
input 2	ABCDEFGH
	MULTIPLICATION
result	ABCDEFGH IJKLMNOP
input 1	10000000
input 2	12345678
	MULTIPLICATION
result	123456780000000

There are three solutions:

1. Retain the complete result with twice the number of digits. For each multiplication of the results, the number of digits will double. Unfortunately this solution very quickly becomes impractical.
2. Retain the lower part of the result. The numbers are considered right aligned and will grow towards the left (C language also operates in this manner).

input 1	ABCDEFGH
input 2	ABCDEFGH
	MULTIPLICATION
result	ABCDEFGH IJKLMNOP
keep lower	IJKLMNOP
input 1	10000000
input 2	12345678
	MULTIPLICATION
result	123456780000000
keep lower	80000000

The problem with this second solution is that the result is incorrect. This solution will work if the numbers are smaller than half the size. In practical terms, this means that $\text{longZ} = \text{longX} * \text{longY}$ can only work if longX and longY are shortX and shortY .

input 1	00001000
input 2	00001234
	MULTIPLICATION
result	0000000001234000
keep lower	01234000

Another case would be where one number is very small compared to the other, typical of an index calculation for example.

3. Retain the upper part of the number.

The numbers are considered left aligned and will grow towards the right. The upper part represents the most significant part of any number. Although C language does not work using this method, it is the preferred format for DSP algorithms.

input 1	ABCDEFGH
input 2	ABCDEFGH
	MULTIPLICATION
result	ABCDEFGH IJKLMNOP
keep upper	ABCDEFGH

input 1	10000000
input 2	12345678
	MULTIPLICATION
result	1234567800000000
keep upper	12345678

12345678 is not the same as 12345678 with 7 zeros behind it, because these numbers are seen as integer values.

When writing fractional values the problem disappears (see next example).

input 1	0.1000000
input 2	0.1234567
	MULTIPLICATION
result	0.012345670000000
keep upper	0.01234567

The fields of computer arithmetic and DSP are in agreement on the fractional representation of numbers.

Note: MULTIPLICATION - To keep the most significant part, retain the UPPER (or LEFT) halfword of the result. Numbers are then interpreted as fractional.

*Note: Any integer can be represented in a multiplier. Consider $1000 * 10000$ as $1000 / 32768 * 10000 / 32768$.*

10.1.3 Multiplication Problem: Signed Multiplication

When multiplying two fractional 16-bit numbers, the minimum number is:

$$2^{-15} * 2^{-15} = 2^{-30}$$

The maximum number is less than 1.

The result will be in the range: $-2^0 \dots 2^{-30}$ which is 31-bit wide. There is no need for a 32nd bit. However the register has 32 bits.

input 1	$-2^0 \dots 2^{-15}$
input 2	$-2^0 \dots 2^{-15}$
	MULTIPLICATION
result	$-2^0 \dots 2^{-30}$
	MULTIPLIER
result	$-2^1 \dots 2^{-30}$
	<<1
result	$-2^0 \dots 2^{-31}$

As shown in this example, a multiplier will produce a result with 1 more bit to the left. There are 2 sign bits. By removing 1 sign bit we keep the same output format as the input. This is called alignment, or left-alignment. Left-shift the result by 1 and bit 0 (2^{-31}) becomes 0.

10.1.4 Q-Format

Q15 format

DSP and multipliers prefer a fractional format. The most common format is called Q15 (or 1Q15).

Q15 means that the binary word is interpreted as having a decimal point just after the most significant bit (or bit 15).

Example: 0.5

Note: Reading from the top to the bottom of the table, the rows are:

Register bits

Hex

Binary

Fractional weight

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4				0				0				0			
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-2^0	2^{-1}	2^{-2}	2^{-3}	-2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

The impact of Q15 format on hardware in TriCore is minimal:

- Addition / shift, etc. = no change
- Multiplier = (with the option) to keep the upper part of the result
- Multiplier = (with the option) to left align the result

Generalized Q format

Q format is a universal format. From a developer's point of view, the definition of Q format is where the q notation indicates the imagined binary point in the binary word.

- For a 16-bit signed fractional value this is called the 1q15 format (Q15).
- For a 16-bit signed value (range $-2..+2$) this is called the 2q14 format (Q14).
- For a 16-bit signed value (range $-4..+4$) this is called the 3q13 format.
- For a 16-bit signed value (range $-16..+16$) this is called the 5q11 format.
- For a 32-bit signed fractional value it is called 1q31 format (Q31).

Example: 0.6774... in Q15 and Q14

This value is in Q15: 0x56B7 and in Q14: 0x2B5B. It should be noted that there is less precision in Q14.

Q15															0.6774597025														
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1															
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}														

Q14														0.6774597025													
0	0	1	0	1	0	1	1	0	1	0	1	1	0	1	1												
-2^{-1}	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}												

The Q format term applies to all fractional arithmetic or (more precisely) all left-aligned arithmetic.

Multiplying two q-format numbers:

- A value in 1q15 multiplied by a value in 1q15 will give a value in 2q30 format.
- A value in 2q14 multiplied by a value in 1q15 will give a value in 3q29 format.

The format of the result is calculated by adding the two formats of the operands.

For example:

1q15	1q15
+ 1q15	+ 2q14
2q30	3q29

Adding q-format numbers:

The input format must be identical, so that the output format is the same as the input format.

Example: $sZ = sX * 0.707 + SY$

sX	1q15
0.707	<u>+ 1q15</u>
	2q30
left aligned (<< 1)	1q31
keep 16-bit	1q15
add sY	1q15
sZ	1q15

Example: $sZ = sX * 1.414 + SY$

sX	1q15
1.414	<u>+ 2q14</u>
	3q29
left aligned (<< 1)	2q30
keep 16-bit	2q14
SY	1q15
SY >> 1	2q14
add sY	2q14
sZ	2q14

The 2q14 format is used because the coefficient belongs to the $[-2; +2)$ range.

Not having all the values in 1q15 format creates the need to shift values. In addition, the final value in 2q14 might create problems further down in the algorithm.

10.1.5 Left-Alignment

All TriCore DSP instructions can left-align their result as part of the MUL, MADD, and MSUB instructions. Left alignment means that the MULTIPLICATION result is left-shifted by 1-bit to discard the redundant sign bit.

Example:

d1	1234 0000
d2	7BFF 0000

After Instruction

`mul.q d0, d1, d2, #1`

d0	11A2 3B98
----	-----------

;The result is left shifted

10.1.6 Multiplication Saturation

All TriCore MAC instructions with .Q and .H suffixes use the q format, so *a multiplication in q-format never overflows*.

However, a problem arises because of the unbalanced nature of 2's complement arithmetic. When multiplying the two maximum negative values (-1 in Q15), the result should be the maximum positive number (+1 in Q15).

For example, $0x8000 * 0x8000 = 0x4000\ 0000$, is correctly interpreted as:

$$-1\ (1q15\ format) * -1\ (1q15\ format) = +1\ (2q30\ format)$$

However, when the result is shifted left by 1, the result is incorrectly interpreted:

$$0x8000 * 0x8000 = 0x4000\ 0000 <<1 = 0x8000\ 0000$$

$$-1\ (1q15\ format) * -1\ (1q15\ format) = -1\ (1q31\ format)$$

To avoid this problem, the result of Q multiplication $(-1 * -1)$ that has been left shifted by 1 (left aligned), is saturated to the maximum positive value. So, $0x8000 * 0x8000 = 0x7FFF\ FFFF$, is correctly interpreted in Q format as:

$$-1\ (1q15\ format) * -1\ (1q15\ format) = (\text{nearest representation of}) +1\ (1q31\ format)$$

This operation is completely transparent to the user and does not set the overflow flags.

As shown in the table which follows, only the 16x16 does the correction.

Instruction	Overflow	Maximum negative saturation
16x16 -> 16	Not possible	Done
16x32 -> 32	Not possible	Not detected
32x32 -> 32	Not possible	Not detected
16x32 -> 64	Not possible	Not detected
32x32 -> 64	Not possible	Not detected

10.1.7 Rounding and Multi-Precision

Instruction	Rounding	Multi-precision
16x16 → 32	Yes	No
16x32 → 32/64	No	Yes
32x32 → 32/64	No	Yes
64 +/- (16x16) → 32/64	Yes	Yes
64 +/- (16x32) → 32/64	No	Yes
64 +/- (32x32) → 32/64	No	Yes

Rounding

A 32-bit number is rounded into a 16-bit value. The process of rounding gives better precision than a simple truncation. It adds 0x00008000 to the 32-bit value. So if the lower half is less than 0x8000, it will not appear on the upper half (and respectively, if the lower half is bigger than 0x8000, it will appear on the upper half). The lower half is then zero-filled.

*Note: Rounding is only applied to 16*16-bit cases to return to the original 16-bit format.*

Multi-precision

Multi-precision means that the result goes into a 64-bit register. Note that there is no multi-precision for 16*16-bit multiplication because the maximum size for this multiplication is 32-bit.

64-bit result is useful in two cases:

- For precision on 16x32 or 32x32 operations
- To allow for growth bits in accumulation (valid for all 3 cases; 16x16, 16x32, and 32x32)

10.1.8 Growth Bits

Instruction	Number of growth bits
64 +/- (16x16) → 64	16 ¹⁾
64 +/- (16x32) → 64	16
64 +/- (32x32) → 64	0

¹⁾ The result of the multiplication is shifted left by 16 before being added.

10.1.9 32-bit DSP

TriCore deals easily with the question of accumulating a 32x32 with no growth bits. Reasons for using 32-bit values include:

- Some applications need more than 16-bit for representing the signals
- Some programmers find it too difficult to work in 16-bit
- As mechanical precision increases, so does the need for more precise coefficients
- Accumulation, especially when accumulating a series of multiplications over time (adaptive filtering is a good example) requires growth (also called guard) bits.

10.2 Multiplication

This section describes 6 types of multiplication, which all use the MUL.Q mnemonic:

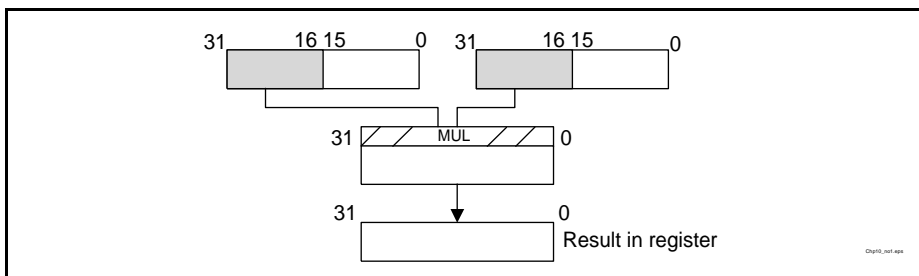
- 16*16-bit → 32 Multiplication (MUL.Q)
- 16*32 → 32 Multiplication (MUL.Q)
- 32*32 → 32 Multiplication (MUL.Q)
- 16*32 → 48 Multiplication (MUL.Q)
- 32*32 → 64 Multiplication (MUL.Q)
- 16*16 → Round(32) Multiplication (MULR.Q)

10.2.1 16*16-bit → 32 Multiplication (MUL.Q)

This 16*16-bit is a signed multiplication. It gives a 32-bit result. There are two possible outcomes:

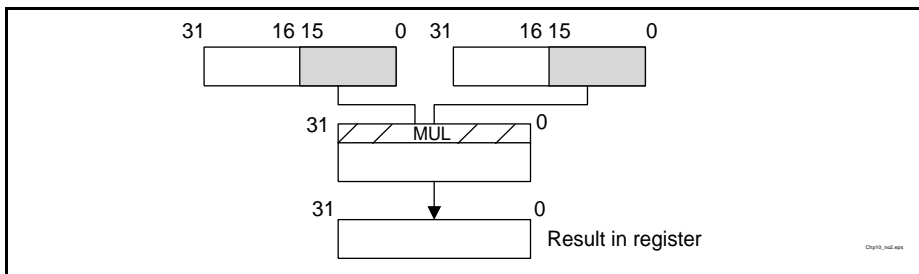
1. Multiplying the upper half of the registers

```
mul.q    d0,d1u,d2u,n    ;n=0 or 1 == left shift
```



2. Multiplying the lower half of the registers

```
mul.q    d0,d1l,d2l,n
```



The suffix “u” (or “U”) that follows the registers indicates that they are working with the upper halves. If the lower halves are preferred, use an “l” (or “L”) suffix.

Example:

d1	6000 5000
d2	0030 0040

After Instruction

```
mul.q d0,d1u,d2u,#1
```

```
mul.q d3,d1l,d2l,#1
```

d0	0024 0000
d3	0028 0000

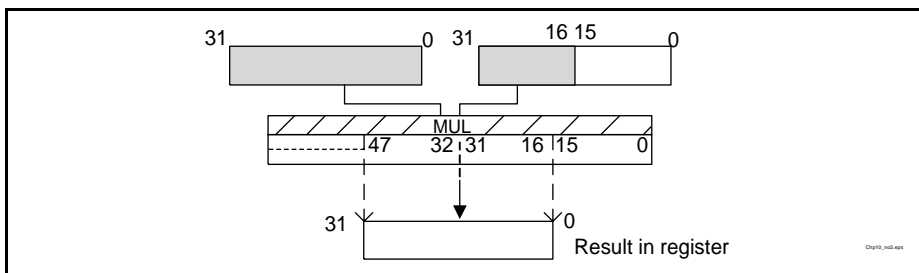
Note: When both inputs are 0x8000 and the result is left aligned, there is automatic correction of overflow. No overflow flag is generated.

10.2.2 16*32 ➡ 32 Multiplication (MUL.Q)

The 16*32-bit is a signed multiplication. It gives better precision than a 16x16, since one input value has twice the precision. Two methods are possible:

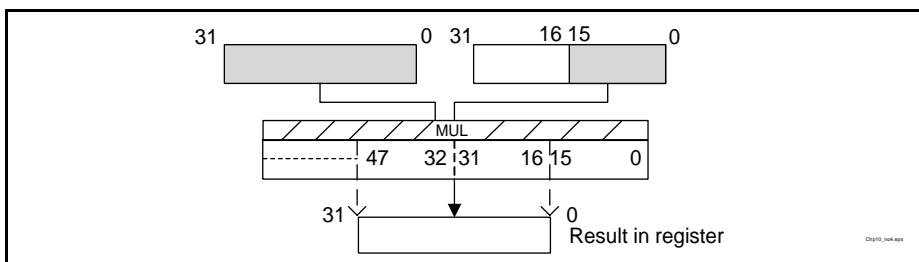
1. Multiplying a 32-bit value with the upper 16-bit of the other register:

```
mul.q d0,d1,d2u,n
```



2. Multiplying a 32-bit value with the lower 16-bit of the other register:

```
mul.q d0,d1,d2l,n
```



Multiply 32*16-bit values, so “u” (or “l”) is indicated only for the register, which contains the upper (or lower) value. This register is always the second source register.

The most significant 32 bits of the 48-bit result are kept.

Examples:

d1	6000 1234
d2	0030 0040

After Instruction

```
mul.q    d0,d1,d2u,#1
```

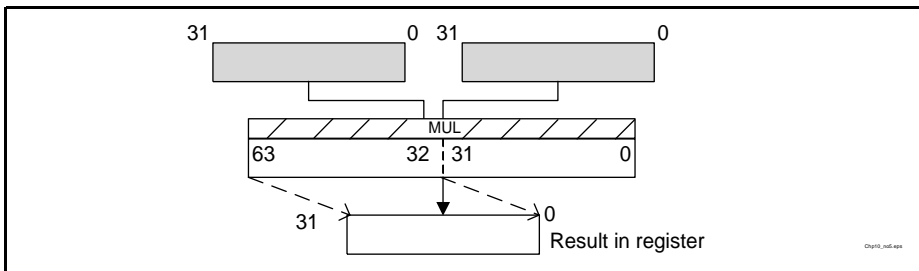
d0	0024 0006
d3	0030 0009

```
mul.q    d3,d1,d2l,#1
```

10.2.3 32*32 → 32 Multiplication (MUL.Q)

The 32*32-bit is a signed multiplication. It provides good precision on initial values but gives a result with reduced precision result because only half of the bits are kept. Since the input values are 32-bit values, the result is more precise than a 16x16 or 16x32 multiplication.

```
mul.q    d0,d1,d2,n
```



As the full 32-bit registers are used, the notation of the registers “u” or “l” is not applied.

Example:

d1	6000 1234
d2	0030 0040

After Instruction

```
mul.q    d0,d1,d2,#1
```

d0	0024 0036
----	-----------

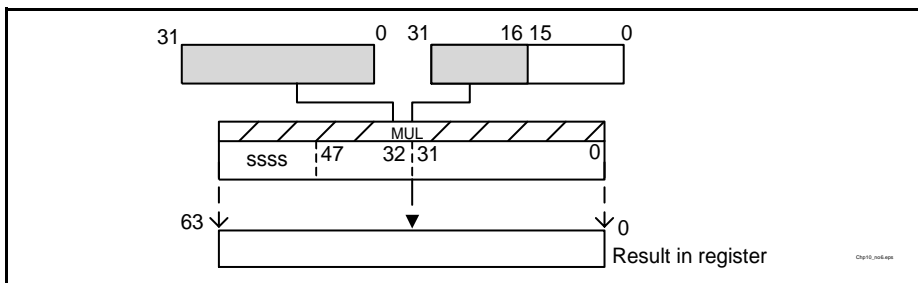
As with the 16*32-bit multiplication, the 32*32 cannot produce overflow, so there is no need for saturation. Only the most significant 32-bit of the 64-bit temporary result is kept.

10.2.4 16*32 → 48 Multiplication (MUL.Q)

To obtain the maximum precision with the 16*32-bit multiplication, the complete 48-bit result is kept in a 64-bit register. There are two syntaxes:

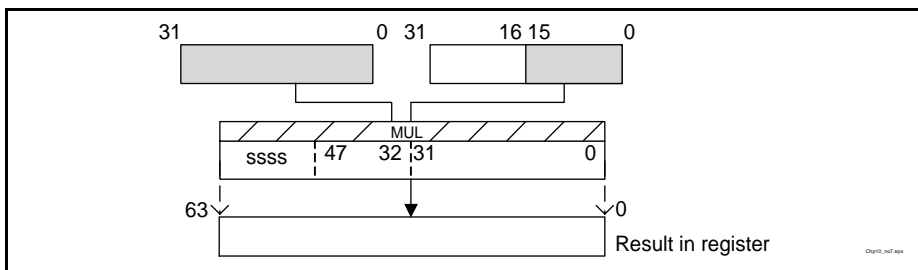
1. Upper half

`mul.q e0,d2,d3u,n`



2. Lower half:

`mul.q e0,d2,d3l,n`



Note: In the figure above the result is put into bits 47...0 of a 64-bit register, and bits 63...48 are filled with signs. This means that the result is right aligned. This is unexpected since all Q format is left-aligned. However, the reason behind that choice is to keep the 16x32 MUL consistent with the 16x32 MAC. In a 16x32 accumulation needs to be right-aligned for growth bits (up to 16).

*Note: **ssss** means sign-extension; (bits 63..48 = sign of the result).*

Example:

d1	6000 1234
d2	6000 5000

After Instruction

`mul.q e4,d1,d2u,#1`

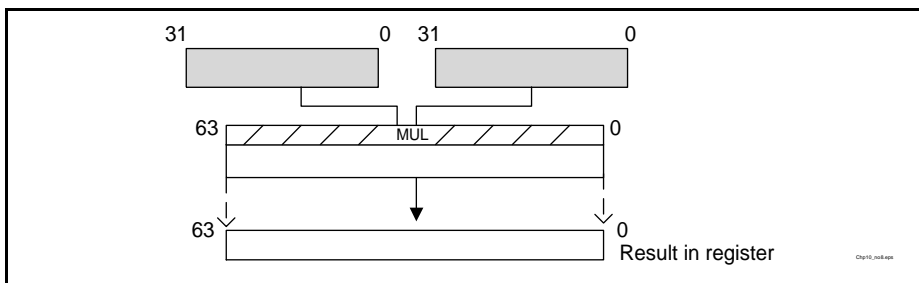
e4	0000 4800 0DA7 0000
e6	0000 3C00 0B60 8000

`mul.q e6,d1,d2l,#1`

10.2.5 32*32 → 64 Multiplication (MUL.Q)

This instruction gives the maximum precision for 32-bit multiplication:

`mul.q e0,d2,d3,n`



Example:

d1	6000 1234
d2	6000 5000

After Instruction

`mul.q e4,d1,d2,#1`

e4	4800 49A7 0B60 8000
----	---------------------

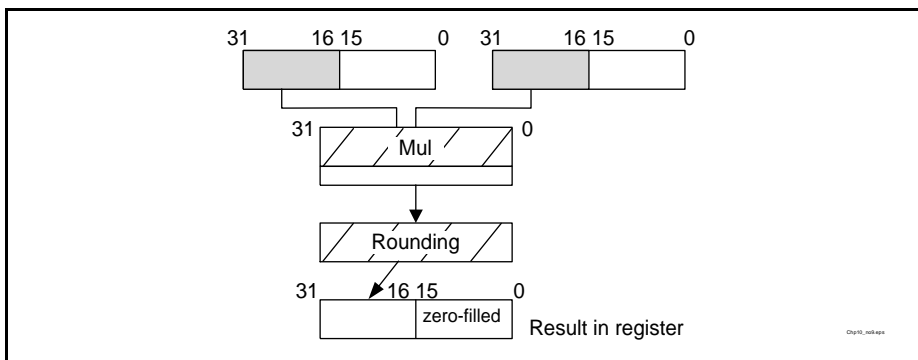
10.2.6 16*16 → Round(32) Multiplication (MULR.Q)

The result is rounded just before it is transferred to the destination register. A result is rounded by adding 0x8000 and clearing bits 15 to 0 (See Rounding and Multi-Precision section).

There are two cases:

`mulr.q d0,d1u,d2u,n`

`mulr.q d0,d1l,d2l,n`



Example 1:

d1	0030 0040
d2	6000 5000

After Instruction

<code>mulr.q d0,d1u,d2u,#1</code>	d0	0024 0000
<code>mulr.q d3,d1l,d2l,#1</code>	d3	0028 0000

The rounded result does not affect the overflow flag, and saturation is not required.

Note: Despite adding 0x8000, the MULR.Q instruction cannot overflow.

Explanation:

An overflow problem would be expected when adding 0x8000 to the maximum positive result of a multiplication, especially if the result has been left aligned (<<1). This does not occur.

In the general case the maximum positive result = $(-max) * (-max + 1)$.

In 16 bits, this will be: $(-2^{15}) * (-2^{15} + 1) = 2^{30} - 2^{15}$

2^{15}	0000 0000 0000 0000	1000 0000 0000 0000
-2^{15} (2's complement)	1111 1111 1111 1111	1000 0000 0000 0000
2^{30}	0100 0000 0000 0000	0000 0000 0000 0000
$2^{30}-2^{15}$	0011 1111 1111 1111	1000 0000 0000 0000
$<<1$	0111 1111 1111 1111	0000 0000 0000 0000
added with 0x8000	0111 1111 1111 1111	1000 0000 0000 0000
Rounded	0111 1111 1111 1111	0000 0000 0000 0000

Example 2:

0x8000 * 0x8001

					1000	0000	0000	0000
				x	1000	0000	0000	0001
	111	1111	1111	1111	1000	0000	0000	0000
	100	0000	0000	0000	0...
	1011	1111	1111	1111	1000	0000	0000	0000
$<<1$	0111	1111	1111	1111	0000	0000	0000	0000
added with 0x8000	0111	1111	1111	1111	1000	0000	0000	0000
Rounded	0111	1111	1111	1111	0000	0000	0000	0000

With a 4-bit number this is:

$-7 * -8$ normalised in fractional would be $-0.875 * -1 = 0.875$

($1001 * 1000 = 0111$ in 4-bit format)

	1000
x	1001
<hr/>	
11111000	
0000000.	
000000..	
01000...	(2's complement of 11000)
<hr/>	
00111000	

The result value needs to be shifted left by 1 to be numerically correct. So we obtain 0111 0000. This is the good result. With rounding applied, it would give 0111.

10.3 Multiply-Add

This section gives 7 sub-types of multiply-add:

1. **32 + 16*16 Multiply-Add (MADD.Q, MADDS.Q)**
2. **32 + 16*32 Multiply-Add (MADD.Q, MADDS.Q)**
3. **32 + 32*32 Multiply-Add (MADD.Q, MADDS.Q)**
4. **48 + 16*16 Multiply-Add (MADD.Q, MADDS.Q)**
5. **64 + 16*32 Multiply-Add (MADD.Q, MADDS.Q)**
6. **64 + 32*32 Multiply-Add (MADD.Q, MADDS.Q)**
7. **32 + 16*16 \pm Round(32) Multiply-Add (MADDR.Q, MADDRS.Q)**

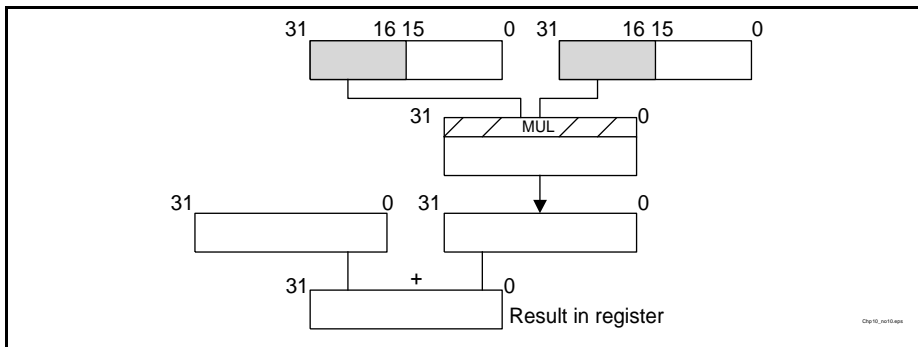
Note: There are actually 14 sub-types, since each sub-type can have normal or saturated behaviour. For all instructions the mnemonic is MADD(S).Q, with the exception of the rounding instruction (MADDR(S).Q).

10.3.1 32 + 16*16 Multiply-Add (MADD.Q, MADDS.Q)

There are two ways to execute a 16-bit multiply-add:

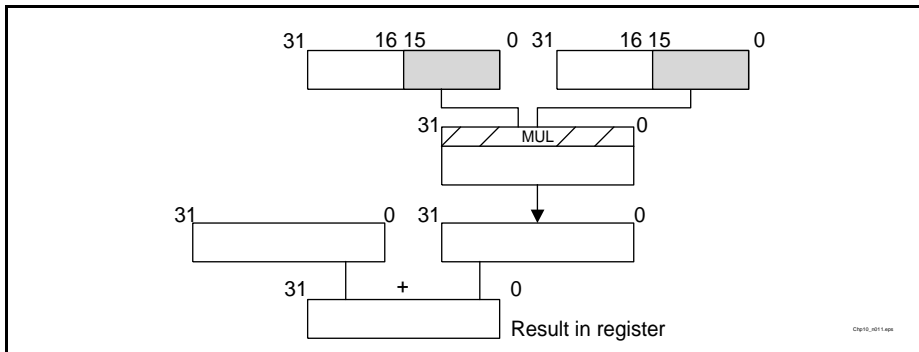
1. The two upper halves of the registers can be multiplied and their result added to another 32-bit value:

madd.q d0, d1, d4u, d5u, n



2. Multiply the two lower halves of the registers and add their result to another 32-bit value:

```
madd.q    d0, d1, d4l, d5l, n
```



Note: When upper halves are used, the suffix is “u”. For the lower halves, “l” is used after the registers.

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

After Instruction

```
madd.s.q  d0, d1, d4u, d5u, #1
```

d0	8023 1234
----	-----------

```
madd.s.q  d3, d1, d4l, d5l, #1
```

d3	8027 1234
----	-----------

OVF: The result of the addition cannot be represented in a 32-bit register.

Both instructions exist with saturation:

```
madds.q   d0, d1, d4u, d5u, #1
```

```
madds.q   d3, d1, d4l, d5l, #1
```

Saturation is 0x7FFFFFFF for positive results and 0x80000000 for negative results.

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

After Instruction

```
madds.q d0,d1,d4u,d5u,#1
```

d0	7FFF FFFF
d3	7FFF FFFF

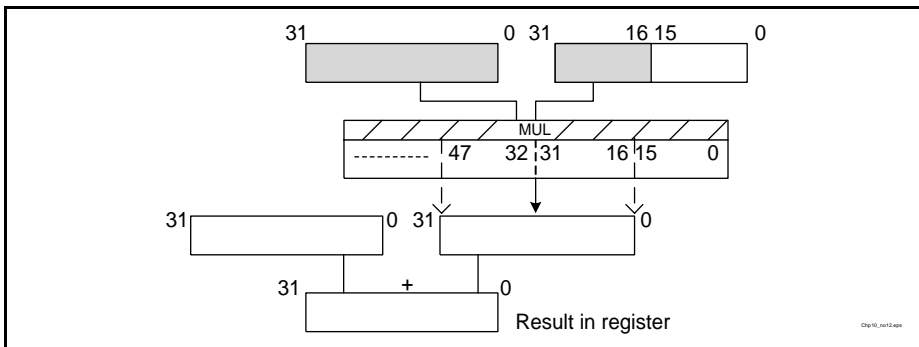
```
madds.q d3,d1,d4l,d5l,#1
```

10.3.2 32 + 16*32 Multiply-Add (MADD.Q, MADDS.Q)

There are two ways exist to execute a 16*32-bit multiply-add:

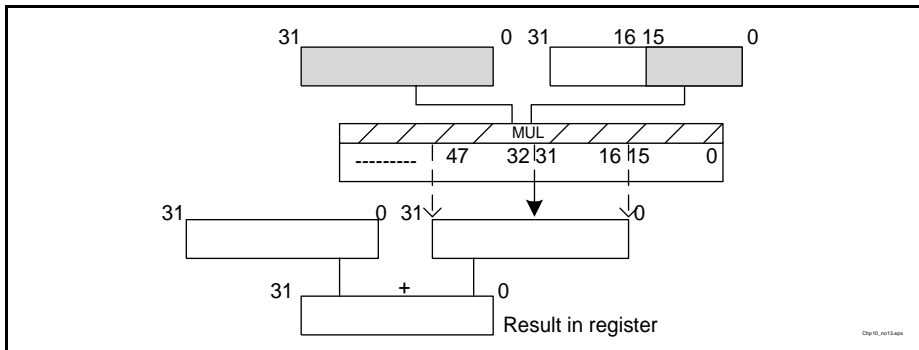
1. Multiply the upper half of the register with the other 32-bit register and add their result with another 32-bit value:

```
madd.q d0,d1,d4,d5u,n
```



2. Multiply the lower half of the register with the other 32-bit register and add the result with another 32-bit value:

```
madd.q    d0,d1,d4,d5l,n
```



Only one “u” (or “l”) is written after the second source register to show that the upper (or lower) half is taken.

OVF: The result of the addition cannot be represented in a 32-bit register.

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

After Instruction

```
madd.q    d0,d1,d4,d5u,#1
```

d0	8023 1252
d3	802F 125C

```
madd.q    d3,d1,d4,d5l,#1
```

This instruction also exists with saturation:

```
madds.q d0,d1,d4,d5u,n
madds.q d0,d1,d4,d5l,n
```

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

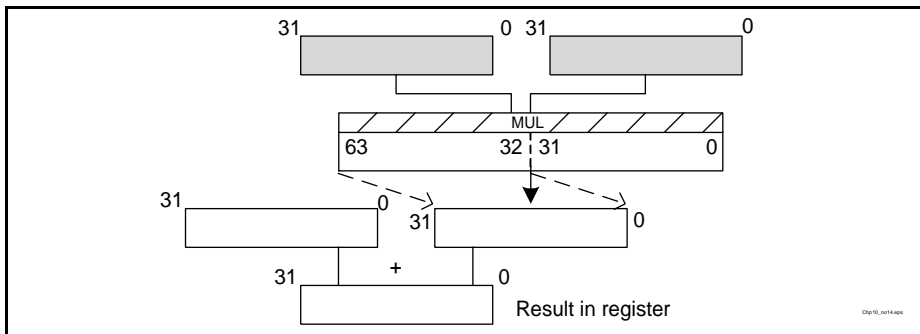
After Instruction

```
madds.q d0,d1,d4,d5u,#1
madds.q d3,d1,d4,d5l,#1
```

d0	7FFF FFFF
d3	7FFF FFFF

10.3.3 32 + 32*32 Multiply-Add (MADD.Q, MADDS.Q)

```
madd.q d0,d1,d4,d5,n
```



OVF: The result of the addition cannot be represented in a 32-bit register.

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

After Instruction

```
madd.q d0,d1,d4,d5,#1
```

d0	8023 1282
----	-----------

This instruction exists also with saturation.

```
madds.q d0,d1,d4,d5,n
```

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

After Instruction

```
madds.q d0,d1,d4,d5,#1
```

d0	7FFF FFFF
----	-----------

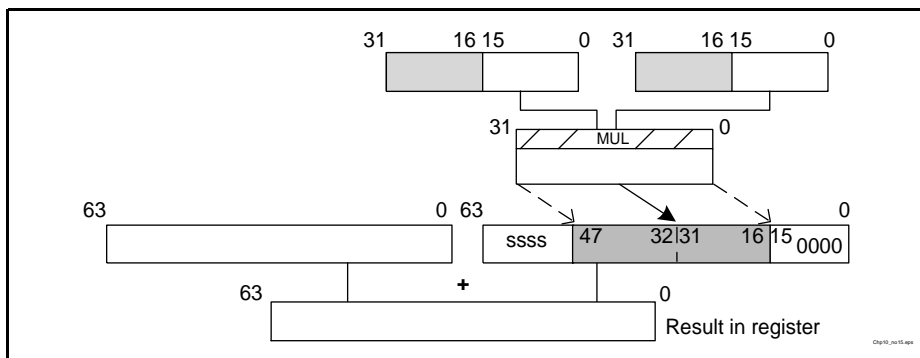
10.3.4 48 + 16*16 Multiply-Add (MADD.Q, MADDS.Q)

The main difference from the previous instances is that here there are two extended registers: one as the destination and the other as a source for the addition. The “u” and “l” are used the same way as previously described.

There are two ways to execute a 16-bit multiply-add:

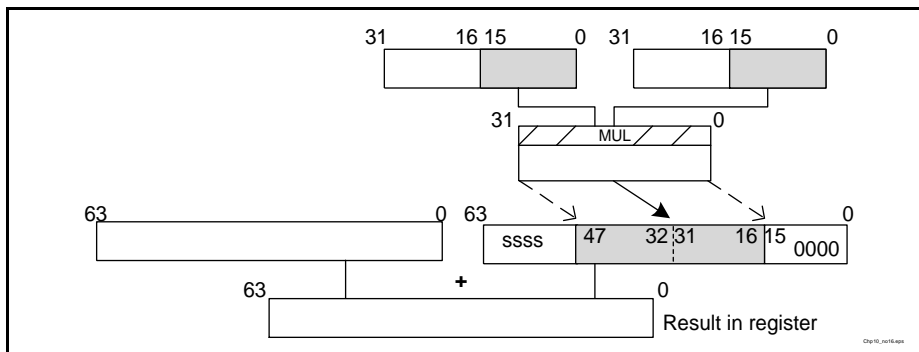
1. Multiply the upper half of the register with the other 32-bit register and add the result with a 64-bit value:

```
madd.q e0,e2,d4u,d5u,n
```



2. Multiply the lower half of the register with the other 32-bit register and add the result with a 64-bit value:

```
madd.q e0,e2,d4l,d5l,n
```



Notice that the 32-bit result of the multiplication is inserted in a 64-bit register starting at the 16th bit. This position is chosen because the result needs guard bits and can not be placed at the 32nd bit. It is not inserted at bit 0 because using 32 bits of guard bits is unnecessary.

The 64-bit value that is added to the result of the multiplication must have 16-bits of guard bits to be aligned like the 64-bit result of the multiplication, and the final result be coherent.

OVF: The result of the addition cannot be represented in a 64-bit register.

Example:

e2	7FFF FFEE 8014 57FA
d4	6000 5000
d5	0030 0040

After Instruction

madd.q e0,e2,d4u,d5u,#1	e0	8000 0022 8014 57FA
madd.q e0,e2,d4l,d5l,#1	e0	8000 0026 8014 57FA

```
madds.q    e0,e2,d4u,d5u,n
madds.q    e0,e2,d4l,d5l,n
```

e2	7FFF FFEE 8014 57FA
d4	6000 5000
d5	0030 0040

```
madds.q e0,e2,d4u,d5u,#1
madds.q e0,e2,d4l,d5l,#1
```

e0	7FFF FFFF FFFF FFFF
e0	7FFF FFFF FFFF FFFF

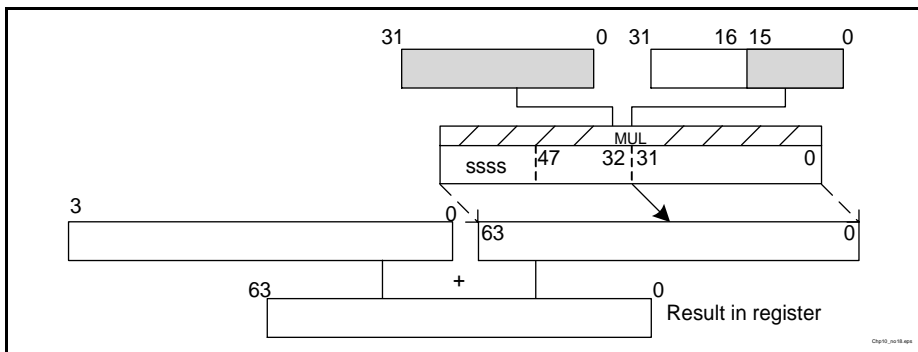
1. Multiply the upper half of the register with the other 32-bit register and add the result with a 64-bit value:

The diagram illustrates the MIPS architecture components and the execution of the `MUL` instruction:

- Registers:** Two 32-bit registers at the top provide inputs to the `MUL` instruction. The left register is labeled with bit positions 31 and 0. The right register is labeled with bit positions 31, 16, 15, and 0.
- MIPS Instruction:** A 32-bit instruction format is shown below the registers. It includes:
 - SSSS (Source Register Selectors):** 4 bits (bits 31-28).
 - MUL (Operation Code):** 6 bits (bits 27-22).
 - Destination Register Selector:** 5 bits (bits 21-17).
 - Immediate:** 16 bits (bits 16-0).
- Product:** A 64-bit horizontal bar represents the result of the multiplication. An arrow points from the `MUL` field of the instruction to this bar. The bar is labeled with bit positions 63 and 0.
- Result Register:** A 64-bit register at the bottom is labeled "Result in register". It contains the 64-bit product. The register is labeled with bit positions 63 and 0. A "+" sign is placed between the two 32-bit halves of the register.

2. Multiply the lower half of the register with the other 32-bit register and add the result with a 64-bit value:

```
madd.q e0,e2,d4,d5l,n
```



OVF: The result of the addition cannot be represented in a 64-bit extended register.

Example:

e2	7FFF FFEE 8014 57FA
d4	6000 5000
d5	0030 0040

After Instruction

```
madd.s q e0,e2,d4,d5u,#1
```

e0	8000 0012 8032 57FA
e0	8000 001E 803C 57FA

```
madd.s q e0,e2,d4,d5l,#1
```

This instruction also exists with saturation (See 16-bit Multiply-Add section):

```
madds.q e0,e2,d4,d5u,n
```

```
madds.q e0,e2,d4,d5l,n
```

Saturation is 0x7FFFFFFF FFFFFFFF for positive results and 0x80000000 00000000 for negative results.

Example:

e2	7FFF FFEE 8014 57FA
d4	6000 5000

d5	0030 0040
----	-----------

After Instruction

madds.q e0,e2,d4,d5u,#1

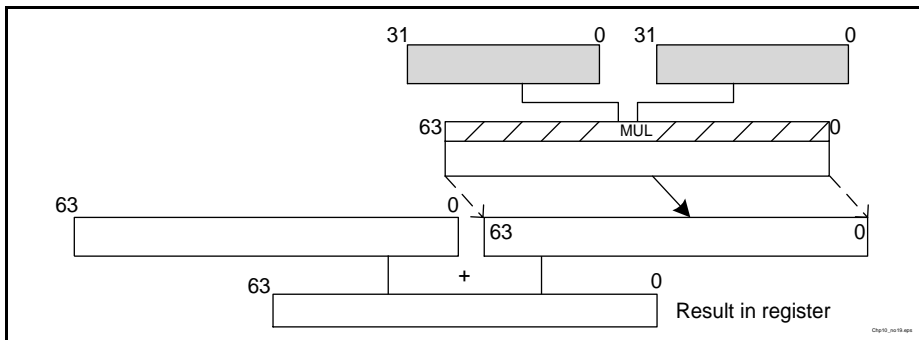
e0	7FFF FFFF FFFF FFFF
e0	7FFF FFFF FFFF FFFF

madds.q e0,e2,d4,d5l,#1

10.3.6 64 + 32*32 Multiply-Add (MADD.Q, MADDS.Q)

This operation is TriCore MAC with maximum precision.

madd.q e0,e2,d4,d5,n



OVF: The result of the addition cannot be represented in a 64-bit register.

Example:

e2	7FFF FFEE 8014 57FA
d4	6000 5000
d5	0030 0040

After Instruction

madd.q e0,e2,d4,d5,#1

e0	8024 003C 803C 57FA
----	---------------------

This instruction also exists with saturation:

```
madds.q e0, e2, d4, d5, n
```

Example:

e2	7FFF FFEE 8014 57FA
d4	6000 5000
d5	0030 0040

After Instruction

```
madds.q e0, e2, d4, d5u, #1
```

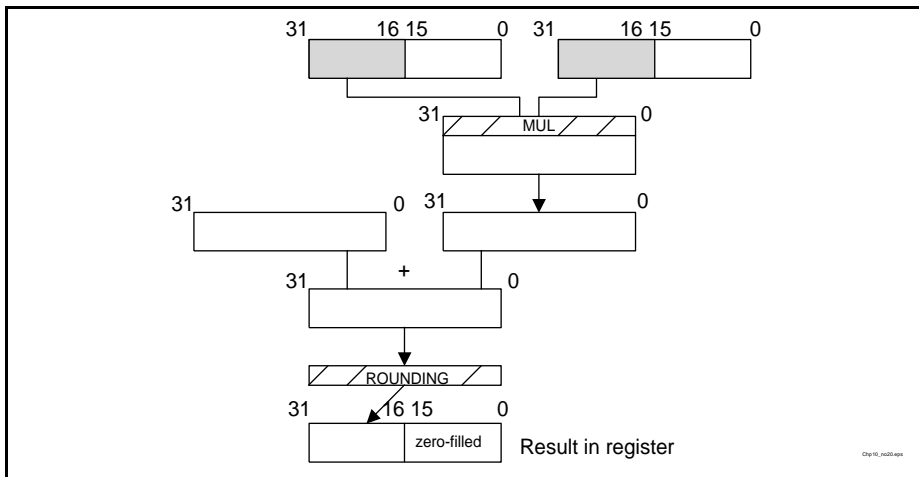
e0	7FFF FFFF FFFF FFFF
----	---------------------

10.3.7 32 + 16*16 ➡ Round(32) Multiply-Add (MADDR.Q, MADDRS.Q)

Rounding is executed after the addition.

```
maddr.q d0, d1, d4u, d5u, n
```

```
maddr.q d0, d1, d4l, d5l, n
```



OVF: The result of the addition cannot be represented in a 32-bit register.

This flag is always computed after, because rounding a value can lead to an overflow.

This instruction also exists with saturation:

```
maddrs.q d0, d1, d4u, d5u, n
```

```
maddrs.q d0, d1, d4l, d5l, n
```

Signed saturation is 0x7FFF0000 for positive results and 0x80000000 for negative results.

Example:

d1	7FFF 1234
d4	6000 5000
d5	0030 0040

After Instruction

maddr.q d0,d1,d4u,d5u,#1

d0	8023 0000
d3	8027 0000
d0	7FFF 0000
d3	7FFF 0000

maddr.q d3,d1,d4l,d5l,#1

maddrs.q d0,d1,d4u,d5u,#1

maddrs.q d3,d1,d4l,d5l,#1

10.4 Multiply-Subtract (MSUB.Q, MSUBR.Q)

All MADD instructions have an MSUB counterpart (see Summary tables at the end of this chapter). Their behavior is identical except that:

- Subtraction replaces addition
- Overflow flags have different logic equations (but an identical definition)

Example:

To solve these two equations:

$$sZ = sX + (sY * sK)$$

$$sZ = sX - (sY * sK)$$

madd.q d0,d4,d2u,d3u,#1

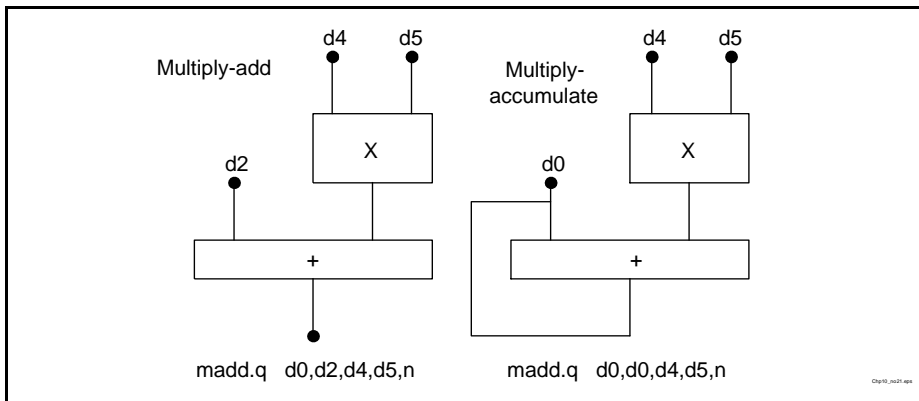
msub.q d1,d4,d2u,d3u,#1

10.5 Multiply-Accumulate

The multiply-add operation has three inputs and one output.

The multiply-accumulate operation has two inputs and accumulates the result.

Multiply-accumulate is implemented with a MADD instruction where one source and destination are identical.



Note that the same notation is applied to all the different cases. When the upper half is used, it is shown with “u” and when using the lower half, with an “l” after the registers. The use of the MADD.Q instruction remains the same.

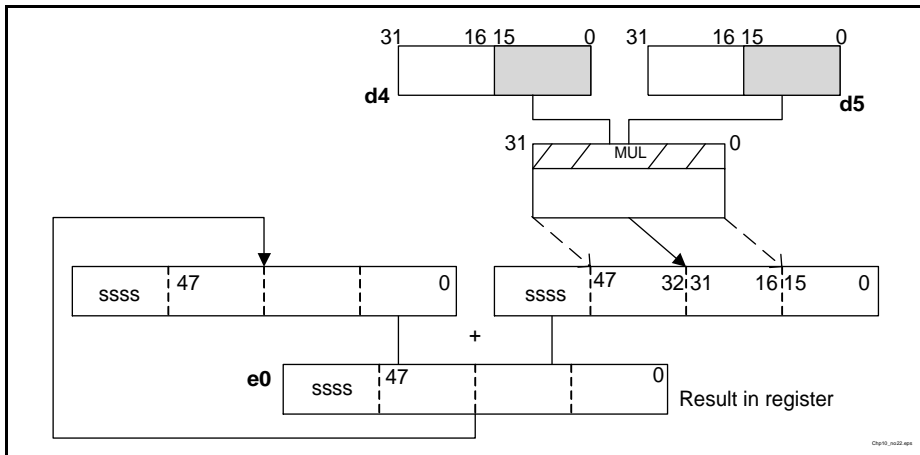
In Multi-precision on 16-bit multiply-add section, the value that is added must have 16 bits of guard bits to give a coherent result. In the following example, the accumulation is initialized to zero so there is no such problem. But if the first value is different from zero, this value must have 16 bits of guard bits.

Example:

```

mov      d0,#0                ;clear d0
mov      d1,#0                ;clear d1
ld.h     d4,[a2+]2            ;load first value
mov      d5,#0x0666           ;init d5
lea      a2,coef              ;init pointer on halfword value
lea      a4,(8-1)             ;init counter
aloop:   madd.q    e0,e0,d4l,d5l,#1
          ld.h     d4,[a2+]2
          loop     a4,aloop
nop

```



10.6 Applications

To illustrate the use of single MAC operations, 2 transcendental functions have been selected; square root and sine. There is 1 square root example, but several examples of sine are developed to illustrate the different cases of precision (16-bit, 16-bit rounded, 32-bit, q14 or q15, saturated or not).

The sine value function can be implemented as a table look-up (space) or as a series expansion (time). It can also be implemented as a combination of space and time (partial look-up table and partial computation). This illustrates the first law of algorithms, that the space-time continuum is constant. The sine value function also requires different precision levels, depending on the application. Telecom application requires less than 16 bits, whereas audio requires more than 20 bits.

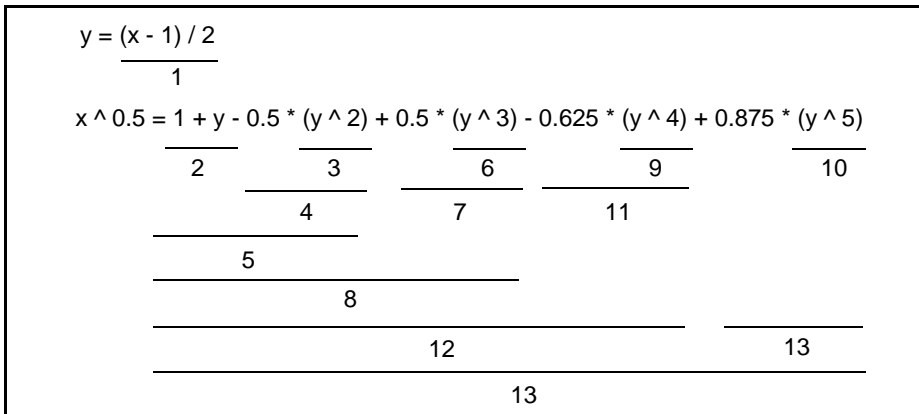
10.6.1 Square Root

Input = $[+0.5..+1)$ in 1Q15.

Output in 1Q15.

For $x \in [0.5 ; 1)$, the square root can be calculated with a six-term Taylor equation.

The order of execution is shown in the diagram and Example which follow.



Example:

After Instruction

```
ld.w  d1, lx7
mov   d4, #0
movh  d10, #0x8000
shas  d0, d1, #-1
movh  d5, #0x4000
subs  d0, d0, d5
subs  d1, d0, d10
msubs.q d2, d4, d0, d0, #1
shas  d3, d2, #-1
adds  d3, d1, d3
msubs.q d2, d4, d2, d0, #1
shas  d6, d2, #-1
adds  d3, d3, d6
mul.q d2, d2, d0, #1
```

d1	0400 0000
d4	0000 0000
d10	8000 0000
d0	2000 0000
d5	4000 0000
d0	e000 0000
d1	6000 0000
d2	f800 0000
d3	fc00 0000
d3	5c00 0000
d2	fe00 0000
d6	ff00 0000
d3	5b00 0000
d2	0080 0000

```

msubs.q  d6,d4,d2,d0,#1
movh    d5,#0x5000
msubs.q  d7,d4,d5,d2,#1
adds    d3,d3,d7
movh    d5,#0x7000
msubs.q  d3,d3,d5,d6,#1

```

d6	0020 0000
d5	5000 0000
d7	ffb0 0000
d3	5ad0 0000
d5	7000 0000
d3	5a94 0000

10.6.2 Sine (coefficient mixed 1q31 and 3q29, result in 1q31)

Input = [-1,1) in 1Q15

Output = [-1,1) in 1Q31

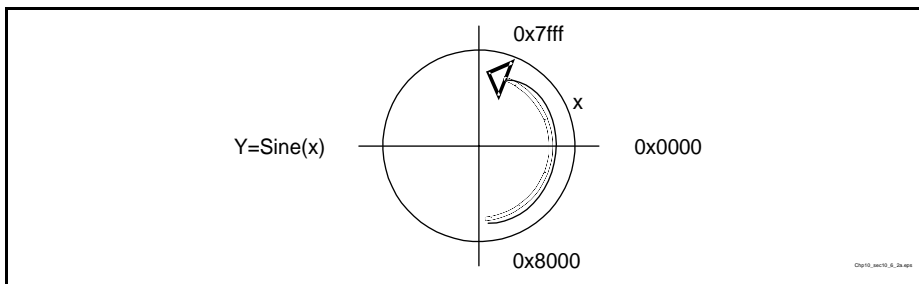
The series of Taylor gives:

$$\sin(x) = k_1 \cdot x + k_2 \cdot x^3 + k_3 \cdot x^5 + k_4 \cdot x^7 + k_5 \cdot x^9 + \dots$$

This can also be written as:

$$\sin(x) = (((((k_5 \cdot x^2 + k_4) \cdot x^2 + k_3) \cdot x^2 + k_2) \cdot x^2 + k_1) \cdot x$$

This series is valid for $x \in [-\pi/2, \pi/2]$, so the input between [-1,+1) is scaled to the range $[-\pi/2, \pi/2)$ and gives a result between -1 and 1.



Example:

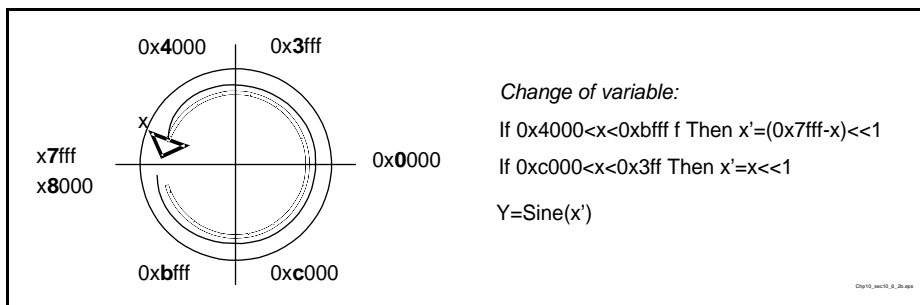
```

lea      a2,mycoef      ;load the address of the first coeff
lea      a3,0x04        ;initialize the counter
ld.q     d4,sX0         ;load the number we would like the sine 1Q15
ld.w     d2,1X8         ;load the factor of norm (PI/2) 2Q30
mul.q    d4,d4,d2,#1    ;x = x*a 2q30
mul.q    d1,d4,d4,#1    ;z = x*x, 3q29
ld.w     d2,[a2+]4      ;load k51q31

```

```
ld.w    d8, [a2+]4           ;load k41q31
l1lloop: madds.q    d2,d8,d2,d1,#1 ;give the result in 3q29
        sh          d2,d2,#2     ;1q31
        ld.w        d8, [a2+]4   ;1q31
        loop        a3,l1lloop   ;(( (k5*z+k4)*z+k3) z+k2) z+k1
mul.q    d6,d2,d4,#1         ;(( (k5*z+k4)*z+k3) z+k2) z+k1) x 2q30
shas     d6,d6,#1            ;1q31
```

The same result can be achieved for the range $[-\pi, \pi]$ with: $\sin(\pi-x) = \sin(x)$.



The change of variable is, with x in d4:

Example:

```
movh     d9, #0x8000          ; -1
xor      d2, d4:31, d4:30     ; 0x4000 < x < 0xbfff ?
jz       d2, lab              ; if not, go to lab
add      d4, d9, d4            ; else x = x-1
rsub     d4, d4, #0            ; x = -(x-1) = 1-x
lab:     sh d4, #1              ; x' = x << 1
```

For all following algorithms, the sine is in a range of $[-\pi/2.. \pi/2]$. The input should be in 1q15.

10.6.3 Sine of a Value (coefficient in 2q14, result in 1q31)

```
lea    a2,coeffq           ;load the address of the first coeff
lea    a3,0x03             ;initialize the counter
ld.q   d4,sX0              ;load the number we would like the sine
mul.q  d1,d4,d4,#1         ;z = x*x, 1q15
ld.q   d2,[a2+]4           ;load k52q14
shas   d1,d1,#-1           ;result in 2q14
ld.q   d3,[a2+]4           ;load k43q13
iloop5: madd.q  d2,d3,d2,d1,#1 ;give the result in 3q13
        ld.q    d3,[a2+]4     ;3q13
        sh      d2,d2,#1      ;result in 2q14
loop   a3,iloop5           ;(((k5*z+k4)*z+k3)z+k2)z+k1
mul.q  d6,d2,d4,#1         ;(((k5*z+k4)*z+k3)z+k2)z+k1)x
shas   d6,d6,#1
```

10.6.4 Sine of a Value (coefficient in 1q15 Mixed with 2q14, result in 1q31)

```
lea    a2,_coeffq         ;load the address of the first coeff
lea    a3,0x02            ;initialize the counter
ld.q   d4,sX0             ;load the number we would like the sine
mul.q  d1,d4,d4,#1        ;z = x*x 1q15
ld.q   d2,[a2+]4          ;load k5 1q15
ld.q   d3,[a2+]4          ;load k4 1q15
iloop4: madd.q  d2,d3,d2,d1,#1 ;
        ld.q    d3,[a2+]4     ;1q15
loop   a3,iloop4          ;((k5*z+k4)*z+k3)z+k2 1q15
madd.q d2,d3,d2,d1,#0     ;(((k5*z+k4)*z+k3)z+k2)z+k1,result in 2q14
mul.q  d6,d2,d4,#1        ;(((k5*z+k4)*z+k3)z+k2)z+k1)z
shas   d6,d6,#1
```

10.6.5 Sine of a Value (coefficient in 1q15, result in 1q31)

```
lea    a2,__coeff         ;load the address of the first coeff
lea    a3,0x03            ;initialize the counter
ld.q   d4,sX0             ;load the number we would like the sine
mul.q  d1,d4,d4,#1        ;z = x*x
ld.q   d2,[a2+]4          ;load k5
ld.q   d3,[a2+]4          ;load k4
iloop3: madd.q  d2,d3,d2,d1,#1 ;
        ld.q    d3,[a2+]4     ;
loop   a3,iloop3          ;(((k5*z+k4)*z+k3)z+k2)z+k'1
madd.q d6,d4,d2,d4,#1     ;(((k5*z+k4)*z+k3)z+k2)z+k'1)x+x
```

10.6.6 Sine of a Value (coefficient in 2q14, result in 1q31 with saturation)

```

lea    a2,coefq1           ;load the address of the first coeff
lea    a3,0x03             ;initialize the counter
ld.q   d4,sX0              ;load the number we would like the sine
mul.q  d1,d4,d4,#1         ;z = x*x, 1q15
ld.q   d2,[a2+]4           ;load k5 2q14
sha    d1,d1,#-1           ;result in 2q14
ld.q   d3,[a2+]4           ;load k4 3q13
iloop6: madds.q d2,d3,d2,d1,#1 ;give the result in 3q13
        ld.q     d3,[a2+]4     ;3q13
        shas     d2,d2,#1      ;result in 2q14
loop    a3,iloop6           ;(((k5*z+k4)*z+k3)z+k2)z+k1
mul.q   d6,d2,d4,#1         ;((((k5*z+k4)*z+k3)z+k2)z+k1)x
shas    d6,d6,#1

```

10.6.7 Sine of a Value (coefficient in 1q15, result in 1q31 with saturation)

```

lea    a2,__coeff1         ;load the address of the first coeff
lea    a3,0x03             ;initialize the counter
ld.q   d4,sX0              ;load the number we would like the sine
mul.q  d1,d4,d4,#1         ;z = x*x
ld.q   d2,[a2+]4           ;load k5
ld.q   d3,[a2+]4           ;load k4
iloop8: madds.q d2,d3,d2,d1,#1 ;
        ld.q     d3,[a2+]4     ;
loop    a3,iloop8           ;(((k5*z+k4)*z+k3)z+k2)z+k'1
madds.q d6,d4,d2,d4,#1     ;((((k5*z+k4)*z+k3)z+k2)z+k'1)x+x

```

10.6.8 Sine of a Value (coefficient in 1q15, result in 1q15 with truncation)

```

lea    a2,__coeff3         ;load the address of the first coeff
lea    a3,0x03             ;initialize the counter
ld.q   d4,sX0              ;load the number we would like the sine
mul.q  d1,d4,d4,#1         ;z = x*x
ld.q   d2,[a2+]4           ;load k5
ld.q   d3,[a2+]4           ;load k4
ilp3:  madd.q   d2,d3,d2,d1,#1 ;
        ld.q     d3,[a2+]4     ;
loop    a3,ilp3           ;(((k5*z+k4)*z+k3)z+k2)z+k'1
madd.q  d6,d4,d2,d4,#1     ;((((k5*z+k4)*z+k3)z+k2)z+k'1)x+x

```

10.6.9 Sine of a Value (coefficient in 1q15, result in 1q15 with rounding)

```
lea    a2, __coeff           ;load the address of the first coeff
lea    a3, 0x03              ;initialize the counter
ld.q   d4, sX0               ;load the number we would like the sine
mul.q  d1, d4, d4, #1        ;z = x*x
ld.q   d2, [a2+]4            ;load k5
ld.q   d3, [a2+]4            ;load k4
ilop3: madd.q  d2, d3, d2, d1, #1 ;
      ld.q    d3, [a2+]4        ;
loop   a3, ilop3              ;(((k5*z+k4)*z+k3)z+k2)z+k'1
maddr.q d6, d4, d2u, d4u, #1 ;(((k5*z+k4)*z+k3)z+k2)z+k'1)x+x
```

10.6.10 Sine of a Value (coefficient in 1q15, result 1q15 with rounding)

The values and result are in 1q15 format with rounding.

```
lea    a2, __coe             ;load the address of the first coeff
lea    a3, 0x03              ;initialize the counter
ld.q   d4, sX0               ;load the number we would like the sine
mulr.q d1, d4u, d4u, #1      ;z = x*x
ld.q   d2, [a2+]4            ;load k5
ld.q   d3, [a2+]4            ;load k4
il3:   maddr.q  d2, d3, d2u, d1u, #1 ;
      ld.q    d3, [a2+]4        ;
loop   a3, il3               ;(((k5*z+k4)*z+k3)z+k2)z+k'1
maddr.q d6, d4, d2u, d4u, #1 ;(((k5*z+k4)*z+k3)z+k2)z+k'1)x
```

10.6.11 Sine of a Value: Comparing Precision

Normal Result

As the sine algorithm uses constants in the $-1..2$ range (because of $+1$), three possibilities for the constants have been tried:

- All constants in Q15
- Constants in Q15 and Q14 for the constant which is bigger than 1 (mixed constant case)
- Constants in Q14 only

The mixed constant case result is less precise than the Q15 result. This is normal since one bit is lost when the result is converted from Q15 to Q14.

The Q14 result is less precise than the two other cases simply because Q14 constants are less precise than Q15 constants (See Q-format section). In conclusion, calculation in Q15 is more precise than the calculation with mixed q-format, which is more precise than calculation with Q14.

Saturated Result

To use saturation, the value out of range during the calculation and its impact on the final result must be known. Saturation will give a coherent result. For example, in the square root or invert algorithms saturation may be used. If not, a non-saturated result can be taken as negative value in the next calculation and will produce an incorrect result.

Truncated and Rounded Results

Truncation and rounding always produce a 16-bit result. Truncation keeps the 16 Most Significant Bits (MSBs) without looking at the lower half, whereas rounding will keep the 16 MSBs after checking the lower half. The value is more precise with rounding than with truncation.

10.7 Summary Tables

Multiplication

Result Size	Result format with n=1	Operation	Syntax
16-bit (rounding)	1q15z16 in 32-bit register	16x16	mulr.q d0,d1u,d2u,n mulr.q d0,d1l,d2l,n
32-bit	1q31 in 32-bit register	16x16	mul.q d0,d1u,d2u,n mul.q d0,d1l,d2l,n
		16x32	mul.q d0,d1,d2u,n mul.q d0,d1,d2l,n
		32x32	mul.q d0,d1,d2,n
48-bit	17q47 in 64-bit register	16x32	mul.q e0,d1,d2u,n mul.q e0,d1,d2l,n
64-bit	1q63 in 64-bit register	32x32	mul.q e0,d1,d2,n

Continued

Result size	Result format with n=1	Multi-plication	MADD.Q	MSUB.Q
16-bit (rounding)	1q15z16 in 32-bit reg.	16x16	maddr.q d0,d1,d4u,d5u,n maddr.q d0,d1,d4l,d5l,n	msubr.q d0,d1,d4u,d5u,n msubr.q d0,d1,d4l,d5l,n
16-bit (rounding, saturation)		16x16	maddrs.q d0,d1,d4u,d5u,n maddrs.q d0,d1,d4l,d5l,n	msubrs.q d0,d1,d4u,d5u,n msubrs.q d0,d1,d4l,d5l,n
32-bit	1q31 in 32-bit reg.	16x16	madd.q d0,d1,d4u,d5u,n madd.q d0,d1,d4l,d5l,n	msub.q d0,d1,d4u,d5u,n msub.q d0,d1,d4l,d5l,n
		16x32	madd.q d0,d1,d4,d5u,n madd.q d0,d1,d4,d5l,n	msub.q d0,d1,d4,d5u,n msub.q d0,d1,d4,d5l,n
		32x32	madd.q d0,d1,d4,d5,n	msub.q d0,d1,d4,d5,n
32-bit (saturation)		16x16	madds.q d0,d1,d4u,d5u,n madds.q d0,d1,d4l,d5l,n	msubs.q d0,d1,d4u,d5u,n msubs.q d0,d1,d4l,d5l,n
		16x32	madds.q d0,d1,d4,d5u,n madds.q d0,d1,d4,d5l,n	msubs.q d0,d1,d4,d5u,n msubs.q d0,d1,d4,d5l,n
		32x32	madds.q d0,d1,d4,d5,n	msubs.q d0,d1,d4,d5,n
48-bit	17q31z16 in 64-bit reg.	16x16	madd.q e0,e2,d4u,d5u,n madd.q e0,e2,d4l,d5l,n	msub.q e0,e2,d4u,d5u,n msub.q e0,e2,d4l,d5l,n
	17q47 in 64-bit reg.	16x32	madd.q e0,e2,d4,d5u,n madd.q e0,e2,d4,d5l,n	msub.q e0,e2,d4,d5u,n msub.q e0,e2,d4,d5l,n
48-bit (saturation)	17q31z16 in 64-bit reg.	16x16	madds.q e0,e2,d4u,d5u,n madds.q e0,e2,d4l,d5l,n	msubs.q e0,e2,d4u,d5u,n msubs.q e0,e2,d4l,d5l,n
	17q47 in 64-bit reg.	16x32	madds.q e0,e2,d4,d5u,n madds.q e0,e2,d4,d5l,n	msubs.q e0,e2,d4,d5u,n msubs.q e0,e2,d4,d5l,n
64-bit	1q63 in 64-bit reg.	32x32	madd.q e0,e2,d4,d5,n	msub.q e0,e2,d4,d5,n
64-bit (saturation)		32x32	madds.q e0,e2,d4,d5,n	msubs.q e0,e2,d4,d5,n

Note: 1q15z16 means 1 bit of sign, 15 bits of magnitude, 16 bits filled with zero.

17q47 means 17 bits of sign (effectively 16 guard bits), 47 bits of magnitude.

11 DSP and Dual MAC

Chapter 10 introduced basic notions on DSP operations, fractional arithmetic and trade-off in precision. This chapter will focus on the more sophisticated use of a Dual MAC.

The TriCore Dual MAC offers the ability to execute 2 multiplications or 2 multiplication/additions in 1 cycle, with 1 instruction. In addition, the Dual MAC offers a great deal of flexibility, such as:

- The multiplier source can be chosen between upper and lower parts (four register halves, combinations)
- Each MAC can independently choose addition or subtraction before giving the result.
- The 2 MAC operations can give independent results (2 results of 32-bit) or a merged result (1 48-bit single result).
- All the flexibility and features of the single MAC are available (choice for left alignment, operations with rounding, availability of saturation on all instructions).

Chapter contents:

- **Notation**
- **Dual MAC Structure**
- **Packed Dual-MAC Instructions (MUL, MADD)**
- **Merged/Packed MAC Instructions (MUL, MADD)**
- **Applications**
- **Summary Tables**
- **List of Instructions**

11.1 Notation

All instructions in this chapter have a specific notation. Two letters are placed after the last source register. These allow for the differentiation between the choices in the register halves. There are four possibilities:

ul lu ll uu

Where:

'u' means the upper half of the register is used.

'l' is for the lower part of the register is used.

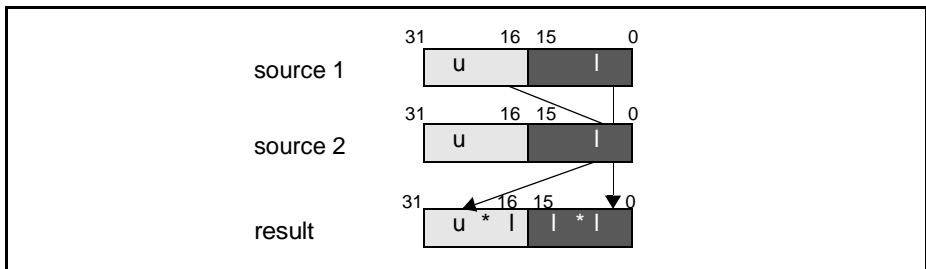
```
mul.h      dest,src1,src2xx,n
madd.h     dest,src1,src2,src3xx,n
```

The letters are written after the last source register ('xx'). Since the letters after the first source register are always fixed to 'ul', they are not explicitly written.

To compute the results using the letters:

	case 1		case 2		case 3		case 4	
letters of source 1	<i>u</i>	<i>l</i>	<i>u</i>	<i>l</i>	<i>u</i>	<i>l</i>	<i>u</i>	<i>l</i>
letters after source 2	<i>u</i>	<i>l</i>	<i>l</i>	<i>u</i>	<i>l</i>	<i>l</i>	<i>u</i>	<i>u</i>
multiplication result in:	upper	lower	upper	lower	upper	lower	<u>lower</u>	upper

Example Instruction used with the letters ll



Note: The position of the result in case 4 is computed differently to case 1, 2 and 3. The result of upper-upper multiplication ('uu' combination) is placed in the lower half. This configuration keeps real and imaginary numbers in the same order in calculation and in memory.

Multiplication of 2 complex:

before calculation	upper	lower
Register 1	imaginary	real
Register 2	imaginary	real
multiplication in configuration 'll'	lu	ll
multiplication in configuration 'uu'	ul	uu
add/sub	lu+ul	ll-uu
after calculation	imaginary	real

Example:

d3	upper1	lower1
d2	upper2	lower2

After Instruction

<code>mul.h e0,d3,d2 ll,#1</code>	e0	u_1l_2	l_1l_2
<code>maddsu.h e2,e0,d3,d2 uu,#1</code>	e2	$u_1l_2 + l_1u_2$	$l_1l_2 - u_1u_2$

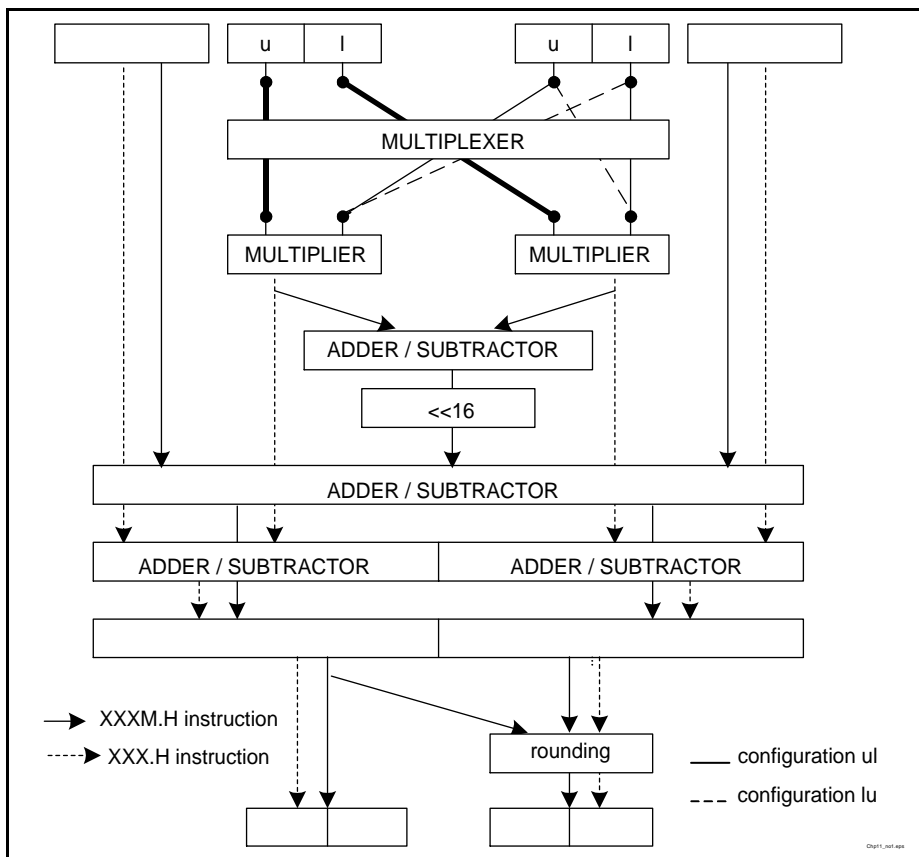
Note: The 'll' and 'uu' configurations avoid applying duplication since the multiplications use the same half of the register.

Each syntax proposes to left-align the result or not, as shown with the 'n' in the syntax. 'n' can be equal to 0 if the result is not left aligned, or 1 if it is left aligned. It will or will not left-align on both results at the same time.

11.2 Dual MAC Structure

11.2.1 Block Diagram

The TriCore dual-MAC is composed of two multipliers and three adder/subtractor's:



This structure, in theory, allows calculation to be performed twice as fast as single MAC operations. In practice however this will only be true if:

- The MAC operations can be made parallel. This is the case of many DSP algorithms, but not true if the algorithm is iterative.
- The algorithm result does not depend on a sequence order. Executing 2 accumulations in parallel and adding the result at the end, may not give the same 'bit exact' result as executing a single accumulation. This happens if an intermediate accumulation has been saturated.
- The 16-bit data must be packed.

To illustrate this final point, below is a comparison of MUL.Q and MUL.H instructions.

MUL.Q	MUL.H
mul.q d0,d4u,d5u,#1	mul.h e0,d4,d5ul,#1
mul.q d6,d7u,d8u,#1	

Although the actions look identical and both produce 2 multiplication results, they are different:

- MUL.Q has 4 sources in 4 different registers (d4, d5, d7, d8)
- MUL.H has 4 sources in 2 different registers (d4 and d5 only)

To use only 1 instruction for two multiplications, the source values must be packed in one register. If this constraint is met, two MUL.Q instructions can be replaced by one MUL.H instruction, as shown in the table.

MUL.Q	MUL.H
mul.q d0,d4u,d5u,#1	mul.h e0,d4,d5ul,#1
mul.q d1,d4l,d5l,#1	

11.2.2 Differences between Packed & Merged/Packed

The difference between a packed operation (described in [Section 11.3](#)) and a merged operation (described in [Section 11.4](#)), is the number of results produced.

A packed operation (commonly called SIMD), gives 2 results. A merged operation gives a single result.

For example:

$Y = Y + X * K$; $Z = Z + W * B$ (packed)

$Y = Y + X * K + W * B$ (merged packed)

However, MAC merged instructions do not exist simply to produce a single result rather than 2 results. The reason for MAC merged instructions is that a packed instruction gives 2 results of 32-bit and this is not always adequate. A user might prefer only 1 result, but with a greater register size. This is the case where merged instructions are used.

Therefore, by reviewing the 2 instructions in the example above, with a shorthand prefix to indicate the data size, it is easier for the programmer to see the differences:

$IY = IY + sX * sK$; $IZ = IZ + sW * sB$ (packed)

$eY = eY + sX * sK + sW * sB$ (merged packed)

Where:

s = SHORT (16-bit register)

I = LONG (32-bit register)

e = extended (48-bit value in 64-bit register) (refer to explanation in relevant section).

11.3 Packed Dual-MAC Instructions (MUL, MADD)

This section describes all packed dual-MAC instructions:

- Packed multiplication: MUL.H
- Packed multiplication followed by addition or subtraction: MADD(S), MSUB(S).H, MADDSU(S).H, MSUBAD(S).H
- Packed multiplication with rounding: MULR.H
- Packed multiplication followed by addition or subtraction and rounding: MADDR(S), MSUBR(S).H, MADDSUB(S).H, MSUBADR(S).H

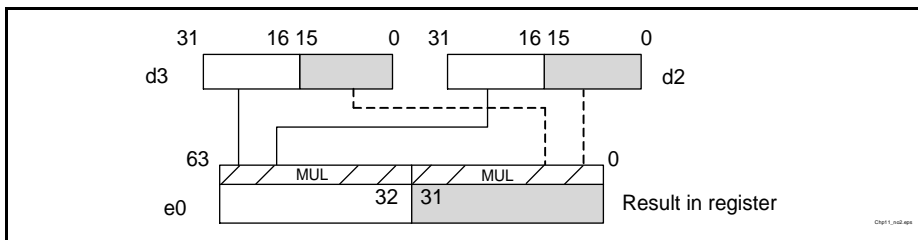
11.3.1 Packed Multiply (MUL.H)

Two 16*16 multiplications are computed simultaneously, producing two 32-bit results.

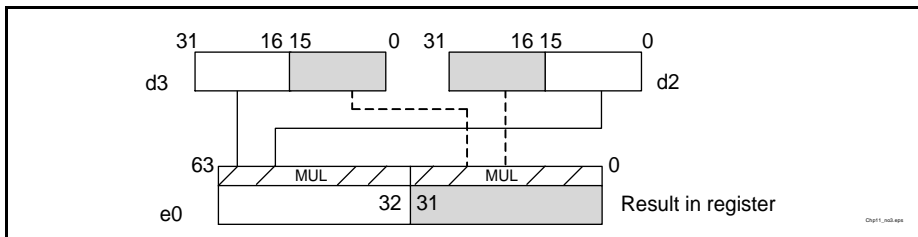
MUL.H uses two source registers and the two results are in an extended register.

- Each instruction can have the four source variants: 'ul', 'lu', 'll' and 'uu' (See the diagrams which follow below).
- Left-alignment can be chosen.
- There is no saturated version as overflow is impossible.

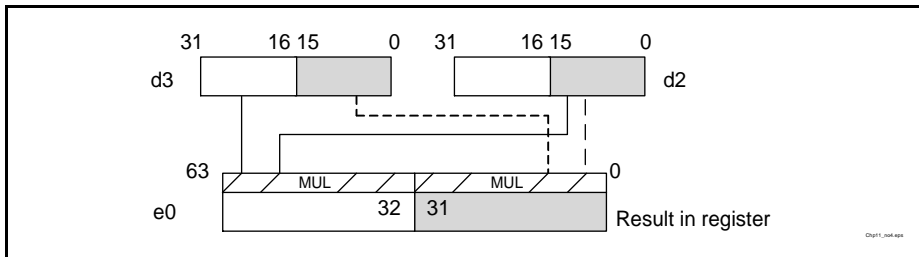
```
mul.h e0, d3, d2ul, n
```



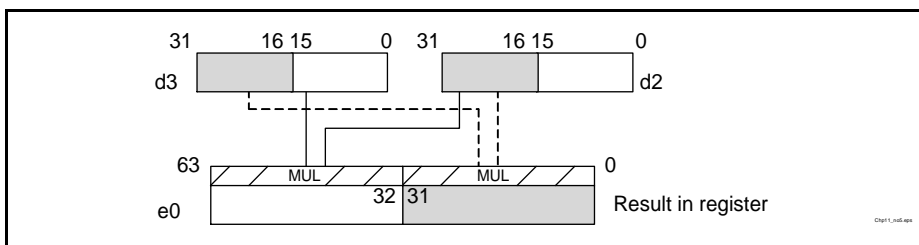
```
mul.h e0, d3, d2lu, n
```




```
mul.h e0,d3,d2ll,n
```



```
mul.h e0,d3,d2uu,n
```



Example:

d5	7870 0090
d4	6430 0040

After Instruction

```
mul.h e0,d5,d4ul,#1
mul.h e0,d5,d4lu,#1
mul.h e0,d5,d4ll,#1
mul.h e0,d5,d4uu,#1
```

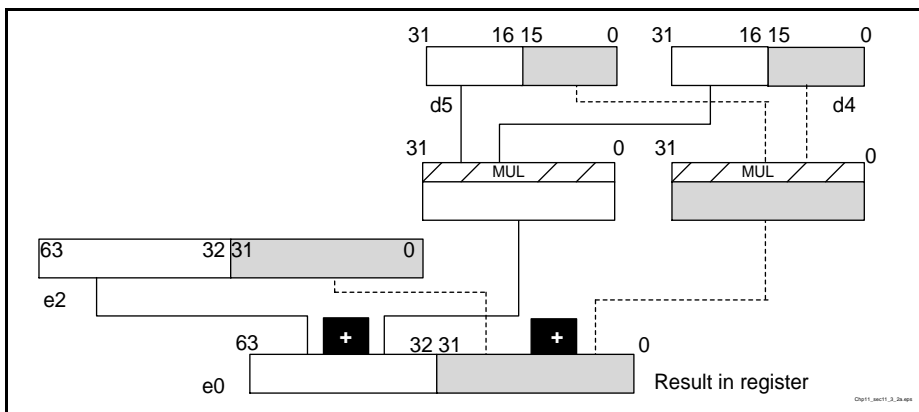
e0	5E44 AA00 0000 4800
e0	003C 3800 0070 B600
e0	003C 3800 0000 4800
e0	0070 B600 5E44 AA00

11.3.2 Packed Multiply-Add (MADD(S).H)

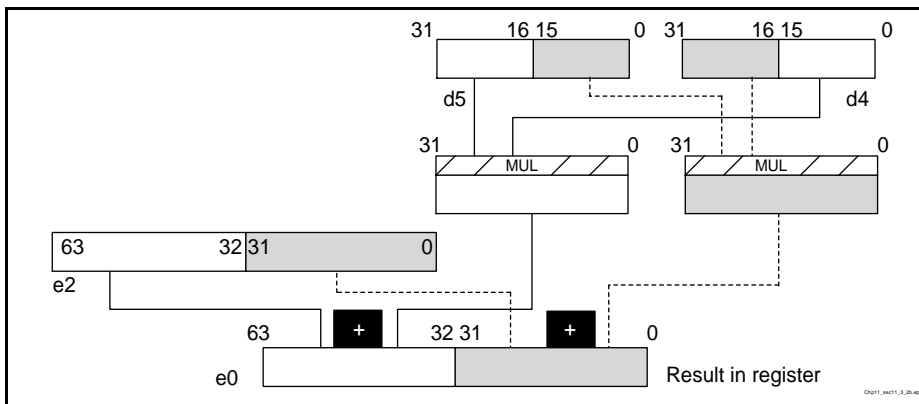
Two 16*16 multiplications are computed simultaneously. The intermediate products are added to a 32-bit value (typically a previous accumulated result). This instruction uses four source registers (two for the multiplication and two for the addition). The results are in an extended register.

- Each instruction can have the usual 4 source variants: 'ul', 'lu', 'll' and 'uu'.
- Left-alignment can be chosen.
- Each instruction has a saturated version.

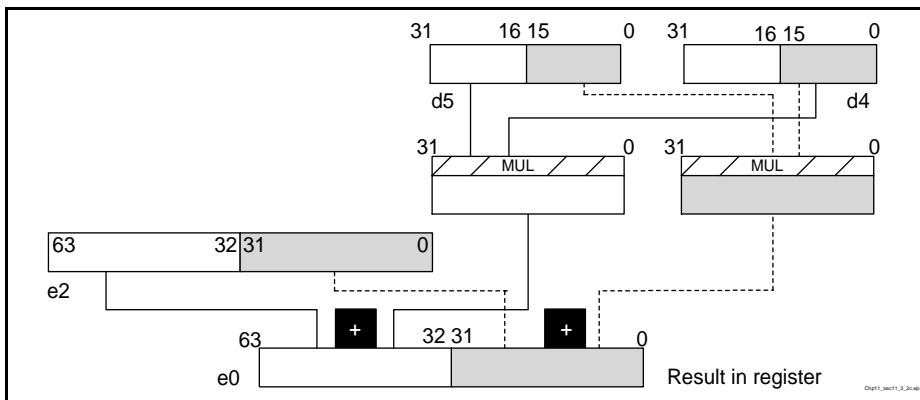
`madd.h e0,e2,d5,d4ul,n`



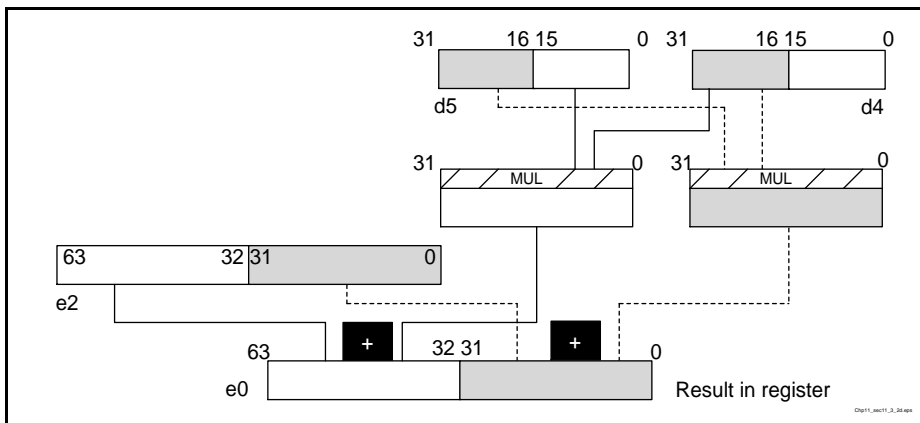
`madd.h e0,e2,d5,d4lu,n`



madd.h e0,e2,d5,d4ll,n



madd.h e0,e2,d5,d4uu,n



OVF: The result of the additions cannot be represented in a 32-bit register

The syntax for the 4 saturation instructions shown, is:

madds.h e6,e2,d5,d4ul,n

madds.h e6,e2,d5,d4lu,n

madds.h e6,e2,d5,d4ll,n

madds.h e6,e2,d5,d4uu,n

While the overflow flag is common to both MAC, the 2 saturations are activated separately (See [Packed Arithmetic](#) in [Section 1.11](#)).

Example:

d2	8001 2547
d3	6000 5000
d5	7870 0090
d4	6430 0040

After Instruction

madd.h	e0,e2,d5,d4ul,#1	e0	BE44 FA00 8001 6D47
madd.h	e0,e2,d5,d4lu,#1	e0	603C 8800 8071 DB47
madd.h	e0,e2,d5,d4ll,#1	e0	603C 8800 8001 6D47
madd.h	e0,e2,d5,d4uu,#1	e0	6071 0600 DE45 CF47
madds.h	e6,e2,d5,d4ul,#1	e6	7FFF FFFF 8001 6D47
madds.h	e6,e2,d5,d4lu,#1	e6	603C 8800 8071 DB47
madds.h	e6,e2,d5,d4ll,#1	e6	603C 8800 8001 6D47
madds.h	e6,e2,d5,d4uu,#1	e6	6071 0600 DE45 CF47

11.3.3 Packed Multiply-Subtract (MSUB(S).H)

Except for the subtraction, the MSUB(S).H instruction has the same behaviour and features as the MADD(S).H instruction (i.e. four variants, left-alignment and saturation)

11.3.4 Packed Multiply-Add/Subtract (MADDSU(S).H, MSUBAD(S).H)

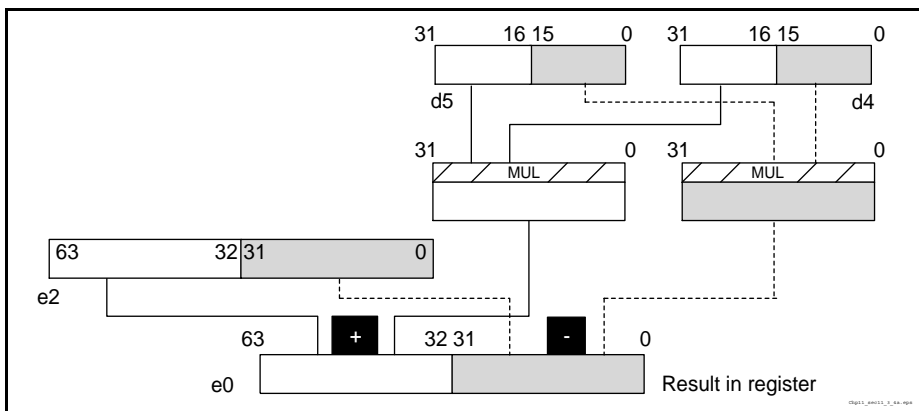
Each MAC can perform independent operations after multiplication. One MAC executes an addition and the second MAC executes a subtraction.

This instruction is called MADDSU if the upper half of the third source register executes the addition and lower half executes the subtraction (see diagrams which follow).

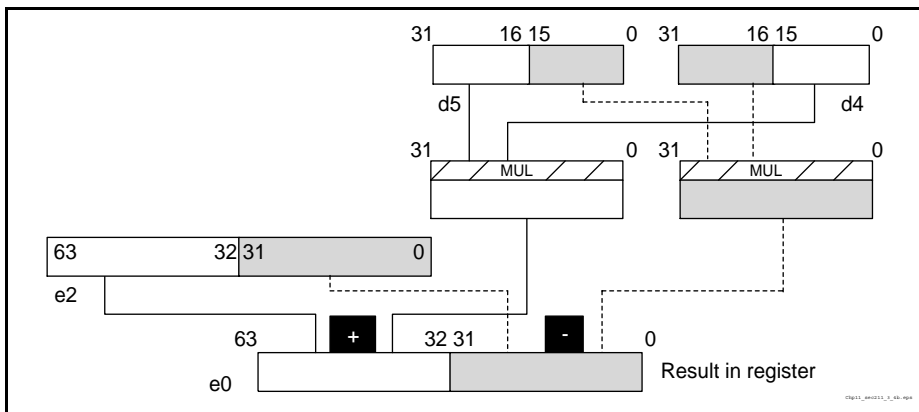
It is called MSUBAD if the upper half of the third source register executes the subtraction and the lower half executes the addition.

Except for subtraction, MADDSU(s).H and MSUBAD(s) both have the same behaviour and features (the four variants, left-alignment and saturation) as the MADD(S).H instruction. Here the four variants are shown for the MADDSU.H instruction.

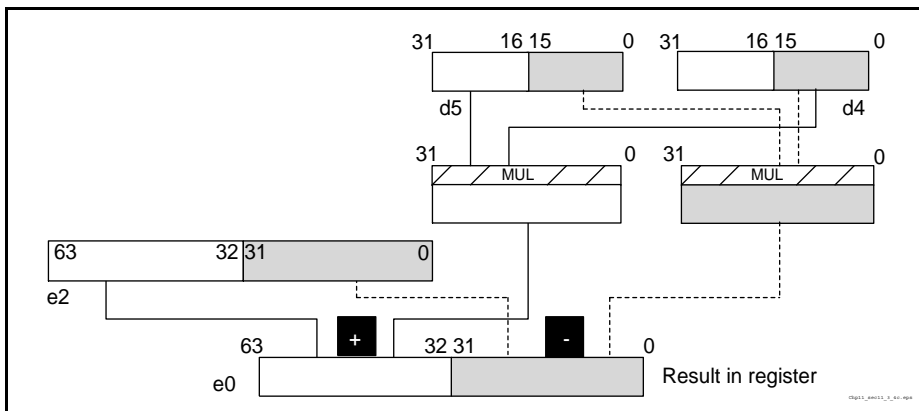
`maddsu.h e0,e2,d5,d4ul,n`



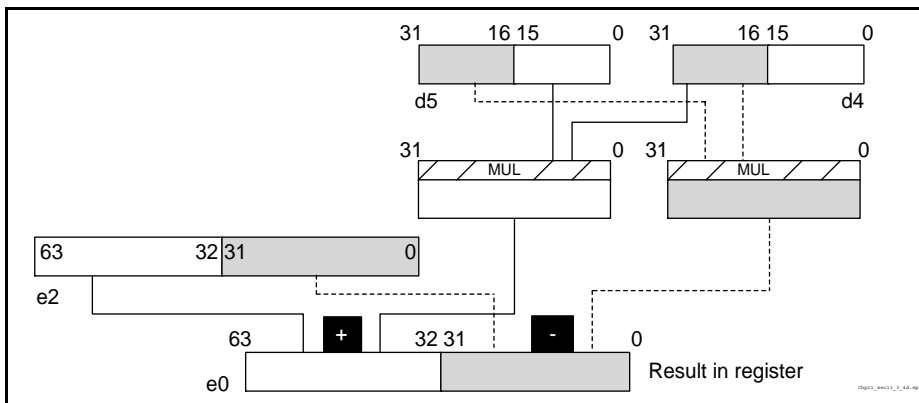
`maddsu.h e0,e2,d5,d4lu,n`



```
maddsu.h e0,e2,d5,d4ll,n
```



```
maddsu.h e0,e2,d5,d4uu,n
```



OVF: The result of the addition/subtraction can *NOT* be represented in a 32-bit register.

Saturation instructions exist for the four variants:

```
maddsus.h e6,e2,d5,d4ul,n
```

```
maddsus.h e6,e2,d5,d4lu,n
```

```
maddsus.h e6,e2,d5,d4ll,n
```

```
maddsus.h e6,e2,d5,d4uu,n
```

Saturation is activated separately.

Example:

d2	8001 2547
d3	6000 5000
d5	7870 0090
d4	6430 0040

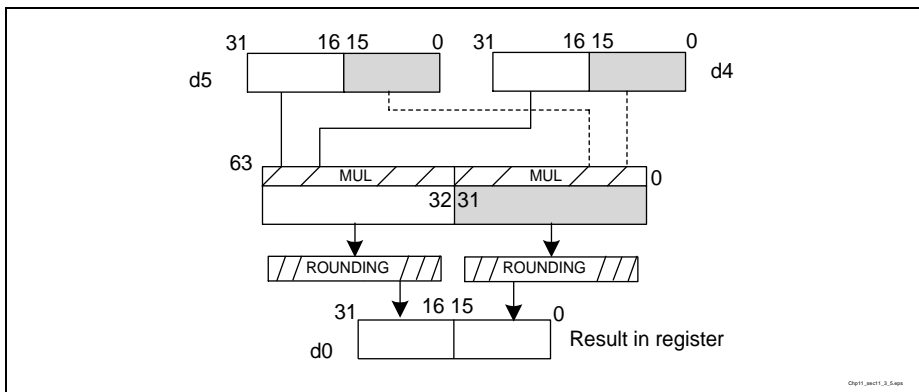
After Instruction

maddsu.h	e0,e2,d5,d4ul,#1	e0	BE44 FA00 8000 DD47
maddsu.h	e0,e2,d5,d4lu,#1	e0	603C 8800 7F90 6F47
maddsu.h	e0,e2,d5,d4ll,#1	e0	603C 8800 8000 DD47
maddsu.h	e0,e2,d5,d4uu,#1	e0	6071 0600 21BC 7B47
maddsus.h	e6,e2,d5,d4ul,#1	e6	7FFF FFFF 8000 DD47
maddsus.h	e6,e2,d5,d4lu,#1	e6	603C 8800 8000 0000
maddsus.h	e6,e2,d5,d4ll,#1	e6	603C 8800 8000 DD47
maddsus.h	e6,e2,d5,d4uu,#1	e6	6071 0600 8000 0000

11.3.5 Packed Multiply - Round (MULR.H)

Two 16*16 multiplications are computed simultaneously. Then 0x8000 is added to each 32-bit result, and each lower 16-bit is cleared. Since the 2 results each only have 16-bit format of precision, they are packed into a 32-bit destination. In this way the source and destination formats are equivalent (packed 16-bit). In addition, packing means that up to 4 results can be stored back to memory in one cycle.

`mulr.h d0,d5,d4ul,n`



- Each instruction can have 4 source variants: 'ul', 'lu', 'll' and 'uu'.
- Left-alignment can be selected.
- There is no saturated version as overflow is impossible.

Example:

d5	7870 0090
d4	6430 0040

After Instruction

<code>mulr.h d0,d5,d4ul,#1</code>	d0	5E45 0000
<code>mulr.h d0,d5,d4lu,#1</code>	d0	003C 0071
<code>mulr.h d0,d5,d4ll,#1</code>	d0	003C 0000
<code>mulr.h d0,d5,d4uu,#1</code>	d0	0071 5E45

11.3.6 Packed Multiply-Add-Round (MADDR(S).H)

This instruction is *NOT* equivalent to a MULR.H instruction followed by addition, but *IS* equivalent to MADD.H followed by rounding.

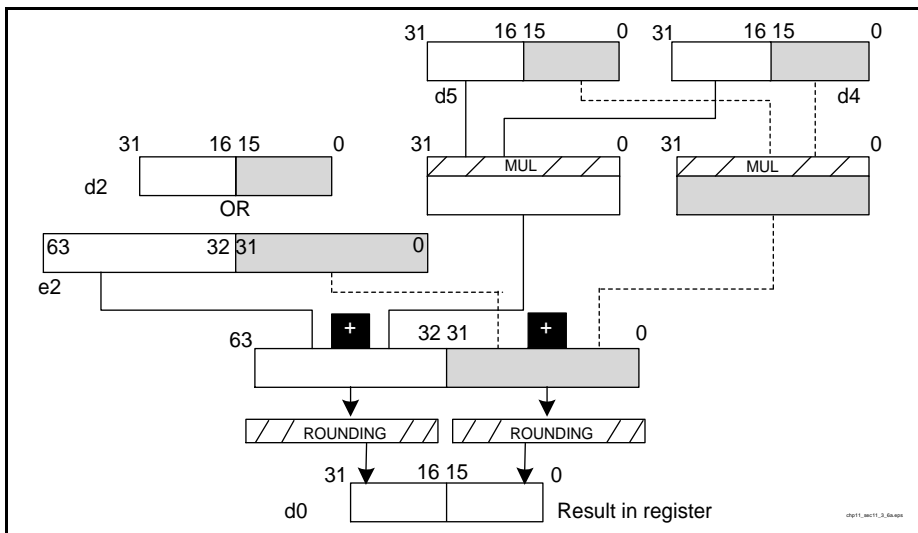
There are two syntax. The first uses a 32-bit register as source register for the addition (or subtraction) so it will add 16-bit values. The second syntax uses an extended register for the addition (or subtraction) so it will add 32-bit values.

```
maddr.h d0, d2, d5, d4ul, n
```

```
maddr.h d0, e2, d5, d4ul, n
```

Note: Rounding is executed before saturation. Rounding a value after accumulation can lead to an overflow.

- The first syntax can have the usual 4 source variants: 'ul', 'lu', 'll' and 'uu'. The second syntax only takes the 'ul' suffix.
- Left-alignment can be selected.
- Each instruction has a saturated version.



Note: Results are packed.

OVF: The result of the addition can *NOT* be represented in a 16-bit register.

Example:

d2	8001 2547
d3	6000 5000
d5	7870 0090
d4	6430 0040
d7	6000 8001

After Instruction

maddr.h d0,e2,d5,d4ul,#1	d0	BE45 8001
maddr.h d0,d7,d5,d4ul,#1	d0	BE45 8001
maddr.h d0,d7,d5,d4lu,#1	d0	603C 8072
maddr.h d0,d7,d5,d4ll,#1	d0	603C 8001
maddr.h d0,d7,d5,d4uu,#1	d0	6071 DE46
maddrs.h d6,e2,d5,d4ul,#1	d6	7FFF 8001
maddrs.h d6,d7,d5,d4ul,#1	d6	7FFF 8001
maddrs.h d6,d7,d5,d4lu,#1	d6	603C 8072
maddrs.h d6,d7,d5,d4ll,#1	d6	603C 8001
maddrs.h d6,d7,d5,d4uu,#1	d6	6071 DE46

Note: In the MADD.H instruction examples, one of the results is 0x603C8800. Rounding this value should result in 0x603D, but it does not. The result is 0x603C because the addition is with a 16-bit value (upper half of d7) instead of a 32-bit value (d3 in the examples of MADD.H instruction). The result is therefore less precise.

11.3.7 Packed Multiply-Subtract-Round (MSUBR(S).H)

Except for subtraction, the MSUBR(S).H instruction has the same behaviour and features as the MADDR(S).H instruction (i.e. 4 variants, left-alignment and saturation).

11.3.8 Packed Multiply-Add/Sub-Round (MADDSUR(s).H, MSUBADR(s).H)

Each MAC can perform independent operations after the multiplication. One MAC executes an addition and the second MAC executes a subtraction.

This instruction is called MADDSUR if the upper half of the third source register executes the addition and the lower half executes the subtraction.

It is called MSUBADR if the upper half of the third source register executes the subtraction and the lower half executes the addition.

Except for subtraction, MADDSU(s).H and MSUBAD(s) both have the same behaviour and features as the MADD(S).H instruction (i.e. 4 variants, left-alignment and saturation). Rounding the result of a MADDSU.H operation has the same effect as rounding the MADD.H instruction.

OVF: The result of the addition/subtraction cannot be represented in a 16-bit register.

Example:

d7	6000 8001
d5	7870 0090
d4	6430 0040

After Instruction

maddsur.h	d0, d7, d5, d4ul, #1	d0	BE45 8001
maddsur.h	d0, d7, d5, d4lu, #1	d0	603C 7F90
maddsur.h	d0, d7, d5, d4ll, #1	d0	603C 8001
maddsur.h	d0, d7, d5, d4uu, #1	d0	6071 21BC
maddsurs.h	d6, d7, d5, d4ul, #1	d6	7FFF 8001
maddsurs.h	d6, d7, d5, d4lu, #1	d6	603C 8000
maddsurs.h	d6, d7, d5, d4ll, #1	d6	603C 8001
maddsurs.h	d6, d7, d5, d4uu, #1	d6	6071 8000

11.4 Merged/Packed MAC Instructions (MUL, MADD)

This section describes the TriCore merged/packed MAC instructions:

- Merged packed multiplication: MULM(S).H
- Merged packed multiplication-addition/subtraction: MADDM(S).H, MSUBM(S).H, MADDSUM(S).H, MSUBADM(S).H

Note on 48-bit precision

Merged operations can also be called Multi-precision operations, because the results are larger than 32-bit.

A typical 'old-style' DSP will have 40-bit accumulators. This is not good for a standard DSP program where context switching becomes a bottleneck for the programmer. Modern DSPs and CPUs address this problem in several ways, and TriCore's solution is to use a 48-bit format. This has several advantages:

- It fits in a 64-bit register with the lower 16-bit unused. The hardware (muxes) already existed as part of 32*32 multiplications.
- 48-bit is more precise than 40-bit, is used in echo cancellation and has been 'standardised' in audio algorithms.
- The 48-bit data type (left aligned) can be handled in software as an imprecise version of extended long (64-bit precision).

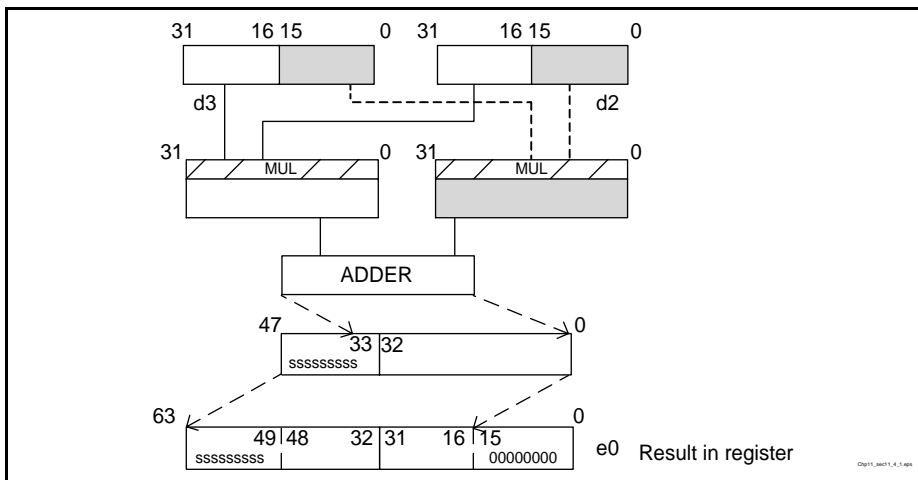
In summary, these instructions are better understood if they are seen as working with a 48-bit accumulator.

11.4.1 Merged Multiplication (MULM(S).H)

MULM.H performs two separate 16-bit multiplications and adds the 2 results into a 64-bit register.

After the two 16-bit multiplications, the two 32-bit results are added together using a 33rd bit to give a 33-bit result. This result is sign extended to 48-bit and then transferred to a 64-bit register with a left shift of 16, as shown in the following diagram:

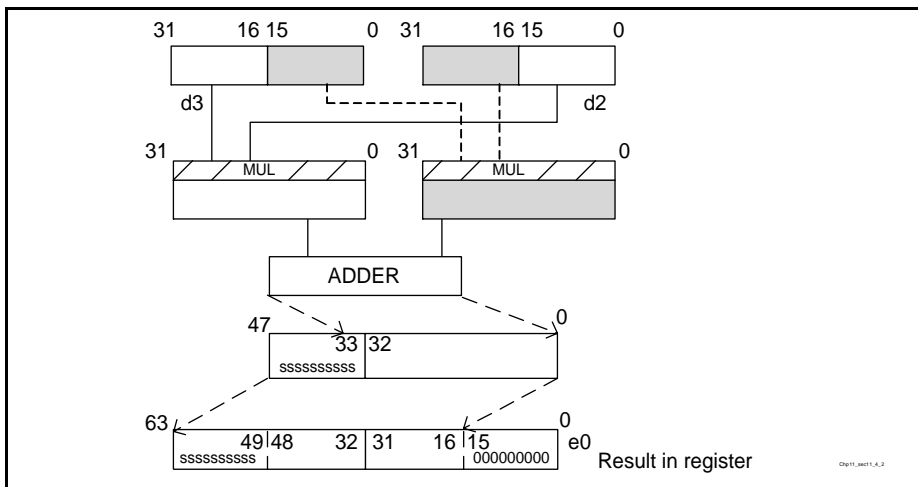
```
mulm.h e0,d3,d2u1,n
```



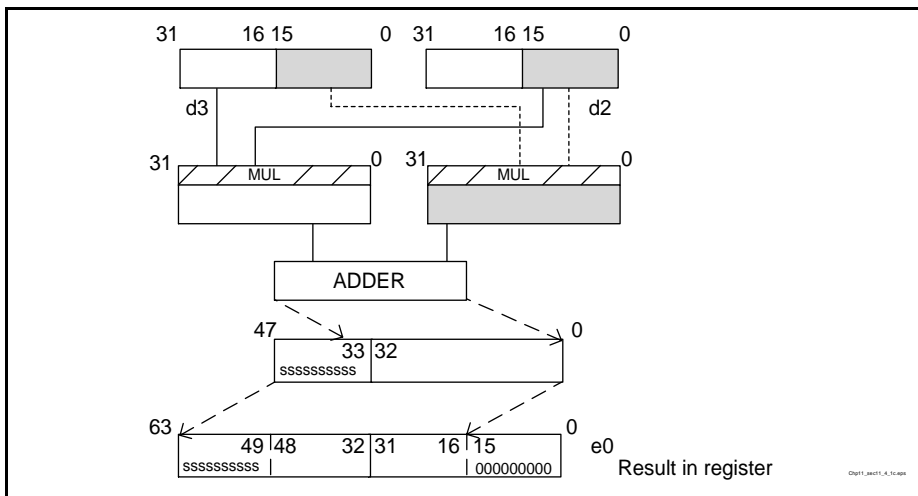
- Each instruction can have the 4 source variants: 'ul', 'lu', 'll' and 'uu'.
- Left-alignment can be selected. In this instance, instead of having 17 redundant signs, the result will have 16 redundant signs.
- Each instruction has a saturated version.

Note: In the current implementation the saturated version is strictly identical to the non-saturated one, and should not be used.

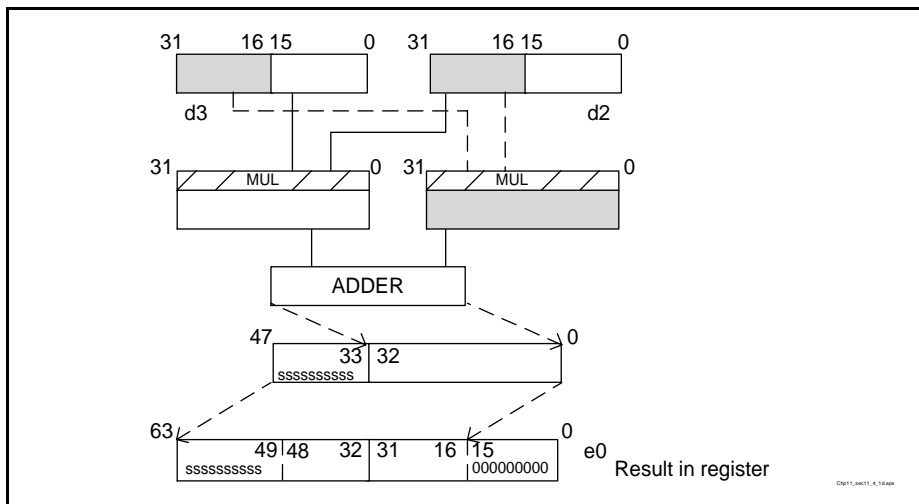
mulm.h e0,d3,d2lu,n



mulm.h e0,d3,d2ll,n



```
mulm.h e0,d3,d2uu,n
```



OVF: = 0; There can not be any overflow.

Example:

d5	4000 5000
d4	5000 6000

After Instruction

```
mulm.h e0,d5,d4ul,#1
```

```
mulm.h e0,d5,d4lu,#1
```

```
mulm.h e0,d5,d4ll,#1
```

```
mulm.h e0,d5,d4uu,#1
```

e0	0000 6400 0000 0000
e0	0000 6200 0000 0000
e0	0000 6C00 0000 0000
e0	0000 5A00 0000 0000

11.4.2 Merged Multiply-Add/Subtract (MADDM(S).H, MSUBM(S).H, MADDSUM(S).H, MSUBADM(S).H)

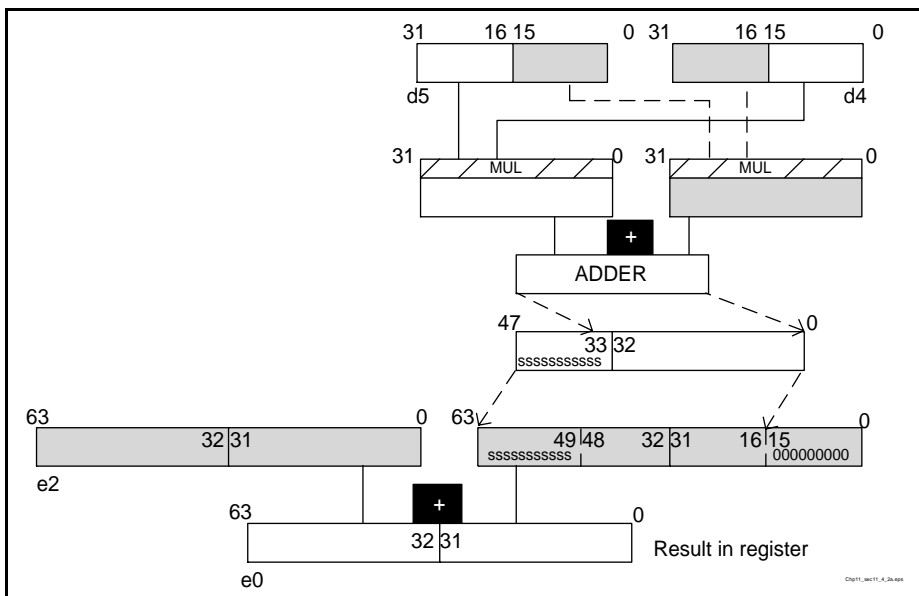
These instructions each perform two separate 16-bit multiplications. The 2 products of these multiplications are added (or subtracted), as an intermediate result. This result is then added (or subtracted) from the third operand, which is typically a 48-bit accumulation (a 64-bit register), and written into a 64-bit destination register.

- Each instruction can have the 4 source variants: 'ul', 'lu', 'll' and 'uu'.
- Left-alignment can be selected.
- Each instruction has a saturated version. Saturation is performed after the last operation.

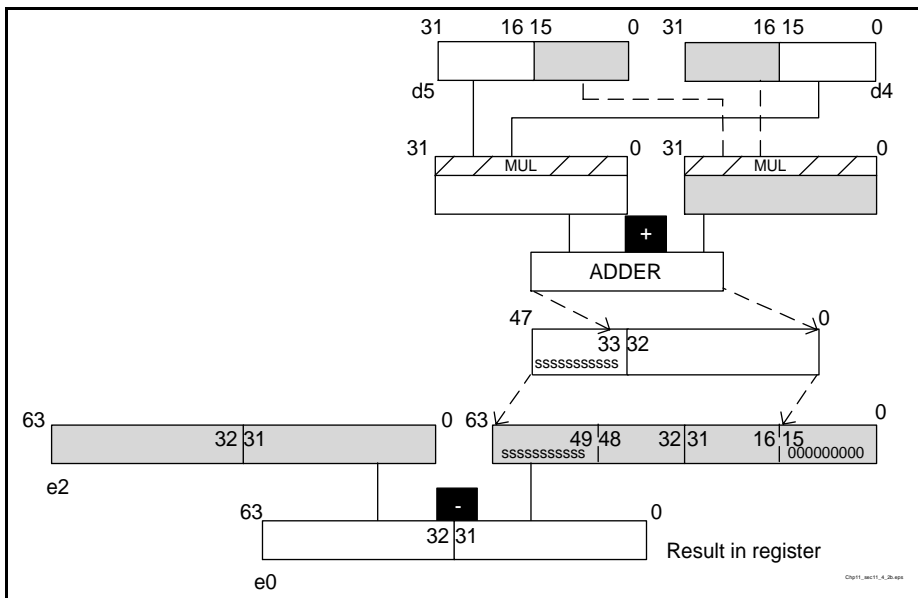
OVF: The result of the last operation (addition or subtraction) can *NOT* be represented in a 64-bit register.

The 4 instructions are shown in the block diagrams which follow. Only the 'lu' variant is shown.

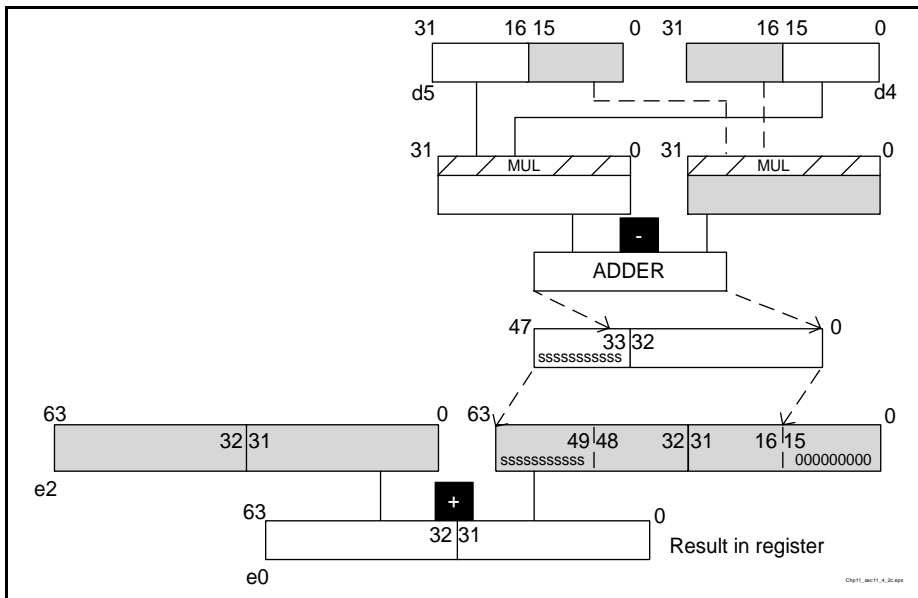
maddm.h e0, e2, d5, d4lu, n



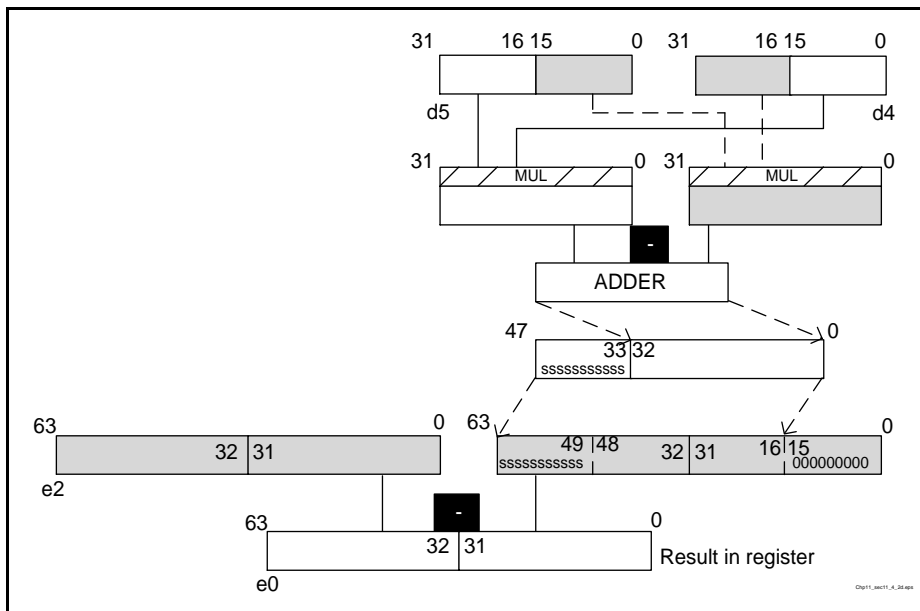
msubm.h e0,e2,d5,d4lu,n



maddsum.h e0,e2,d5,d4lu,n



msubadm.h e0, e2, d5, d4lu, n



Question: Why, in the diagrams, do MSUBM.H and MSUBADM.H seem inverted at first ?

MSUBM.H operations are ADD followed by SUB, whereas MSUBADM.H operations are SUB followed by SUB.

Although this may seem incorrect, it is the exact behaviour and the explanation for this is to do with the function:

If we begin with a single MAC where we compare MADD and MSUB;

```
madd.q == z += x[0]*K[0]
msub.q == z -= x[0]*K[0]
```

Then the same for dual MAC;

```
maddm.h == z += (x[0]*k[0] + x[1]*K[1])
msubm.h == z -= (x[0]*k[0] + x[1]*K[1])
```

Effectively, MSBUM.H is an addition followed by a subtraction. In the same way, MSUBADM is a subtract followed by a subtract:

```
msubadm.h == z -= ( x[0]*K[0] - x[1]*K[1])
```

Note that if a 3-input adder were to be used for the block diagrams, this question would not arise.

```

maddm.h == z = z + x[0]*K[0] + x[1]*K[1]
msubm.h == z = z - x[0]*K[0] - x[1]*K[1]
msubadm.h == z = z - x[0]*K[0] + x[1]*K[1]
madsubm.h == z = z + x[0]*K[0] - x[1]*K[1]

```

The following examples are given for MADDM(S).H and MADDSUM(S).H.

Example 1:

d2	1234 5678
d3	7FFF EFFF
d5	7000 7000
d4	5000 6000

After Instruction

maddm.h	e0, e2, d5, d4ul, #1	e0	8000 89FF 1234 5678
maddm.h	e0, e2, d5, d4lu, #1	e0	8000 89FF 1234 5678
maddm.h	e0, e2, d5, d4ll, #1	e0	8000 97FF 1234 5678
maddm.h	e0, e2, d5, d4uu, #1	e0	8000 7BFF 1234 5678
maddms.h	e6, e2, d5, d4ul, #1	e6	7FFF FFFF FFFF FFFF
maddms.h	e6, e2, d5, d4lu, #1	e6	7FFF FFFF FFFF FFFF
maddms.h	e6, e2, d5, d4ll, #1	e6	7FFF FFFF FFFF FFFF
maddms.h	e6, e2, d5, d4uu, #1	e6	7FFF FFFF FFFF FFFF

Example 2:

d2	1234 5678
d3	8000 0035
d5	C000 5000
d4	A000 7000

After Instruction

maddsum.h	e0, e2, d5, d4ul, #1	e0	7FFF EA35 1234 5678
-----------	----------------------	----	---------------------

maddsum.h e0, e2, d5, d4lu, #1	e0	8000 0435 1234 5678
maddsum.h e0, e2, d5, d4ll, #1	e0	7FFF 8235 1234 5678
maddsum.h e0, e2, d5, d4uu, #1	e0	7FFF 9435 1234 5678
maddsum.h e6, e2, d5, d4ul, #1	e6	8000 0000 0000 0000
maddsum.h e6, e2, d5, d4lu, #1	e6	8000 0435 1234 5678
maddsum.h e6, e2, d5, d4ll, #1	e6	8000 0000 0000 0000
maddsum.h e6, e2, d5, d4uu, #1	e6	8000 0000 0000 0000

Why 48-bit aligned on the left ?

The result (or accumulation) size is a 48-bit. There are 2 ways to put 48-bit in a 64-bit register - on the left or on the right.

Why on the left ?

There are multiple reasons to have the 48-bit on the left:

- 32-bit of guard bit was deemed to be excessive
- It simplifies the hardware as the detection of overflow and advanced overflow is performed on the same bits as for other 64-bit results (bit 63 and bit 62)
- Extracting the number (CL.S followed by DEXTR) is straightforward since the resulting most significant bits (bit 31+growth bits), will lie in the middle of a single register (bit 16 of the odd register).

If the result were right aligned, the most significant bits could be in the odd register (the accumulator has grown above 32-bit), but it could also be in the even register (the accumulator has grown downward, smaller than 32-bit).

11.5 Applications

The following examples illustrate the previous instructions:

- **Dot Product**
- **Vector Multiplication**
- **Matrix Product**

11.5.1 Dot Product

Equation: $Y = \sum X_l * K_l$ For $l = 0..15$ X, K are 16-bit values

Example:

```
lea      a3,3                                ;initialization of counter (n/4 -1)
lea      a2,xdotvalue
lea      a1,kdotvalue
mov      d0,#0                               ;initialization of y =0 (lower)
ld.w     d5,[a1+]4                           ;|| x0x1
mov      d1,#0                               ;initialization of y =0 (upper)
ld.d     e2,[a2+]8                           ;|| k0k1k2k3
dotloop: madds.h  e0,e0,d2,d5u1,#1           ;y=y+x0k0; z=z+x1k1
         ld.d     e4,[a1+]8                   ;|| x2x3x4x5
         madds.h  e0,e0,d3,d4u1,#1           ;y=y+x2k2; z=z+x3k3
         ld.d     e2,[a2+]8                   ;|| k4k5k6k7
loop     a3,dotloop
adds     d0,d1,d0                            ;y = y+z
```

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7 / d6	Load / Store
mov d0,#0	y(=0)		x1x0		ld x0x1
mov d1,#0	y(=0)	k3k2 / k1k0			ld k0k1k2k3
madds.h e0,e0,d2,d5u1,#1	y+x1k1 / z+x0k0		x5x4 / x3x2		ld x2x3x4x5
madds.h e0,e0,d3,d4u1,#1	y+x3k3 / z+x2k2	k7k6 / k5k4			ld k4k5k6k7
adds d0,d1,d0	y+z				

An alternative method is to use MADDH.H, as it avoids using the last addition since it adds the results of the multiplications.

Example:

```

lea      a3,3                                ;initialization of counter (n/4 -1)
lea      a2,x2value
lea      a1,y2value
mov      d0,#0                               ;initialization of y =0 (lower)
ld.w     d5,[a1+]4                           ;|| x0x1
mov      d1,#0                               ;initialization of y =0 (upper)
ld.d     e2,[a2+]8                           ;|| k0k1 | k2k3
dot2loop: maddms.h e0,e0,d2,d5ul,#1           ;y=y+x0k0+x1k1
ld.d     e4,[a1+]8                           ;|| x2x3 | x4x5
maddms.h e0,e0,d3,d4ul,#1                   ;y=y+x2k2+x3k3
ld.d     e2,[a2+]8                           ;|| k4k5 | k6k7
loop     a3,dot2loop

```

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7 / d6	Load / Store
mov d0,#0	y(=0)		x1x0		ld x0x1
mov d1,#0	y(=0)	k3k2 / k1k0			ld k0k1k2k3
maddms.h e0,e0,d2,d5ul,#1	y+x0k0+x1k1		x5x4 / x3x2		ld x2x3x4x5
maddms.h e0,e0,d3,d4ul,#1	y+x2k2+x3k3	k7k6 / k5k4			ld k4k5k6k7

The drawback here is that the result is accumulated with 16 guard bits. Saturation will only act if the accumulation is bigger than 48-bit. In the example the loop is very short and this cannot happen.

To write the result back to memory, the programmer has several options:

- Use CLS (Count Leading Sign) to find where the value lies in the 64-bit register. Follow this with a DEXTR to align the result, before writing back to memory. This is valid if the result is greater than 32-bit.
- The programmer can write back a 32-bit (or 16-bit) result directly to memory without performing any shift. Then the programmer must be sure that the result is not bigger than 32-bit (or 16-bit), since no saturation is applied.

11.5.2 Vector Multiplication

Equation: $Z[i] = X[i] * Y[i]$ For $i = 0 \dots 15$ Z, X, Y are 16-bit values

Example:

```
lea      a3, 0x03                                ; initialization of counter
lea      a2, x2values
lea      a1, y2values
lea      a4, z2values
ld.w     d5, [a2+]4                               ; x0x1
ld.d     e6, [a1+]8                               ; y0y1y2y3
vect2loop: mulr.h  d0, d5, d6ul, #1                ; x0*y0 | x1*y1
          ld.d     e4, [a2+]8                       ; || x2x3 | x4x5
          mulr.h  d1, d4, d7ul, #1                ; x2*y2 | x3*y3
          ld.d     e6, [a1+]8                       ; || y4y5 | y6y7
          st.d     [a4+]8, e0                       ;
loop     a3, vect2loop
```

Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7 / d6	Load / Store
			x1x0 (d5)		ld x0x1
				y1y0 / y3y2	ld y0y1y2y3
mulr.h d0, d5, d6ul, #1	x1y1 / x0y0		x5x4 / x3x2		ld x2x3x4x5
mulr.h d1, d4, d7ul, #1		x3y3 / x2y2		y7y6 / y5y4	ld y4y5y6y7
					st e0

11.5.3 Matrix Product

Equation:

$$\begin{array}{|c|c|c|} \hline A & K & X \\ \hline a & b & c \\ d & e & f \\ g & h & i \\ \hline \end{array}
 \begin{array}{|c|} \hline * \\ \hline \end{array}
 \begin{array}{|c|} \hline k_0 \\ k_1 \\ k_2 \\ \hline \end{array}
 =
 \begin{array}{|c|} \hline x \\ y \\ z \\ \hline \end{array}$$

$$x = a * k_0 + b * k_1 + c * k_2$$

$$y = d * k_0 + e * k_1 + f * k_2$$

$$z = g * k_0 + h * k_1 + i * k_2$$

$$X_i = \sum A_{ij} * K_j \quad \text{for } j=0..n \quad X_i, A_{ij}, K_j \text{ are 16-bit values}$$

Example:

```

lea      a2,mvalue
lea      a1,kvalue
lea      a4,value
mov      d7,#0
ld.w     d3,[a2+]4           ;ab
movh     d6,#0x8000         ;move 0x8000 for rounding
ld.d     e4,[a1+]8          ;k0k1k2,0

maddm.h  e0,e6,d3,d4ul,#1    ;x=a*k0+b*k1+0x8000 (to round)
ld.d     e2,[a2+]8           ;c,0,de
madd.q   e0,e0,d2l,d5l,#1    ;x=x+c*k2
st.h     [a4+]2,d1           ;store 16-bit result

maddm.h  e0,e6,d3,d4ul,#1    ;y=d*k0+e*k1+0x8000 (to round)
ld.d     e2,[a2+]8           ;f,0,gh
madd.q   e0,e0,d2l,d5l,#1    ;y=y+f*k2
st.h     [a4+]2,d1           ;store 16-bit result

maddm.h  e0,e6,d3,d4ul,#1    ;z=g*k0+h*k1+0x8000 (to round)
ld.d     e2,[a2+]8           ;i,0,0,0
madd.q   e0,e0,d2l,d5l,#1    ;z=z+i*k2
st.h     [a4+]2,d1           ;store 16-bit result

```


Register diagram:

Instruction	d1 / d0	d3 / d2	d5 / d4	d7/ d6	Load / Store
		ba			ld ab
			0k2k1k0		ld k0k1k2,0
maddm.h e0, e6, d3, d4u1, #1	$x = a * k_0 + b * k_1 + 0x8000$	de,0,c			ld c,0,de
madd.g e0, e0, d2l, d5l, #1		$x = x + c * k_2$			

The MADDM.H instruction is applied so that the result is a 16-bit value (since the inputs are 16-bit values and must remain in the same format) and a 64-bit result is obtained. The location of the most significant bits in the 64-bit register must be located so that the result can be rounded.

The calculation is:

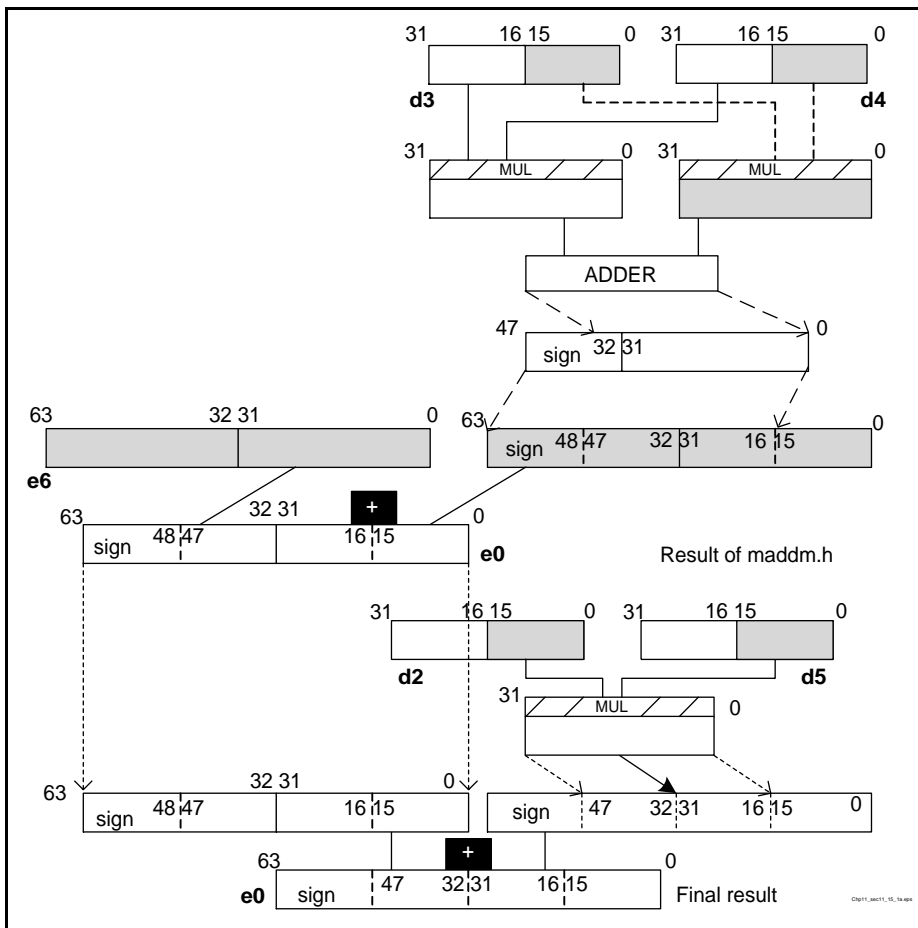
$$a * k_0 + b * k_1 + c * k_2 + 0x8000$$

which can be reorganized in:

$$\underline{a * k_0 + b * k_1 + 0x8000} + c * k_2 \quad \text{because addition is commutative.}$$

maddm.h

The page which follows shows an example of how one result of the matrix is computed.



After the adder, the result is put in the 64-bit register with 16 bits of guard bits. The 32 most significant bits are between the bits 47 and 16. Rounding must be done with the upper half of the lower register. This is why 0x80000000 is added in d6.

The result of the MADDM.H instruction has 16 bits of sign. The result of the multiplication of the MADD.Q instruction is in the same format. The two instructions can therefore be interleaved without problems.

11.6 Summary Tables

11.6.1 Dual MAC Operation Differences

	Operation on left	Operation on right	Equations
MADD.H	+	+	$D1 = D3 + ((D4[up] * D5[up]) \ll n);$ $D0 = D2 + ((D4[lo] * D5[lo]) \ll n)$
MSUB.H	-	-	$D1 = D3 - ((D4[up] * D5[up]) \ll n);$ $D0 = D2 - ((D4[lo] * D5[lo]) \ll n)$
MADDSU.H	+	-	$D1 = D3 + ((D4[up] * D5[up]) \ll n);$ $D0 = D2 - ((D4[lo] * D5[lo]) \ll n)$
MSUBAD.H	-	+	$D1 = D3 - ((D4[up] * D5[up]) \ll n);$ $D0 = D2 + ((D4[lo] * D5[lo]) \ll n)$

		Operation on the first adder	
		+	-
Operation on the second adder	+	MADDM.H	MADDSUM.H
	-	MSUBM.H	MSUBADM.H

11.7 List of Instructions

	MUL.H
	mul.h e0,d4,d5u1,#1 mul.h e0,d4,d5lu,#1 mul.h e0,d4,d5ll,#1 mul.h e0,d4,d5uu,#1
rounding	mulr.h d0,d4,d5u1,#1 mulr.h d0,d4,d5lu,#1 mulr.h d0,d4,d5ll,#1 mulr.h d0,d4,d5uu,#1

	MADD.H	MSUB.H
	madd.h e0,e2,d4,d5ul,#1 madd.h e0,e2,d4,d5lu,#1 madd.h e0,e2,d4,d5ll,#1 madd.h e0,e2,d4,d5uu,#1	msub.h e0,e2,d4,d5ul,#1 msub.h e0,e2,d4,d5lu,#1 msub.h e0,e2,d4,d5ll,#1 msub.h e0,e2,d4,d5uu,#1
saturation	madds.h e0,e2,d4,d5ul,#1 madds.h e0,e2,d4,d5lu,#1 madds.h e0,e2,d4,d5ll,#1 madds.h e0,e2,d4,d5uu,#1	msubs.h e0,e2,d4,d5ul,#1 msubs.h e0,e2,d4,d5lu,#1 msubs.h e0,e2,d4,d5ll,#1 msubs.h e0,e2,d4,d5uu,#1
rounding	maddr.h d0,e2,d4,d5ul,#1 maddr.h d0,d2,d4,d5ul,#1 maddr.h d0,d2,d4,d5lu,#1 maddr.h d0,d2,d4,d5ll,#1 maddr.h d0,d2,d4,d5uu,#1	msubr.h d0,e2,d4,d5ul,#1 msubr.h d0,d2,d4,d5ul,#1 msubr.h d0,d2,d4,d5lu,#1 msubr.h d0,d2,d4,d5ll,#1 msubr.h d0,d2,d4,d5uu,#1
rounding saturation	maddrs.h d0,e2,d4,d5ul,#1 maddrs.h d0,d2,d4,d5ul,#1 maddrs.h d0,d2,d4,d5lu,#1 maddrs.h d0,d2,d4,d5ll,#1 maddrs.h d0,d2,d4,d5uu,#1	msubrs.h d0,e2,d4,d5ul,#1 msubrs.h d0,d2,d4,d5ul,#1 msubrs.h d0,d2,d4,d5lu,#1 msubrs.h d0,d2,d4,d5ll,#1 msubrs.h d0,d2,d4,d5uu,#1

	MADDSU.H	MSUBAD.H
	maddsu.h e0,e2,d4,d5ul,#1 maddsu.h e0,e2,d4,d5lu,#1 maddsu.h e0,e2,d4,d5ll,#1 maddsu.h e0,e2,d4,d5uu,#1	msubad.h e0,e2,d4,d5ul,#1 msubad.h e0,e2,d4,d5lu,#1 msubad.h e0,e2,d4,d5ll,#1 msubad.h e0,e2,d4,d5uu,#1
saturation	maddsus.h e0,e2,d4,d5ul,#1 maddsus.h e0,e2,d4,d5lu,#1 maddsus.h e0,e2,d4,d5ll,#1 maddsus.h e0,e2,d4,d5uu,#1	msubads.h e0,e2,d4,d5ul,#1 msubads.h e0,e2,d4,d5lu,#1 msubads.h e0,e2,d4,d5ll,#1 msubads.h e0,e2,d4,d5uu,#1
rounding	maddsur.h d0,d2,d4,d5ul,#1 maddsur.h d0,d2,d4,d5lu,#1 maddsur.h d0,d2,d4,d5ll,#1 maddsur.h d0,d2,d4,d5uu,#1	msubadr.h d0,d2,d4,d5ul,#1 msubadr.h d0,d2,d4,d5lu,#1 msubadr.h d0,d2,d4,d5ll,#1 msubadr.h d0,d2,d4,d5uu,#1
rounding saturation	maddsurs.h d0,d2,d4,d5ul,#1 maddsurs.h d0,d2,d4,d5lu,#1 maddsurs.h d0,d2,d4,d5ll,#1 maddsurs.h d0,d2,d4,d5uu,#1	msubadrs.h d0,d2,d4,d5ul,#1 msubadrs.h d0,d2,d4,d5lu,#1 msubadrs.h d0,d2,d4,d5ll,#1 msubadrs.h d0,d2,d4,d5uu,#1

	MULM.H
	mulm.h e0,d4,d5ul,#1
	mulm.h e0,d4,d5lu,#1
	mulm.h e0,d4,d5ll,#1
	mulm.h e0,d4,d5uu,#1

	MADDM.H	MSUBM.H
	maddm.h e0,e2,d4,d5ul,#1	msubm.h e0,e2,d4,d5ul,#1
	maddm.h e0,e2,d4,d5lu,#1	msubm.h e0,e2,d4,d5lu,#1
	maddm.h e0,e2,d4,d5ll,#1	msubm.h e0,e2,d4,d5ll,#1
	maddm.h e0,e2,d4,d5uu,#1	msubm.h e0,e2,d4,d5uu,#1
saturation	maddms.h e0,e2,d4,d5ul,#1	msubms.h e0,e2,d4,d5ul,#1
	maddms.h e0,e2,d4,d5lu,#1	msubms.h e0,e2,d4,d5lu,#1
	maddms.h e0,e2,d4,d5ll,#1	msubms.h e0,e2,d4,d5ll,#1
	maddms.h e0,e2,d4,d5uu,#1	msubms.h e0,e2,d4,d5uu,#1

	MADDSUM.H	MSUBADM.H
	maddsum.h e0,e2,d4,d5ul,#1	msubadm.h e0,e2,d4,d5ul,#1
	maddsum.h e0,e2,d4,d5lu,#1	msubadm.h e0,e2,d4,d5lu,#1
	maddsum.h e0,e2,d4,d5ll,#1	msubadm.h e0,e2,d4,d5ll,#1
	maddsum.h e0,e2,d4,d5uu,#1	msubadm.h e0,e2,d4,d5uu,#1
saturation	maddsums.h e0,e2,d4,d5ul,#1	msubadms.h e0,e2,d4,d5ul,#1
	maddsums.h e0,e2,d4,d5lu,#1	msubadms.h e0,e2,d4,d5lu,#1
	maddsums.h e0,e2,d4,d5ll,#1	msubadms.h e0,e2,d4,d5ll,#1
	maddsums.h e0,e2,d4,d5uu,#1	msubadms.h e0,e2,d4,d5uu,#1

12 Floating-Point Help

TriCore implements Floating-Point (FP) operations in two ways:

- Hardware: By using a Floating-Point coprocessor (not covered in this document)
- Software Library: Available with different compilers (not covered in this document)

Two 'FP Help' instructions are described in this chapter.

12.1 Floating Point Packing / Unpacking (PACK, UNPACK)

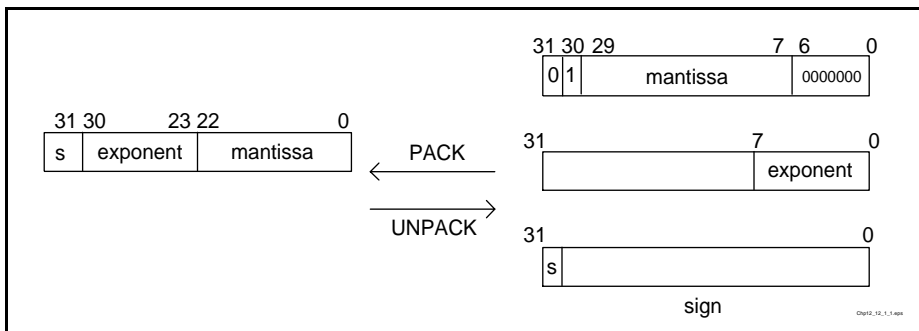
The standard FP 32-bit format represents a 'packed' number. Three types of information are packed into a single 32-bit format (sign, exponent, magnitude). Emulating a floating point operation using standard instructions requires the sign, magnitude and exponent to be in different registers (unpacked). TriCore can therefore pack and unpack a floating-point number in hardware. This saves approximately 10 cycles on each FP operation.

12.1.1 Using Pack and Unpack

The syntax of the PACK and UNPACK instructions are very similar:

```
pack    dest, exponent / mantissa (register pair), sign
```

```
unpack exponent / mantissa (register pair), src
```



Note: UNPACK requires 3 destinations. Two of them are given by a register pair and the third (sign) is given by the source.

Example:

e4	00000025 00056123
d1	8000 0000

After Instruction

`pack d0,e4,d1`

`unpack e2,d0`

d0	8000 0561
e2	FFFFFF82 0002B080

Note: The use of 2 instructions are not symmetrical.

13 Pipeline Operations

This chapter explains the rules governing the organization of instructions to ensure the minimum number of cycles. This is generally called pipeline or pipeline operations. A CPU pipeline is highly dependent on a given implementation. The pipeline operations given here correspond to the TriCore TC10GP class of devices. Since TriCore is a static superscalar device, the rules will not necessarily be the same for future implementations.

Pipeline rules for the classical optimized DSP case, the loop branch, are covered, as are the rules to count the number of cycles. Coprocessor and system instructions are not discussed here.

Four application examples are provided at the end of the chapter, to help visualize the rules.

Chapter contents:

- **General Rules**
- **16x16-bit and 16x32-bit MAC Instructions**
- **32x32-bit MAC Instructions**
- **Loop**
- **Branch**
- **Applications**

Note: Please note that the following acronyms are used in this chapter.

Acronym	Full Title	Comment
IP	Integer Processing unit	Type of instructions working on the data register. For the purpose of this chapter, this excludes all MAC and DIVIDE instructions. These are described from chapter 2 to 7.
LS	Load and Store	Type of instructions working on the address register. These instructions are defined in Chapter 8.
MAC	Multiply Accumulate	Type of instructions using the MAC unit. Normally these are a subset of IP instructions. However, for the purposes of this chapter they are dealt with on their own. These instructions are described in chapters 9 and 10.

13.1 General Rules

13.1.1 One Cycle

All IP instructions take one cycle, unless otherwise indicated.

Example:

```
add    d0,d2,d1           ;one cycle
eq     d1,d0,d3           ;one cycle
```

All LS instructions take one cycle, unless otherwise indicated.

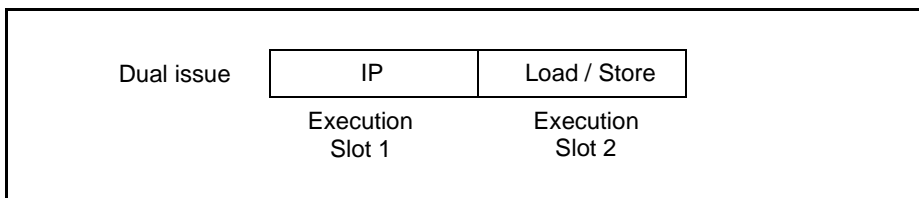
Example:

```
ld.w   d0,[a0]            ;one cycle
st.w   [a0],d0            ;one cycle
add.a  a0,a1,a2           ;one cycle
```

Note: The address arithmetic instructions are part of the LS unit. They take the same number of cycles as a Load or Store instruction.

13.1.2 One Cycle - Dual Issue

TriCore is capable of executing two instructions in a single cycle, provided the first instruction is issued to the IP, followed by an instruction issued to the LS. It is therefore referred to as a dual-issue machine.



Example:

```
add    d0,d2,d1           ;one cycle
ld.w   d4,1x0
eq     d1,d0,d3           ;one cycle
add.a  a0,a1,a2
```

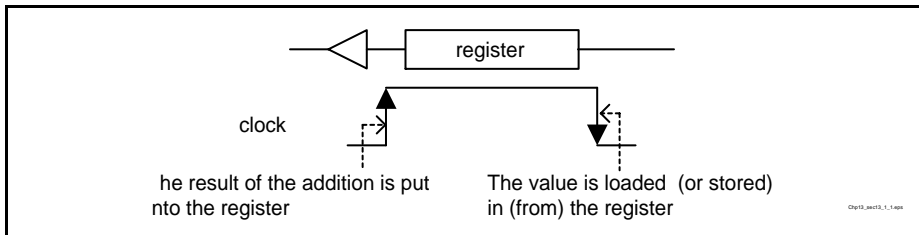
Dependency

There is never any dependency problem when an LS instruction follows an IP instruction.

Example:

```
add    d0,d2,d1           ;one cycle
ld.w   d0,1x0
```

These instructions take one cycle to be executed. However, the value in register d0 will not be the result of the addition. It is overwritten by the loaded value (IX0)), because the load is semantically computed after the addition.



Example 1:

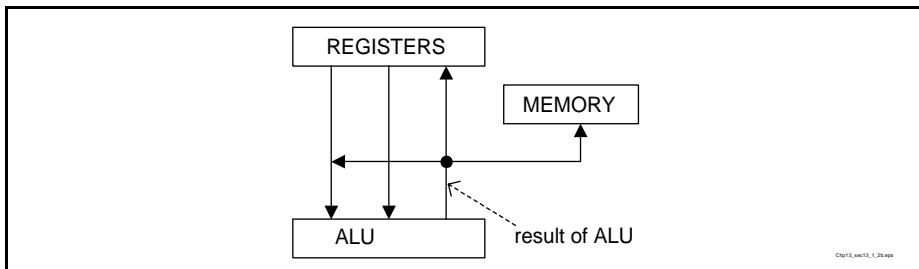
```
add    d1, d0, d3    ;one cycle
ld.w   d0, 1x0
```

Register d0 is read for the addition and then written by the load instruction. There is no problem in this instance, as the addition preceded the load.

Example 2:

```
add    d0, d1, d2    ;one cycle
st.w   [a2], d0      ;
sub    d3, d0, d1    ;one cycle
```

There is no problem with this Store instruction, because access to the memory takes place after the addition. A problem could have arisen from the fact that the destination register d0 is used as the source register in the next instruction. Here, TriCore uses forwarding. The result of the ALU can be sent back to the register and reused as a source to be computed again in the ALU, or can be sent back to the register and stored directly in memory.



13.1.3 Table of Dual Issue Instructions

The following table lists all IP and LS instructions which can be issued in parallel:

List of IP Instructions			List of LS Instructions
ABS.*	EQ	NAND.*	ADD.A
ABSDIF.*	EQ.B	NE	EQ.A
ABSDIFS.*	EQ.H	NOR.*	EQZ.A
ABSS.*	EQ.W	OR.*	GE.A
ADD	EQANY.*	ORN.*	LD.*
ADDC	EXTR.*	PACK	LEA
ADDI	GE	PARITY	LT.A
ADDIH	GE.U	RSUB	MOV.AA
ADDS.*	INS.T	RSUBS.*	MOVH.A
ADDX	INSERT	SAT.*	NE.A
AND.*	INSN.T	SEL	NEZ.A
ANDN.*	LT	SELN	ST.*
BMERGE	LT.B	SH.*	SUB.A
BSPLIT	LT.BU	SHA.*	
CADD	LT.H	SHAS	
CADDN	LT.HU	SUB	
CLO.*	LT.U	SUB.B	
CLS.*	LT.W	SUB.H	
CLZ.*	LT.WU	SUBC	
CMOV	MADD.* ¹⁾	SUBS.*	
CMOVN	MAX.*	SUBX	
CSUB	MIN.*	UNPACK	
CSUBN	MOV.*	XNOR.*	
DEXTR	MOVH	XOR.*	
	MSUB* ¹⁾		
	MUL* ¹⁾		

¹⁾ The MUL, MADD, MSUB instructions follow some special rules which are explained in the sections which follow.

13.2 16x16-bit and 16x32-bit MAC Instructions

This section presents the pipelining rules applied to the 16x16-bit and 16x32-bit MAC instructions.

13.2.1 One Cycle –Rule 1

Since the MAC is a pipeline unit, all single instructions take more than one cycle, except 16x16 (mul.q).

Example: sOffset+ (sX-sY)²

```
sub.h      d1,d2,d3          ;one cycle
mul.q      d0,d1u,d1u,#1     ;one cycle
add.h      d5,d4,d0          ;one cycle
```

Example: sWeight * (sX-sY)²

```
sub.h      d1,d2,d3          ;one cycle
mul.q      d0,d1u,d1u,#1     ;one cycle
mul.q      d5,d4u,d0u,#1     ;one cycle
```

13.2.2 One Cycle –Rule 2

A 16*16 (16*32) MAC instruction will take one cycle if it is followed by another MAC instruction *AND* it does not meet Two Cycles - Rule 2, *AND* it does not meet Two Cycles - Rule 3 (See section [Section 13.2.6](#)).

Example:

```
mul.h      e0,d4,d2u1,#1     ;one cycle
madd.h     e0,e4,d4,d2u1,#1  ;one cycle
mul.h      e8,d2,d3u1,#1     ;one cycle
```

The destination registers (e0, e8) are never used as sources to the multiplier, so there is no stall.

Example:

```
madd.h     e0,e2,d4,d2u1,#1  ;one cycle
madd.h     e2,e0,d6,d7u1,#1  ;one cycle
madd.h     e2,e0,d6,d7u1,#1  ;one cycle
```

The destination register e0 is used as a source register in the next instruction. This is possible, as it is an accumulation.

13.2.3 One Cycle - Dual Issue

TriCore is capable of executing two instructions in a single cycle, provided that the first instruction is issued to the MAC, followed by an instruction issued to the LS. However, unlike the IP instructions, there are some dependency issues to be taken into account. These are discussed later.

Dual Issue	MAC	Load / Store
	Execution Slot 1	Execution Slot 2

Example:

```
madd.h    e0,e2,d4,d2u1,#1 ;one cycle
ld.w      d6,lx0
madd.h    e2,e0,d6,d7u1,#1 ;one cycle
st.d      llx0,e0
msub.h    e4,e2,d4,d7u1,#1 ;one cycle
st.d      llx1,e2
```

	MAC Execution Slot 1	Load / Store Execution Slot 2
Dual Issue	madd.h e0,e2,d4,d2u1,#1	ld.w d6,lx0
Dual Issue	madd.h e2,e0,d6,d7u1,#1	st.d llx0,
Dual Issue	msub.h e4,e2,d4,d7u1,#1	st.d llx1,e2

13.2.4 Two Cycles – Rule 1

If a 16x16 (16x32) MAC instruction is followed by any IP instruction, it will take two cycles. The exception is 16x16 mul.q, which will take one cycle.

Example:

```
madd.q    d0,d1,d2l,d3l,#1 ;two cycles
add       d4,d5,d6          ;one cycle
```

13.2.5 Two Cycles – Rule 2

When the destination register is used as a source register in the next multiplication, the 16x16 (16x32) MAC instruction takes two cycles.

Example:

```
madd.q    d0,d1,d2,d3u,#1    ;two cycles
madd.q    d4,d5,d0u,d6u,#1   ;one cycle
madd.q    d4,d5,d0u,d6u,#1   ;one cycle
```

13.2.6 Two Cycles – Rule 3

If a 16x16 (16x32) MAC instruction is followed by a store instruction (i.e. store the destination register just computed by the MAC), the store will *NOT* be executed in parallel. The MAC instruction will take two cycles instead of one.

Example:

```
madd.h    e0,e4,d2,d3ul,#0   ;
madd.h    e0,e4,d2,d3ul,#1   ;two cycles
st.d      [a2],e0
```

	MAC Execution Slot 1	Load / Store Execution Slot 2
Single Issue	madd.he 0,e4,d2,d3ul,#0	-
Single Issue	madd.he 0,e4,d2,d3ul,#1	-
Single Issue	-	st.d [a2],e0

In this example the data dependency will cause a stall. It takes two cycles to get the MAC result and the store instruction must wait for that result.

Note: If the MAC instruction is followed by a load instruction and then by a store instruction, all three instructions will still take two cycles.

Example:

```
madd.q    d0,d1,d2,d3u,#1    ;
madd.q    d0,d1,d2,d3u,#1    ;one cycle
ld.w      d2,lX3
st.w      [a2],d0             ;one cycle
```

	MAC Execution Slot 1	Load / Store Execution Slot 2
Single Issue	madd.q d0,d1,d2,d3u,#1	-
Dual Issue	madd.q d0,d1,d2,d3u,#1	ld.w d2,lX3
Single Issue	-	st.w [a2],d0

13.3 32x32-bit MAC Instructions

This section presents the pipelining rules applied to the 32x32-bit MAC instructions.

13.3.1 Two Cycles – Rule 1

A 32x32-bit MAC instruction takes two cycles when it is followed by another MAC instruction, *except* if it meets one of the three cycles rules.

Example:

```
mul.q    d0,d1,d2,#1        ;two cycles (32x32-bit)
madd.q   d3,d4,d5,d6,#1     ;two cycles (32x32-bit)
mul.q    d7,d8,d9,#1        ;
```

13.3.2 Two Cycles – Rule 2

If the 32x32 MAC instruction is followed by a load/store instruction and then by a MAC instruction, it will take two cycles.

Example:

```
madd.q   d0,d1,d2,d3,#1     ;two cycles
ld.w     d2,lX0              ;
mul.q    d4,d5,d6,#1        ;two cycles
st.w     [a2],d0             ;
madd.q   d7,d8,d9u,d10u,#1   ;one cycle (16x16-bit)
```

13.3.3 Three Cycles – Rule 1

If the 32x32 MAC instruction is followed by a load/store instruction, and then by something other than a MAC, it will take three cycles.

Example 1:

```
madd.q    d0,d1,d2,d3,#1    ;three cycles
ld.w      d2,1X0            ;
add       d4,d5,d6
```

Example 2:

```
madd.q    d0,d1,d2,d3,#1    ;three cycles
st.w      [a2],d0           ;
add       d4,d5,d6          ;one cycle
```

13.3.4 Three Cycles – Rule 2

If a 32x32 MAC instruction is followed by an IP instruction, it will take three cycles.

Example:

```
madd.q    d0,d1,d2,d3,#1    ;three cycles
add       d4,d5,d6          ;one cycle
```

13.3.5 Three Cycles – Rule 3

When the destination register is used as a source register in the next multiplication, the first 32x32 MAC instruction will take three cycles since the second instruction must wait for its result.

Example:

```
mul.q     d0,d1,d2,#1        ;three cycles
madd.q    d3,d4,d0,d6,#1     ;two cycles
mul.q     d3,d4,d5,#1        ;
```


13.4 Dependency Table

The dependency table gives the current instruction cycle depending on the previous instruction. For example:

- Row 1 shows that if the current instruction is an IP instruction, it will always take 1 cycle.
- Row 3 shows that if the current instruction is a Store instruction (ST), it will take 0 cycles if dual issued (IP or MAC), 1 cycle if following a LD or ST instruction, and 1 cycle if there is data dependency following a MAC instruction.

		Previous Instruction				
		IP	LD	ST	MAC 16*16 and 16*32	MAC 32*32
Current Instruction (Cycles)	IP	1	1	1	1	1
	LD	0	1	1	1	1
	ST	0	1	1	0 or 1 ¹⁾	0 or 1 ¹⁾
	MAC 16*16 and 16*32	2	2	2	1	1
	MAC 32*32	3	3	3	2	2

¹⁾ Data Dependency;

Note: Division and system instructions are not considered here.

Example:

```
adds    d2,d1,d1      ;one cycle
madd    d0,d2,d5,d6    ;three cycles
```

The first instruction is an IP instruction, and will take one cycle. The second is a MAC 32*32-bit instruction preceded by a MAC 32*32 instruction, and will take 2 cycles. The table shows that a dependent store after a MAC 32*32 will take one cycle.

Note: A problem with the dependency table is that it over-simplifies the behaviour of the MAC followed by STORE. For a complete picture, the programmer will need to refer to all of the rules given in the previous 'Rules' sections.

13.5 Loop

The loop instruction takes one cycle on the first execution of the loop, and one cycle to exit the loop. During the execution of the loop, the LOOP instruction is 'folded out' and effectively takes zero cycles.

Example:

```
lea      a2,3
ld.w     d1,1X0
aloop:
    add    d0,d1,#4    ;one cycle * counter
    rsub   d2,d0,#0    ;one cycle * counter
    ld.w   d1,1X0      ;
loop     a2,aloop      ;two cycles (one to enter and one to exit)
```

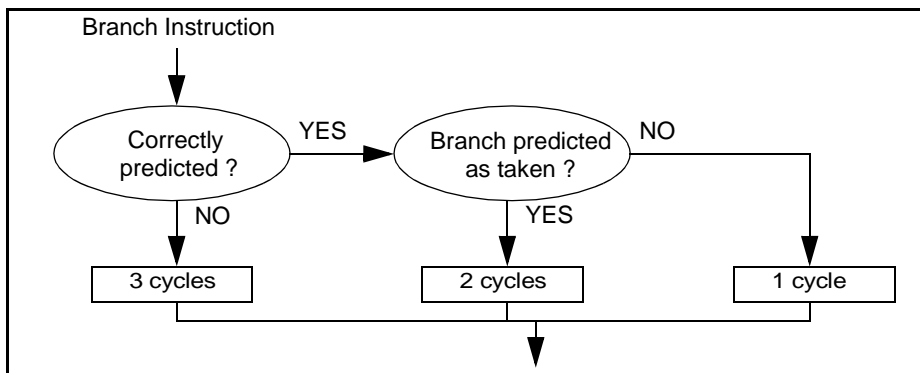
The total number of cycles in the loop is $4 \times (1+1) + 2 = 10$.

13.6 Branch

13.6.1 Counting Cycles for Branch Instructions

The branch behaviour is based on prediction. The rule is that if the branch is a 16-bit instruction (backward or short forward jump), the branch is predicted as taken. If the branch is a 32-bit instruction and is a backward jump, it is predicted as taken. However if it is a forward jump it is predicted as not taken.

The following figure describes the cycles:



- A branch correctly predicted as taken (including unconditional) is 2 cycles.
- a branch correctly predicted as not taken is 1 cycle.
- a branch incorrectly predicted is 3 cycles.

Example:

```
;d1 = 2, d2 = 16, d4 = 20
add    d3,d1,#3          ;one cycle
started: jlt    d2,d3,end  ;prediction = no (long forward)
        mul    d3,d1,d3    ;one cycle
end:    add    d3,d3,#7    ;one cycle
jge     d4,d3,started     ;prediction = yes (backward)
```

Execution	Correctly predicted ?	Cycles
add d3,d1,#3		1
jlt d2,d3,end	yes (not taken)	1
mul d3,d1,d3		1
add d3,d3,#7		1
jge d4,d3,started	yes (taken)	2
jlt d2,d3,end	no	3
add d3,d3,#7		1
jge d4,d3,started	no	3

Total number of cycles: 13

13.7 Applications

In this section four examples illustrate the rules previously described. Only the loops are studied.

13.7.1 FIR

```

lea      LC, (n/4 - 1)                ;get loop number
mov      d0, #0                      ;acc= 0 lower
ld.w     d5, [Kptr+]4                ;||load   k0, k1,
mov      d1, #0                      ;acc= 0 upper
ld.d     e2, [XBptr+c]8              ;||load   x0, x1, x2, x3
sfirloop :                          ;-----
      maddm.h   e0,e0,d2,d5ul,#0      ;acc +=  x0*k0 + x1*k1      (1)
      ld.d      e4, [Kptr+]8          ;||load   k2,k3,k4,k5
      maddm.h   e0,e0,d3,d4ul,#0      ;acc +=  x2*k2 + x3*k3      (2)
      ld.d      e2, [XBptr+c]8        ;||load   x4,x5,x6,x7
loop     LC,sfirloop                  ;-----
;;epilog
dextr    d0,d1,d0,#16                ;
                                           ;acc= extract_32bit(acc)

```

There is no problem on the number of cycles. Each MADDM.H instruction takes one cycle because it is always followed by another MADDM.H instruction (MAC instruction - One Cycle – Rule 2).

13.7.2 Vector Quantization

```

lea      LC, (n/2-1)                ;get loop number
mov      d0, #0                      ;accEV=0
ld.w     k, [Kptr+]4                 ;||load k0, k1
mov      d1, #0                      ;accOD=0
ld.w     x, [Xptr+]4                 ;||load x0, x1
vqloop:                                     ;-----
      subs.h    d5,k,x                ;k0-x0 || k1-x1      (1)
      ld.w      k, [Kptr+]4           ;||load   k2, k3
      madds.h   e0,e0,d5,d5ul,#1      ;accEV+= (k0-x0)**2
                                           ;accOD+= (k1-x1)**2 (2,3)
      ld.w      x, [Xptr+]4           ;||load   x2,x3
loop     LC,vqloop                    ;-----
adds     d0,d0,d1                     ;acc = accEV+accOD

```

The MADDS.H instruction takes two cycles because it is followed by a SUBS.H instruction. This is illustrated in the issue slot diagram which follows.

Cycle	IP	LS	Note
1		lea LC, (n/2-1)	
2	mov d0, #0	ld.w k, [Kptr+]4	
3	mov d1, #0	ld.w x, [Xptr+]4	
4	subs.h d5, k, x	ld.w k, [Kptr+]4	;first pass
5	madds.h e0, e0, d5, d5ul, #1	ld.w x, [Xptr+]4	
6		loop LC, vqloop	
7	subs.h d5, k, x	ld.w k, [Kptr+]4	;second pass
8,9	madds.h e0, e0, d5, d5ul, #1	ld.w x, [Xptr+]4	
10	subs.h d5, k, x	ld.w k, [Kptr+]4	;third pass
11,12	madds.h e0, e0, d5, d5ul, #1	ld.w x, [Xptr+]4	

13.7.3 Complex Wave Generation

```

lea      LC, (n-1)                ;
ld.w     k, [Kptr+]4              ;ld b0,k0
ldloop:                                     ;-----
mulr.h    temp,xy,k ll,#1         ; y' = y*b || x' = x*b(1)
st.w     [OUTptr+]4,xy            ; ||st x1,y1 (next loop)
maddsur.h xy,temp,xy,k uu,#1     ; y' += x*k || x' -= y*k(2,3)
ld.w     k, [Kptr+]4              ; ||ld b1,k1
loop     LC,ldloop                ;-----
st.w     [OUTptr+]4,xy            ; ||st last x,y

```

Consider the following issue slot diagram:

Cycle	IP	LS	Note
1	mulr.h temp,xy,ke,#1	st.w [OUTptr+]4,xy	;first iteration
2	maddsur.h xy,temp,xy,kuu,#1		
3	mulr.h temp,xy,ke,#1	st.w [OUTptr+]4,xy	;second iteration
4	maddsur.h xy,temp,xy,kuu,#1		

A problem arises from the fact that xy, which is a destination register of MADDSUR.H, is used as a multiplication source register by MULR.H. The MADDSUR.H will take two cycles (See MAC instruction - **Two Cycles – Rule 2**).

13.7.4 FFT - DIT – Butterfly

Example:

```
lea      LC, (n-1)                ;
ld.w     x, [Xptr+]4              ;ld yr0,yi0
ld.w     y, [Yptr+]4              ;ld xr0,xi0
ld.w     k, [Kptr]                ;ld kr,ki ( k fixed )
loopdit:                               ;-----
      maddr.h    xx,x,y,k e,#1      ; (1)
      st.w       [Yptr]-8,yy         ; ||st yr'-1, yi'-1 in place
      maddsur.h   xx,xx,y,k uu,#1    ; (2,3)
      st.w       [Xptr]-4,xx         ; ||st xr'0, xi'0 in place
      msubr.h     yy,x,y,k e,#1      ; (4)
      ld.w       x, [Xptr+]4         ; ||ld xr1,xi1
      msubadr.h   yy,yy,y,k uu,#1    ; (5)
      ld.w       y, [Yptr+]4         ; ||ld yr1,yi1
loop     LC, loopdit                ;-----
st.w     [Yptr]-8,yy
```

The MADDSUR.H instruction will take two cycles because xx, which is the destination register of the MADDSUR.H instruction, is stored in the next instruction (See MAC instruction – **Two Cycles – Rule 3**).

14 Instruction Set Summary

The Instruction Set Summary follows the functional order of the chapters, so instructions working on data registers are given first (arithmetic, logic, bit-fields, and so on), then address registers and finally 'system' instructions.

Conventions

Note: This Instruction Set Summary uses the conventions described in this chapter.

The following example from the first two lines from one of the tables in this section ([Section 14.1.2, Add, Sub, Rsub](#)), are used to illustrate the conventions employed:

Name / Example	Description
ADD(S) D0, D1, D2	D0 = D1 + D2
ADD.S.U D0, D1, D2	D0 = D1 + D2 ; note : U means unsigned

The first column, Name / Example, is constructed as follows:

- **Name**

TriCore Instruction names are based on a functional mnemonic (ADD).

The suffix (S) is added as necessary to indicate the availability of the Saturated data type.

- **Registers**

Rather than using the formal syntax with symbolic registers (ADD Da,Db,Dc for example), real register names are used (ADD D0,D1,D2). As a general rule, any registers can replace the registers used as examples. There are no dedicated registers.

- D1, D2, D3, D4, D5 are used as examples of Source Data registers
- D0 is used as an example of a Destination Data register
- E2 and E4 are used as examples of Source Extended Data registers
- E0 is used as an example of a Destination Extended Data register
- A3, A4 are used as examples of Source Address registers
- A2 is used as an example of a Destination Address register

- **Constants**

- K4: 4-bit constant sign extended to 32-bit
- K8: 8-bit constant
- K9Z: 9-bit constant zero-extended to 32-bit
- K9S: 9-bit constant sign-extended to 32-bit
- K16: 16-bit constant
- K16S: 16-bit constant sign-extended to 32-bit
- K16Z: 16-bit constant zero-extended to 32-bit

Note: When the instruction can take registers or constants as operands, the described syntax is for registers. Refer to the Appendix 'Constants' to check which instructions are also available with constants.

The second column in the example, Description, is intended to give a formal but simplified description. A condensed notation has been used:

Condensed Notation	Definition
D0	Data register (32bit)
D0[4]	Bit 4 of Data register D0
D0[31..0]	Bits 31 to 0 of Data register D0
D0[up]	D0[31..16]
D0[lo]	D0[15..0]
D0[H]	D0[31..16], D0[15..0] == Packed Half-word Data register
D0[B]	D0[31..24], D0[23..16], D0[15..8], D0[7..0] == Packed Byte Data register
D0[SH1]	D0 left-shifted by 1
E0	D1[31..0] D0[31..0] Extended Data register (64-bit)
psw.c	carry flag

14.1 List of Instructions

14.1.1 Data Move

MOV D0,D1	D0 = D1
MOV D0,K16S	DO = K16S
MOV.U D0,K16Z	DO = K16Z
MOVH DO,K16	DO[up] = k16; D0[lo] = 0x0
MOV.D D0,A3	DO = A3
MOV.A A3,D0	A3 = D0

14.1.2 Add, Sub, Rsub

ADD(S) D0, D1, D2	D0 = D1 + D2
ADDS.U D0, D1, D2	D0 = D1 + D2 ; note : U means unsigned
SUB(S) D0, D1, D2	D0 = D1 – D2
SUBS.U D0, D1, D2	D0 = D1 – D2
ADDI D0, D1, K16S	D0 = D1 + K16S
ADDIH D0, D1, K16	D0[up] = D1[up] + K16 ; D0[lo] = D1[lo]
ADD(S).H D0, D1, D2	D0[H] = D1[H] + D2[H]
ADDS.HU D0, D1, D2	D0[H] = D1[H] + D2[H]
SUB(S).H D0, D1, D2	D0[H] = D1[H] - D2[H]
SUBS.HU D0, D1, D2	D0[H] = D1[H] - D2[H]
ADD.B D0, D1, D2	D0[B] = D1[B] + D2[B]
SUB.B D0, D1, D2	D0[B] = D1[B] – D2[B]
ADDX D0, D1, D2	D0 = D1 + D2 ; psw.c = Carry out
ADDC D0, D1, D2	D0 = D1 + D2 + psw.c ; psw.c = Carry out
SUBX D0, D1, D2	D0 = D1 - D2 ; psw.c = Carry out
SUBC D0, D1, D2	D0 = D1 - D2 + (psw.c -1) ; psw.c = Carry out
RSUB(S) D0, D1, K9S	D0 = K9S – D1
RSUBS.U D0, D1, K8Z	D0 = K8Z - D1 ; (note constant is sign extended)

14.1.3 Abs, Absdif

ABS(S) D0, D1	if (D1 >= 0) D0 = D1 else D0 = -D1
ABS(S).H D0, D1	if (D1[H] >= 0) D0[H] = D1[H] else D0[H] = -D1[H]
ABS.B D0, D1	if (D1[B] >= 0) D0[B] = D1[B] else D0[B] = -D1[B]
ABSDIF(S) D0, D1, D2	if (D1 > D2) D0 = D1-D2 else D0 = D2-D1
ABSDIF(S).H D0, D1, D2	if (D1[H] > D2[H]) D0[H] = D1[H] - D2[H] else D0[H] = D2[H] - D1[H]
ABSDIF.B D0, D1, D2	if (D1[B] > D2[B]) D0[B] = D1[B] - D2[B] else D0[B] = D2[B] - D1[B]

14.1.4 Min, Max, Sat

MIN D0, D1, D2	if (D1 < D2) D0 = D1 else D0 = D2
MIN.U D0, D1, D2	if (D1 < D2) D0 = D1 else D0 = D2
MAX D0, D1, D2	if (D1 > D2) D0 = D1 else D0 = D2
MAX.U D0, D1, D2	if (D1 > D2) D0 = D1 else D0 = D2
MIN.H D0, D1, D2	if (D1[H] < D2[H]) D0[H] = D1[H] else D0[H] = D2[H]
MIN.HU D0, D1, D2	if (D1[H] < D2[H]) D0[H] = D1[H] else D0[H] = D2[H]
MAX.H D0, D1, D2	if (D1[H] > D2[H]) D0[H] = D1[H] else D0[H] = D2[H]
MAX.HU D0, D1, D2	if (D1[H] > D2[H]) D0[H] = D1[H] else D0[H] = D2[H]
MIN.B D0, D1, D2	if (D1[B] < D2[B]) D0[B] = D1[B] else D0[B] = D2[B]
MIN.BU D0, D1, D2	if (D1[B] < D2[B]) D0[B] = D1[B] else D0[B] = D2[B]
MAX.B D0, D1, D2	if (D1[B] > D2[B]) D0[B] = D1[B] else D0[B] = D2[B]
MAX.BU D0, D1, D2	if (D1[B] > D2[B]) D0[B] = D1[B] else D0[B] = D2[B]
SAT.H D0, D1	if (D1 < 0xFFFF8000) D0 = 0xFFFF8000; if (D1 > 0x7FFF) D0 = 0x00007FFF
SAT.HU D0, D1	if (D1 > 0xFFFF) D0 = 0x0000FFFF
SAT.B D0, D1	if (D1 < 0xFFFFF80) D0 = 0xFFFFF80; if (D1 > 0x7F) D0 = 0x0000007F
SAT.BU D0, D1	if (D1 > 0xFF) D0 = 0x000000FF

14.1.5 Mul (Integer) Multiplication

MUL(S) D0, D1, D2 MUL E0, D2, D3	D0 = D1 * D2 E0 = D2 * D3	32 = 32*32 64 = 32*32
MUL.U ¹⁾ D0,D1,D2 MUL.U E0, D2, D3	D0 = D1 * D2 E0 = D2 * D3	32 = 32*32 64 = 32*32
MULS.U D0, D1, D2	D0 = D1 * D2	32 = 32*32

¹⁾ MUL.U D0,D1,D2 does not exist; it is equivalent to MUL D0,D1,D2

14.1.6 Madd, Msub (Integer) MAC

MADD(S) D0, D1, D4, D5 MADD(S) E0, E2, D4, D5	D0 = D1 + (D4 * D5) E0 = E2 + (D4 * D5)	32 + = (32*32) 64 + = (32*32)
MADD(S).U D0, D1, D4, D5 MADD(S).U E0, E2, D4, D5	D0 = D1 + (D4 * D5) E0 = E2 + (D4 * D5)	32 + = (32*32) 64 + = (32*32)
MSUB(S) D0, D1, D4, D5 MSUB(S) E0, E2, D4, D5	D0 = D1 - (D4 * D5) E0 = E2 - (D4 * D5)	32 - = (32*32) 64 - = (32*32)
MSUB(S).U D0, D1, D4, D5 MSUB(S).U E0, E2, D4, D5	D0 = D1 - (D4 * D5) E0 = E2 - (D4 * D5)	32 - = (32*32) 64 - = (32*32)

14.1.7 Division

DVINIT E0, D5, D4	E0 = divide_init(D5,D4)	E0 = extended dividend
DVINIT.U E0, D5, D4	E0 = divide_init_u(D5,D4)	E2 = quotient and remainder
DVINIT.B E0, D5, D4	E0 = divide_init_b(D5,D4)	D2 = quotient
DVINIT.BU E0, D5, D4	E0 = divide_init_bu(D5,D4)	D3 = remainder
DVINIT.H E0, D5, D4	E0 = divide_init_h(D5,D4)	D4 = divisor
DVINIT.HU E0, D5, D4	E0 = divide_init_hu(D5,D4)	D5 = dividend
DVSTEP E2, E0, D4	E2 = divide_step(E0,D4)	
DVSTEP.U E2, E0, D4	E2 = divide_step(E0,D4)	
DVADJ E2, E0, D4	E2 = divide_adjust(E0,D4)	

Divide_init checks if the division is possible:

- 32-bit quotient value requires 4 DVSTEP (+ DVADJ if signed division)
- 16-bit quotient value requires 2 DVSTEP (+ DVADJ if signed division)
- 8-bit quotient value requires 1 DVSTEP (+ DVADJ if signed division e)

14.1.8 Sel, Cmov, Cadd, Csub

	Condition	True	False
SEL D3,D0,D1,D2	D0 != 0	D3 = D1	D3 = D2
SELN D3,D0,D1,D2	D0 != 0	D3 = D2	D3 = D1
CMOV D0,D15,D1	D15 != 0	D0 = D1	D0 = D0
CMOVN D0,D15,D1	D15 != 0	D0 = D0	D0 = D1
CADD D3,D0,D1,D2	D0 != 0	D3 = D1+D2	D3 = D1
CADDN D3,D0,D1,D2	D0 != 0	D3 = D1	D3 = D1 + D2
CSUB D3,D0,D1,D2	D0 != 0	D3 = D1 - D2	D3 = D1
CSUBN D3,D0,D1,D2	D0 != 0	D3 = D1	D3 = D1 - D2

14.1.9 Logical

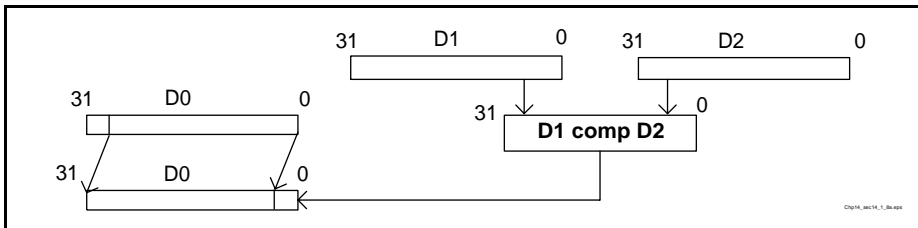
AND D0,D1,D2	D0 = (D1 AND D2)
OR D0,D1,D2	D0 = (D1 OR D2)
XOR D0,D1,D2	D0 = (D1 XOR D2)
ANDN D0,D1,D2	D0 = (D1 AND !D2)
ORN D0,D1,D2	D0 = (D1 OR !D2)
NAND D0,D1,D2	D0 != (D1 AND D2)
NOR D0,D1,D2	D0 != (D1 OR D2)
XNOR D0,D1,D2	D0 != (D1 XOR D2)

14.1.10 Compare

	Condition	True	False
EQ D0,D1,D2	D1 == D2	D0 = 0x00000001	D0 = 0x00000000
EQ.B D0,D1,D2	D1[B] == D2[B]	D0[B] = 0xFF	D0[B] = 0x00
EQ.H D0,D1,D2	D1[H] == D2[H]	D0[H] = 0xFFFF	D0[H] = 0x0000
EQ.W D0,D1,D2	D1 == D2	D0 = 0xFFFFFFFF	D0 = 0x00000000
LT (.U) D0,D1,D2	D1 < D2	D0 = 0x00000001	D0 = 0x00000000
LT.B (.BU) D0,D1,D2	D1[B] < D2[B]	D0[B] = 0xFF	D0[B] = 0x00
LT.H (.HU) D0,D1,D2	D1[H] < D2[H]	D0[H] = 0xFFFF	D0[H] = 0x0000
LT.W (.WU) D0,D1,D2	D1 < D2	D0 = 0xFFFFFFFF	D0 = 0x00000000
GE (.U) D0,D1,D2	D1 >= D2	D0 = 0x00000001	D0 = 0x00000000
NE D0,D1,D2	D1 != D2	D0 = 0x00000001	D0 = 0x00000000
EQANY.H D0,D1,D2	(D1[31:16]==D2[31:16]) or (D1[15:0]==D2[15:0])	D0 = 0x00000001	D0 = 0x00000000
EQANY.B D0,D1,D2	(D1[31:24]==D2[31:24]) or (D1[23:16]==D2[23:16]) or (D1[15:8]==D2[15:8]) or (D1[7:0]==D2[7:0])	D0 = 0x00000001	D0 = 0x00000000

Compare and Shift

SH.EQ D0,D1,D2	D0 = (D0[SH1], (if (D1 == D2) D0[0] = 1 else 0))
SH.GE D0,D1,D2	D0 = (D0[SH1], (if (D1 >= D2) D0[0] = 1 else 0))
SH.GE.U D0,D1,D2	D0 = (D0[SH1], (if (D1 >= D2) D0[0] = 1 else 0))
SH.LT D0,D1,D2	D0 = (D0[SH1], (if (D1 < D2) D0[0] = 1 else 0))
SHLT.U D0,D1,D2	D0 = (D0[SH1], (if (D1 < D2) D0[0] = 1 else 0))
SH.NE D0,D1,D2	D0 = (D0[SH1], (if (D1 != D2) D0[0] = 1 else 0))



Compare and Logical

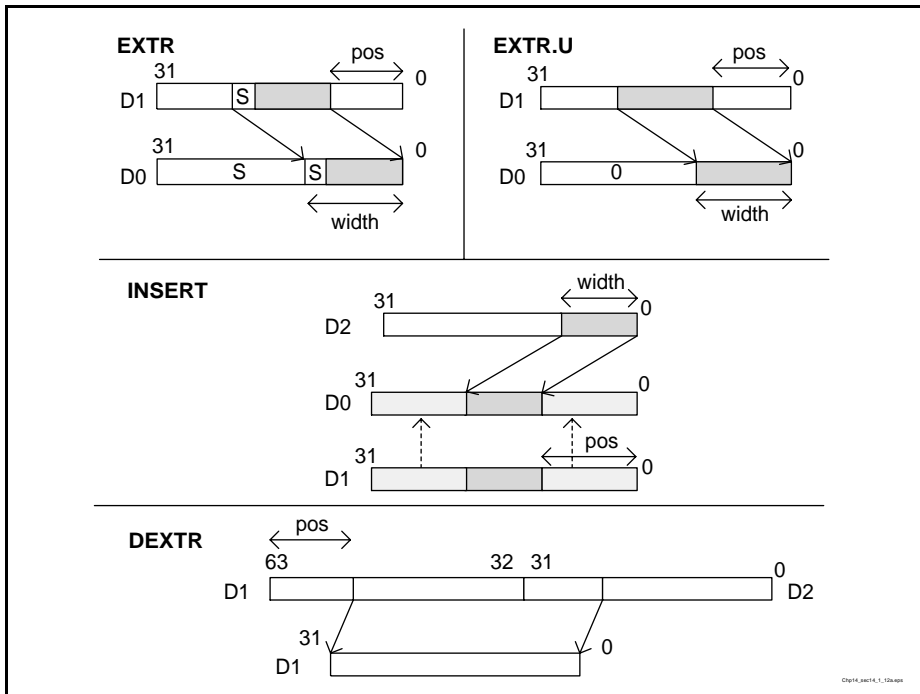
AND.EQ D0, D1, D2	$D0[0] = (D0[0] \text{ AND } (D1 == D2))$	$D0[31:1] = 0$
AND.GE D0, D1, D2	$D0[0] = (D0[0] \text{ AND } (D1 \geq D2))$	$D0[31:1] = 0$
AND.GE.U D0, D1, D2	$D0[0] = (D0[0] \text{ AND } (D1 \geq D2))$	$D0[31:1] = 0$
AND.LT D0, D1, D2	$D0[0] = (D0[0] \text{ AND } (D1 < D2))$	$D0[31:1] = 0$
AND.LT.U D0, D1, D2	$D0[0] = (D0[0] \text{ AND } (D1 < D2))$	$D0[31:1] = 0$
AND.NE D0, D1, D2	$D0[0] = (D0[0] \text{ AND } (D1 \neq D2))$	$D0[31:1] = 0$
OR.EQ D0, D1, D2	$D0[0] = (D0[0] \text{ OR } (D1 == D2))$	$D0[31:1] = 0$
OR.GE D0, D1, D2	$D0[0] = (D0[0] \text{ OR } (D1 \geq D2))$	$D0[31:1] = 0$
OR.GE.U D0, D1, D2	$D0[0] = (D0[0] \text{ OR } (D1 \geq D2))$	$D0[31:1] = 0$
OR.LT D0, D1, D2	$D0[0] = (D0[0] \text{ OR } (D1 < D2))$	$D0[31:1] = 0$
OR.LT.U D0, D1, D2	$D0[0] = (D0[0] \text{ OR } (D1 < D2))$	$D0[31:1] = 0$
OR.NE D0, D1, D2	$D0[0] = (D0[0] \text{ OR } (D1 \neq D2))$	$D0[31:1] = 0$
XOR.EQ D0, D1, D2	$D0[0] = (D0[0] \text{ XOR } (D1 == D2))$	$D0[31:1] = 0$
XOR.GE D0, D1, D2	$D0[0] = (D0[0] \text{ XOR } (D1 \geq D2))$	$D0[31:1] = 0$
XOR.GE.U D0, D1, D2	$D0[0] = (D0[0] \text{ XOR } (D1 \geq D2))$	$D0[31:1] = 0$
XOR.LT D0, D1, D2	$D0[0] = (D0[0] \text{ XOR } (D1 < D2))$	$D0[31:1] = 0$
XOR.LT.U D0, D1, D2	$D0[0] = (D0[0] \text{ XOR } (D1 < D2))$	$D0[31:1] = 0$
XOR.NE D0, D1, D2	$D0[0] = (D0[0] \text{ XOR } (D1 \neq D2))$	$D0[31:1] = 0$

		example D2 = -4 D1 = 11010001111100101010000011110101
SH D0, D1, D2	D0= D1<<D2 (D2 positive) D0= D1>>D2 (D2 negative)	D0 = 0000 1101000111110010101000001111
SHA D0, D1, D2		D0 = 1111 11010001111100101010 00001111
SHAS D0, D1, D2		D0 = 1111 11010001111100101010 00001111
SH.H D0, D1, D2	D0[H]= D1[H]<<D2 (D2 positive) D0[H]= D1[H]>>D2 (D2 negative)	D0 = 0000 110100011111 0000 101000001111
SHA.H D0, D1, D2		D0 = 1111 110100011111 1111 101000001111

Range = [-32;31] for SH,SHA,SHAS ; [-16;15] for SH.H, SHA.H.

14.1.12 Extract, Insert

EXTR D0, D1, D2, 16 EXTR D0, D1, E0	D0 = D1, position, width D0 = D1, E0(up), E0(lo)	sign_ext
EXTR.U D0, D1, D2, 16 EXTR.U D0, D1, E0	D0 = D1, position, width D0 = D1, E0(up), E0(lo)	zero_ext
INSERT D0, D1, D2, D3, 16 INSERT D0, D1, D2, E0	D0 = D1, D2, position, width D0 = D1, D2, E0(up), E0(lo)	
DEXTR D0, D1, D2, D3	D0 = D1, D2, position	



CG014, 304714_1_12a.pdf

14.1.13 Count Leading

CLZ D0, D1	D0 = Leading_zeros(D1)	D1= <u>0000000</u> 1110100011010011110010101 D0= 0x7
CLZ.H D0, D1	D0[H]= Leading_zeros(D1[H])	D1= <u>0000000</u> 111010001 1010011110010101 D0= 0x00070000
CLO D0, D1	D0 = Leading_ones(D1)	D1= <u>1111</u> 0001110100011010011110010101 D0= 0x4
CLO.H D0, D1	D0[H] = Leading_ones(D1[H])	D1= <u>1111</u> 000111010001 <u>1</u> 010011110010101 D0= 0x00040001
CLS D0, D1	D0 = Leading_signs(D1) - 1	D1= <u>0000000</u> 1110100011111110010101000 D0= 0x6
CLS.H D0, D1	D0[H]= Leading_signs(D1[H])-1	D1= <u>0000000</u> 111010001 <u>11111</u> 10010101000 D0= 0x00060005

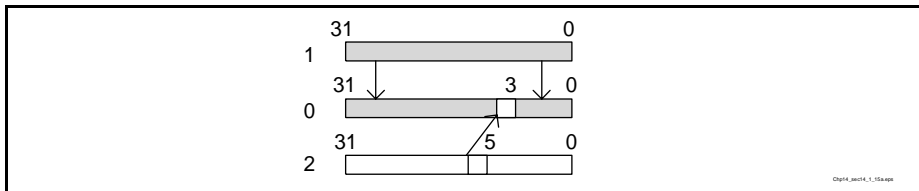
14.1.14 Bit split; Bit Merge; Parity

BSPLIT E0, D2	D1[31:16] = D0[31:16] = 0 D1[n/2] = D2[n+1] D0[n/2] = D2[n]	n = 0, 2,...30
BMERGE D2, D1, D0	D2[n] = ((n%2)? D1[n/2]: D0[n/2])	n = 0..31
PARITY D1, D0	D1[B] = Parity(D2[B])	n = 0, 8, 16, 24

14.1.15 Boolean Processing

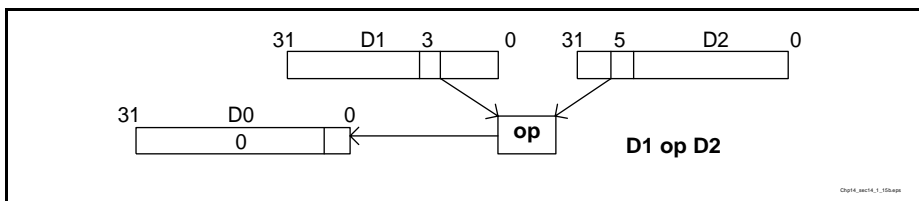
Bit Insert

INS.T D0, D1: 3, D2: 5	D0[31:4]=D1[31:4] D0[3]=D2[5] D0[2:0]=D1[2:0]
INSN.T D0, D1: 3, D2: 5	D0[31:4]=D1[31:4] D0[3]=!D2[5] D0[2:0]=D1[2:0]



2 Operand

AND.T D0, D1: 3, D2: 5	D0[0] = (D1[3] AND D2[5])	D0[31:1]= 0
NAND.T D0, D1: 3, D2: 5	D0[0] = !(D1[3] AND D2[5])	D0[31:1]= 0
OR.T D0, D1: 3, D2: 5	D0[0] = (D1[3] OR D2[5])	D0[31:1]= 0
NOR.T D0, D1: 3, D2: 5	D0[0] = !(D1[3] OR D2[5])	D0[31:1]= 0
XOR.T D0, D1: 3, D2: 5	D0[0] = (D1[3] XOR D2[5])	D0[31:1]= 0
XNOR.T D0, D1: 3, D2: 5	D0[0] = !(D1[3] XOR D2[5])	D0[31:1]= 0
ANDN.T D0, D1: 3, D2: 5	D0[0] = (D1[3] AND !D2[5])	D0[31:1]= 0
ORN.T D0, D1: 3, D2: 5	D0[0] = (D1[3] OR !D2[5])	D0[31:1]= 0

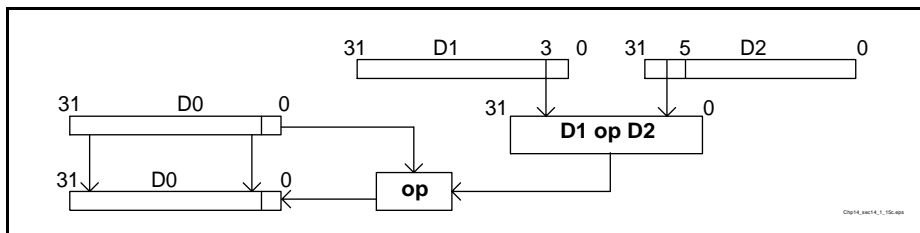


2 Operand with Shift

SH.AND.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = D1[3] \text{ AND } D2[5]$
SH.ANDN.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = D1[3] \text{ AND } !D2[5]$
SH.NAND.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = !(D1[3] \text{ AND } D2[5])$
SH.OR.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = D1[3] \text{ OR } D2[5]$
SH.ORN.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = D1[3] \text{ OR } !D2[5]$
SH.NOR.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = !(D1[3] \text{ OR } D2[5])$
SH.XOR.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = D1[3] \text{ XOR } D2[5]$
SH.XNOR.T D0, D1: 3, D2: 5	$D0[31:1]=D0[30:0] \quad D0[0] = !(D1[3] \text{ XOR } D2[5])$

3 Operand

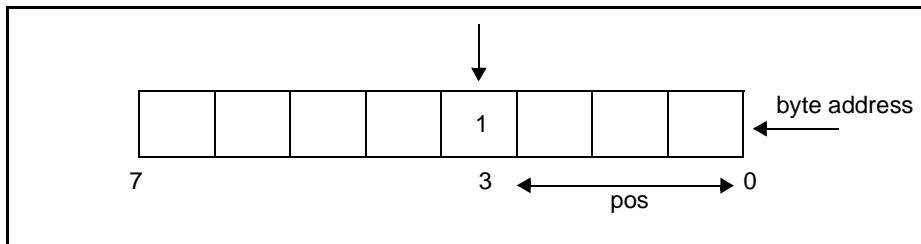
AND.AND.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ AND } (D1[3] \text{ AND } D2[5])) \quad D0[31:1]=0$
AND.ANDN.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ AND } (D1[3] \text{ ANDN } D2[5])) \quad D0[31:1]=0$
AND.OR.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ AND } (D1[3] \text{ OR } D2[5])) \quad D0[31:1]=0$
AND.NOR.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ AND } (D1[3] \text{ NOR } D2[5])) \quad D0[31:1]=0$
OR.AND.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ OR } (D1[3] \text{ AND } D2[5])) \quad D0[31:1]=0$
OR.ANDN.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ OR } (D1[3] \text{ ANDN } D2[5])) \quad D0[31:1]=0$
OR.OR.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ OR } (D1[3] \text{ OR } D2[5])) \quad D0[31:1]=0$
OR.NOR.T D0, D1: 3, D2: 5	$D0 = (D0[0] \text{ OR } (D1[3] \text{ NOR } D2[5])) \quad D0[31:1]=0$



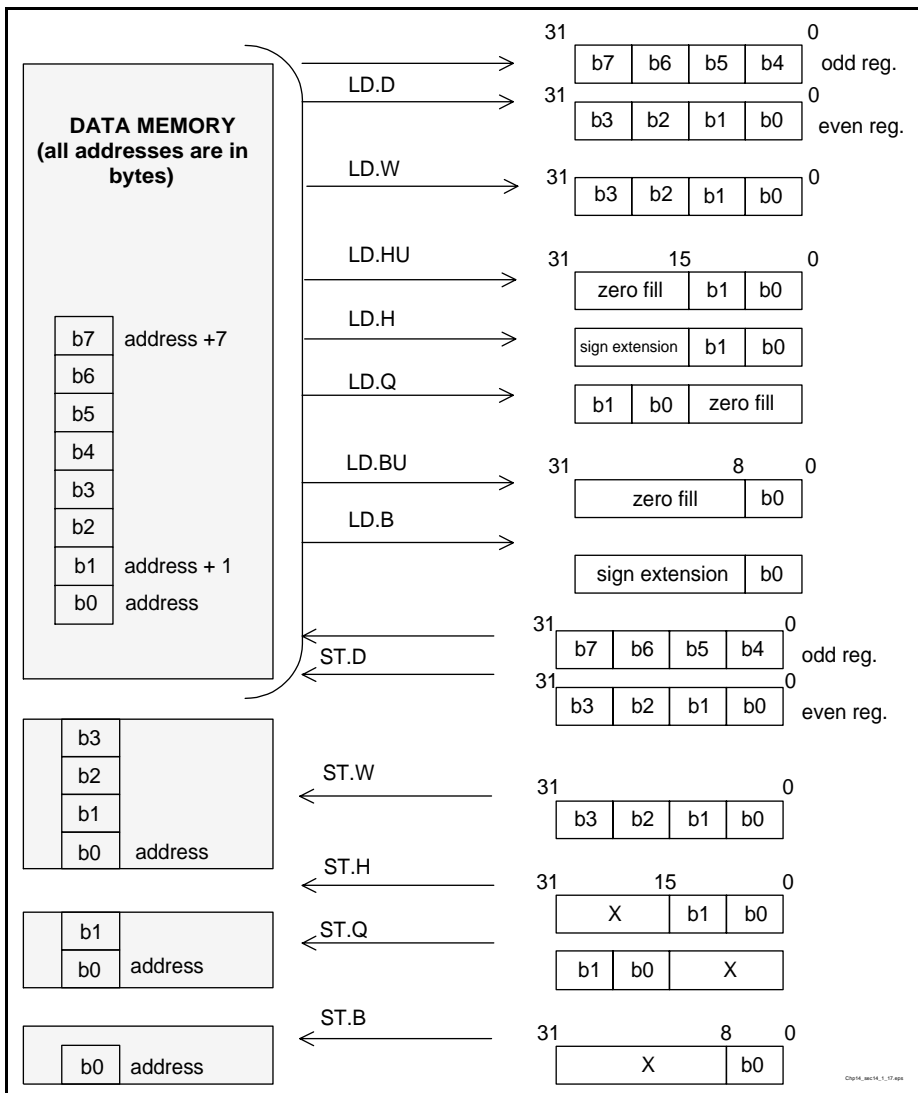
14.1.16 Bit Store, Load/Modify/Store

ST.T <mode>,3,1	ST.T destination, bit position, bit value
-----------------	---

Store a '1' at position '3' , at memory address byte (given by the <mode>):



14.1.17 Memory Load / Store



14.1.18 Address Arithmetic

ADD.A A2, A3, A4	$A2 = A3 + A4$
SUB.A A2, A3, A4	$A2 = A3 - A4$
ADDIH.A A2, A3, K16	$A2[up] = A3[up] + K16; A2[lo] = A3[lo]$
ADDSC.A A2, A3, D1, n	$A2 = A3 + (D1 \ll n) ; n = 0..3$
ADDSC.AT A2, A3, D1	$A2 = (A3 + (D1 \gg 3)) \text{ AND } 0xFFFFFFFFC$
EQ.A D0, A3, A4	if $(A3 == A4)$ $D0 = 0x1$ else $D0 = 0$
EQZ.A D0, A3	if $(A3 == 0)$ $D0 = 0x1$ else $D0 = 0$
GE.A D0, A3, A4	if $(A3 \geq A4)$ $D0 = 0x1$ else $D0 = 0$
LT.A D0, A3, A4	if $(A3 < A4)$ $D0 = 0x1$ else $D0 = 0$
NE.A D0, A3, A4	if $(A3 \neq A4)$ $D0 = 0x1$ else $D0 = 0$
NEZ.A D0, A3, A4	if $(A3 \neq 0)$ $D0 = 0x1$ else $D0 = 0$
MOV.A A2, D1	$A3 = D1$
MOV.AA A2, A3	$A3 = A2$
MOVH.A A2, K16	$A2[up] = K16 ; A2[lo] = 0x0$
LEA A2, <mode>	$A2 = EA$ (address register = effective address)

14.1.19 Jump and Loop

Name	Condition	PC =	Auto-dec / increment
JA branch	Always	effective address	
J branch	Always	PC + signext(2*disp8)	
JI A2	Always	A2[31..1] 0	
JEQ D0,D1,branch	D0 == D1	PC + signext(2*disp15)	
JNE D0,D1,branch	D0 != D1	PC + signext(2*disp15)	
JLT D0,D1,branch	D0 < D1	PC + signext(2*disp15)	
JGE D0,D1,branch	D0 >= D1	PC + signext(2*disp15)	
JLT.U D0,D1,branch	D0 < D1	PC + signext(2*disp15)	
JGE.U D0,D1,branch	D0 >= D1	PC + signext(2*disp15)	
JZ D15, branch	D15 = 0	PC + signext(2*disp8)	
JNZ D15, branch	D15 != 0	PC + signext(2*disp8)	
JLTZ D2, branch	D2 < 0	PC + zeroext(2*disp4)	
JLEZ D2, branch	D2 <= 0	PC + zeroext(2*disp4)	
JGTZ D2, branch	D2 > 0	PC + zeroext(2*disp4)	
JGEZ D2, branch	D2 >= 0	PC + zeroext(2*disp4)	
JEQ.A A0, A1, branch	A0 == A1	PC + signext(2*disp15)	
JNE.A A0, A1, branch	A0 != A1	PC + signext(2*disp15)	
JZ.A A2, branch	A2 = 0	PC + signext(2*disp15)	
JNZ.A A2, branch	A2 != 0	PC + signext(2*disp15)	
JZ.T D0: 9, branch	D0[9]=0	PC + signext(2*disp15)	
JNZ.T D0: 9, branch	D0[9]!=0	PC + signext(2*disp15)	
JNEI D0, D1, branch	D0 != D1	PC + signext(2*disp15)	D0 = D0+ 1
JNED D0, D1, branch	D0 != D1	PC + signext(2*disp15)	D0 = D0- 1

LOOP A2, aloop	A2 != 0	PC + signext(2*disp15)	A2 = A2- 1
LOOPU aloop	Always	PC + signext(2*disp15)	

PC + zeroext(2*disp4) JZ , JNZ D2, branch

14.1.20 Call, Return, Context Switch

Name	A11 =	PC =
JLA branch	PC +4	effective address
JL branch	PC +4	PC + signext(2*disp24)
JLI A2	PC +4	A2[31..1] 0
CALLA branch	PC +4	effective address
CALL branch	PC +4	PC + signext(2*disp24)
CALL branch	PC +4	PC + signext(2*disp8)
CALLI A2	PC +4	A2[31..1] 0
RET	PC +4	A11
RFE	PC +4	A11

Name	Functions	Saved Registers
LDLCX <mode>	Load lower context	A2..A7, D0..D7
STLCX <mode>	Store lower context	A2..A7, D0..D7 , PSW, PCXI
LDUCX <mode>	Load upper context	A10..A15 , D8..D15
STUCX <mode>	Store upper context	A10..A15, D8..D15 and PSW, PCXI
SVLCX	Save lower context	A2..A7, D0..D7 , A11
RSLCX	Restore lower context	A2..A7, D0..D7, A11

14.1.21 Q-format MUL

MUL.Q D0, D1u, D2u, n	$D0 = (D1[up] * D2[up]) \ll n$
MUL.Q D0, D1l, D2l, n	$D0 = (D1[lo] * D2[lo]) \ll n$
MUL.Q D0, D1, D2u, n	$D0 = (D1 * D2[up]) \ll n$
MUL.Q D0, D1, D2l, n	$D0 = (D1 * D2[lo]) \ll n$
MUL.Q D0, D1, D2, n	$D0 = (D1 * D2) \ll n$
MUL.Q E0, D2, D3u, n	$E0 = (D2 * D3[up]) \ll n$
MUL.Q E0, D2, D3l, n	$E0 = (D2 * D3[lo]) \ll n$
MUL.Q E0, D2, D3, n	$E0 = (D2 * D3) \ll n$
MULR.Q D0, D1u, D2u, n	$D0 = \text{round}((D1[up] * D2[up]) \ll n)$
MULR.Q D0, D1l, D2l, n	$D0 = \text{round}((D1[lo] * D2[lo]) \ll n)$

n=0,1

14.1.22 Q-format MAC

MADD(S).Q D0, D1, D4u, D5u, n (note 1: exists in 9 other variants)	$D0 = D1 + (D4[up] * D5[up]) \ll n$	$32 = 32 + 16 * 16$
MADDR(S).Q D0, D1, D4u, D5u, n (note 2: exists in 1 other variant)	$D0 = \text{round}(D1 + (D4[up] * D5[up]) \ll n)$	$16 = 32 + 16 * 16$

n=0,1

Note: 1) MADD(S).Q 9 other variants

MADD(S).Q D0, D1, D4l, D5l, n	$D0 = D1 + (D4[lo] * D5[lo]) \ll n$	$32 += 16 * 16$
MADD(S).Q D0, D1, D4, D5, n	$D0 = D1 + (D4 * D5) \ll n$	$32 += 32 * 32$
MADD(S).Q D0, D1, D4, D5u, n	$D0 = D1 + (D4 * D5[up]) \ll n$	$32 += 32 * 16$
MADD(S).Q D0, D1, D4, D5l, n	$D0 = D1 + (D4 * D5[lo]) \ll n$	$32 += 32 * 16$
MADD(S).Q E0, E2, D4l, D5l, n	$E0 = E2 + (D4[lo] * D5[lo]) \ll 16+n$	$48 += 16 * 16$
MADD(S).Q E0, E2, D4u, D5u, n	$E0 = E2 + (D4[up] * D5[up]) \ll 16+n$	$48 += 16 * 16$
MADD(S).Q E0, E2, D4, D5, n	$E0 = E2 + (D4 * D5) \ll n$	$64 += 32 * 32$
MADD(S).Q E0, E2, D4, D5u, n	$E0 = E2 + (D4 * D5[up]) \ll n$	$64 += 32 * 16$
MADD(S).Q E0, E2, D4, D5l, n	$E0 = E2 + (D4 * D5[lo]) \ll n$	$64 += 32 * 16$

n=0,1

Note: 2) MADDR(s).Q variant with ll

MADDR(S).Q D0, D1, D4l, D5l, n	$D0 = \text{round}(D1 + (D4[lo] * D5[lo]) \ll n)$	$16 = 32 + 16 * 16$
--------------------------------	---	---------------------

MSUB

MSUB is totally orthogonal with MADD. It has the 2 same main instructions as MADD (indicated below) and all the same derived variants as MADD (not shown)

MSUB(S).Q D0, D1, D4u, D5u, n	$D0 = D1 - (D4[up] * D5[up]) \ll n$	32 = 16*16
MSUBR(S).Q D0, D1, D4u, D5u, n	$D0 = \text{round}(D1 - (D4[up] * D5[up]) \ll n)$	16 = 32- 16*16

14.1.23 Q-format Dual MUL

There are 3 instructions main types:

- The 2 multiplication results are 32-bit
- The 2 multiplication results are 16-bit after rounding
- The 2 multiplication results are added on 48-bit

Each type has 3 variants, giving a total of 9 different instructions.

MUL.H E0, D2, D3ul, n (note 1 : exists in 3 other variants)	$D1 = (D2[up] * D3[up]) \ll n$ $D0 = (D2[lo] * D3[lo]) \ll n$	32 = 16*16 32 = 16*16
MULR.H D0, D1, D2ul, n (note 2 : exists in 3 other variants)	$D0[up] = \text{round}((D1[up] * D2[up]) \ll n)$ $D0[lo] = \text{round}((D1[lo] * D2[lo]) \ll n)$	16 = 16*16 16 = 16*16
MULM(S).H E0, D2, D3ul, n (note 3 : exists in 3 other variants)	$E0 = (D2[up] * D3[up] + D2[lo] * D3[lo]) \ll 16+n$	48 = 16*16 + 16*16

n=0,1

Note: 1) MUL.H lu,ll,uu variants

MUL.H E0, D2, D3lu n	$D1 = (D2[up] * D3[lo]) \ll n$ $D0 = (D2[lo] * D3[up]) \ll n$
MUL.H E0, D2, D5ll, n	$D1 = (D2[up] * D3[lo]) \ll n$ $D0 = (D2[lo] * D3[lo]) \ll n$
MUL.H E0, D2, D5uu, n	$D1 = (D2[lo] * D3[up]) \ll n$ $D0 = (D2[up] * D3[up]) \ll n$

Note: 2) MULR.H lu,ll,uu variants

MULR.H D0, D1, D2lu n	$D0[up] = \text{round}(D1[up] * D2[lo]) \ll n$ $D0[lo] = \text{round}(D1[lo] * D2[up]) \ll n$
-----------------------	--

MULR.H D0, D1, D2e, n	$D0[up] = \text{round}(D1[up] * D2[lo]) << n$ $D0[lo] = \text{round}(D1[lo] * D2[lo]) << n$
MULR.H D0, D1, D2uu, n	$D0[up] = \text{round}(D1[lo] * D2[up]) << n$ $D0[lo] = \text{round}(D1[up] * D2[up]) << n$

Note: 3) MULM(S).H lu,ll,uu variants

MULM(S).H E0, D2,D3lu, n	$E0 = (D2[up] * D3[lo]) + (D2[lo] * D3[up]) << 16+n$
MULM(S).H E0, D2,D3ll, n	$E0 = (D2[up] * D3[lo]) + (D2[lo] * D3[lo]) << 16+n$
MULM(S).H E0, D2,D3uu, n	$E0 = (D2[lo] * D3[up]) + (D2[up] * D3[up]) << 16+n$

14.1.24 Q-format Dual MAC

There are 3 main types of instruction: result on 48-bit, 32-bit and 16-bit. Each type has 16 variants, giving a total of 48 different instructions. The 16 variants are results of combining the 4 adder variants * 4 multiplication variants.

This has been simplified by showing only the adder variants giving 12 instructions (3 instruction types * 4 adder variants). It is necessary to replace the multi_alias to get the definition of prod1 and prod0.

multi_alias	Definition of the results (prod0, prod1) of the 2 multiplications
D4, D5ul, n	$\text{prod1} = (D4[up] * D5[up]) << n$ $\text{prod0} = (D4[lo] * D5[lo]) << n$
D4, D5lu, n	$\text{prod1} = (D4[up] * D5[lo]) << n$ $\text{prod0} = (D4[lo] * D5[up]) << n$
D4, D5ll, n	$\text{prod1} = (D4[up] * D5[lo]) << n$ $\text{prod0} = (D4[lo] * D5[lo]) << n$
D4, D5uu, n	$\text{prod1} = (D4[lo] * D5[up]) << n$ $\text{prod0} = (D4[up] * D5[up]) << n$

n=0,1

48-bit result

MADDM(S).H E0, E2, multi_alias	$E0 = E2 + (prod1 + prod0) \ll 16$	$48 += (16*16 + 16*16)$
MSUBM(S).H E0, E2, multi_alias	$E0 = E2 - (prod1 + prod0) \ll 16$	$48 -= (16*16 + 16*16)$
MADDSUM(S).H E0, E2, multi_alias	$E0 = E2 + (prod1 - prod0) \ll 16$	$48 += (16*16 - 16*16)$
MSUBADM(S).H E0, E2, multi_alias	$E0 = E2 - (prod1 - prod0) \ll 16$	$48 -= (16*16 - 16*16)$

32-bit result

MADD(S).H E0, E2, multi_alias	$D1 = D3 + prod1$ $D0 = D2 + prod0$	$32 += 16*16$ $32 += 16*16$
MSUB(S).H E0, E2, multi_alias	$D1 = D3 - prod1$ $D0 = D2 - prod0$	$32 -= 16*16$ $32 -= 16*16$
MADDSU(S).H E0, E2, multi_alias	$D1 = D3 + prod1$ $D0 = D2 - prod0$	$32 += 16*16$ $32 -= 16*16$
MSUBAD(S).H E0, E2, multi_alias	$D1 = D3 - prod1$ $D0 = D2 + prod0$	$32 -= 16*16$ $32 += 16*16$

16-bit result

MADDR(S).H D0, D1, multi_alias	$D0[up] = round(D1[up] + prod1)$ $D0[lo] = round(D1[lo] + prod0)$	$16 += 16*16$ $16 += 16*16$
MSUBR(S).H D0, D1, multi_alias	$D0[up] = round(D1[up] - prod1)$ $D0[lo] = round(D1[lo] - prod0)$	$16 -= 16*16$ $16 -= 16*16$
MADDSUR(S).H D0, D1, multi_alias	$D0[up] = round(D1[up] + prod1)$ $D0[lo] = round(D1[lo] - prod0)$	$16 += 16*16$ $16 -= 16*16$
MSUBADR(S).H D0, D1, multi_alias	$D0[up] = round(D1[up] - prod1)$ $D0[lo] = round(D1[lo] + prod0)$	$16 -= 16*16$ $16 += 16*16$

Note: These 2 MADDR instructions only exist with the ul variant.

MADDR(S).H D0, E2, D4, D5ul, n	$D0[up] = \text{round}(D3 + ((D4[up]*D5[up]) \ll n)$ $D0[lo] = \text{round}(D2 + ((D4[lo]*D5[lo]) \ll n)$	$16 = 32 + 16*16$ $16 = 32 + 16*16$
MSUBR(S).H D0, E2, D4, D5ul, n	$D0[up] = \text{round}(D3 - ((D4[up]*D5[up]) \ll n)$ $D0[lo] = \text{round}(D2 - ((D4[lo]*D5[lo]) \ll n)$	$16 = 32 - 16*16$ $16 = 32 - 16*16$

14.1.25 Floating Point Help

PACK D0, E4, D1	$D0 = \text{Pack}(E4, D1)$
UNPACK E0, D1	$E0 = \text{Unpack}(D1)$

14.1.26 Cache Management

CACHEA.I A2, K8	Cache address, invalidate
CACHEA.W A2, K8	Cache address, writeback
CACHEA.WI A2, K8	Cache address, writeback and invalidate

14.1.27 MMU Management

TLBMAP E0	Install mapping in MMU
TLBDEMAP D0	Uninstall mapping from MMU
TLBFLUSH.A	Flush mapping from MMU
TLBFLUSH.B	Flush mapping from MMU
TLBPROBE.A D0	Probe address in MMU ; uses virtual address
TLBPROBE.I D0	Probe address in MMU ; uses index

14.1.28 Exceptions (Interrupts, Traps, System Calls, Debug)

ENABLE	Enable interrupts
DISABLE	Disable interrupts
BISR K8	Begin ISR
SYSCALL K8	System call
TRAPV	Trap on overflow
TRAPSV	Trap on sticky overflow
DEBUG	Debug

14.1.29 Core Register and PSW Management

MFCR D0, K16	Move from core register
MTCR K16, D2	Move to core register
RSTV	Reset overflow

14.1.30 Pipeline Control

DSYNC	Synchronize data
ISYNC	Synchronize instructions
NOP	No operation

14.1.31 16-bit Opcode Instructions

Arithmetic, logic, shift, compare	Arithmetic, logic, shift, compare	Arithmetic, logic, shift, compare
ADD D1, D2	CMOVN D1, D15, D2	SAT.B D1
ADD D15, D1, D2	CMOVN D1, D15, K4	SAT.BU D1
ADD D1, D15, D2	EQ D1, D15, D2	SAT.H D1
ADD D1, K4	EQ D1, D15, K4	SAT.HU D1
ADD D15, D1, K4	MOV D1, K4	SH D1, K4
ADD.A A3, A4	MOV D15, K8	SHA D1, K4
ADD.A A3, K4	MOV D1, D2	SUB D1, D2
ADDSC.A A2, A3, D15, n	MOV.A A3, D2	SUB D15, D1, D2
AND D1, D2	MOV.AA A3, A4	SUB D1, D15, D2
AND D15, K8	NOR D1	SUBS D1, D2
CADD D1, D15, K4	OR D1, D2	SUB.A A10, K8
CADDN D1, D15, K4	OR D15, K8	XOR D1, D2
CMOV D1, D15, D2	RSUB D1	
CMOV D1, D15, K4		

D15 is the required source/destination register

14.1.32 Load & Store Instructions

Load / store	Address mode 1	Address mode 2	Address mode 3	Address Mode 4	Address mode 5
LD.BU		D2, [A3]	D2, [A15]offset4	D2, [A3]offset4	D2, [A3+1]
LD.H		D2, [A3]	D2, [A15]offset4	D2, [A3]offset4	D2, [A3+2]
LD.W	D2, [A3]offset16	D2, [A3]	D2, [A15]offset4	D2, [A3]offset4	D2, [A3+4]
LD.A	A4, [A3]offset16	A4, [A3]	A4, [A15]offset4	A15, [A3]offset4	A4, [A3+4]
ST.B		[A3], D2	[A15]offset4, D2	[A3]offset4, D2	[A3+1], D2
ST.H		[A3], D2	[A15]offset4, D2	[A3]offset4, D2	[A3+2], D2
ST.W	[A3]offset16, D2	[A3], D2	[A15]offset4, D2	[A3]offset4, D2	[A3+4], D2
ST.A		[A3], A4	[A15]offset4, A4	[A3]offset4, A15	[A3+4], A4

Addr. mode 1 = base + long offset

Addr. mode 2 = register indirect

Addr. mode 3 = implicit base register is A15

Addr. mode 4 = implicit source register

Addr. mode 5 = post-increment

14.1.33 Branch Instructions

Name	Condition	PC =
J	No	PC + signext(2*disp8)
JI	No	A2[31..1] 0
JZ	if D15 = 0	PC + signext(2*disp8)
JNZ	if D15 != 0	PC + zeroext(2*disp8)
JEQ	if D15 = D2	PC + zeroext(2*disp4)
JEQ	if D15 = K4	PC + zeroext(2*disp4)
JNE	if D15 = D2	PC + zeroext(2*disp4)
JNE	if D15 = K4	PC + zeroext(2*disp4)
JZ	if D2 = 0	PC + zeroext(2*disp4)
JNZ	if D2 != 0	PC + zeroext(2*disp4)
JLTZ	if D2 < 0	PC + zeroext(2*disp4)
JLEZ	if D2 <= 0	PC + zeroext(2*disp4)
JGTZ	if D2 > 0	PC + zeroext(2*disp4)
JGEZ	if D2 >= 0	PC + zeroext(2*disp4)
JZ.T	if D15[n] = 0	PC + zeroext(2*disp4)
JNZ.T	if D15[n] != 0	PC + zeroext(2*disp4)
JZ.A	if A2 = 0	PC + zeroext(2*disp4)
JNZ.A	if A2 != 0	PC + zeroext(2*disp4)
LOOP	if A2 != 0	PC + one-ext(2*disp4)
LOOPU	No	PC + signext(2*disp15)

14.1.34 Packed Arithmetic Instructions

Halfword Packed Instructions			
ADD(S).H	D0, D1, D2	MSUB(S).H	E0, E2, D4, D5, n
ADDS.HU	D0, D1, D2	MADDR(S).H	D0, E2, D4, D5, n
SUB(S).H	D0, D1, D2	MSUBR(S).H	D0, E2, D4, D5, n
SUBS.HU	D0, D1, D2	MADDM(S).H	E0, E2, D4, D5, n
ABS(S).H	D0, D1	MSUBM(S).H	E0, E2, D4, D5, n
ABSDIF(S).H	D0, D1, D2	MADDSU(S).H	E0, E2, D4, D5, n
MIN.H	D0, D1, D2	MSUBAD(S).H	E0, E2, D4, D5, n
MIN.HU	D0, D1, D2	MADDSUM(S).H	E0, E2, D4, D5, n
MAX.H	D0, D1, D2	MSUBADM(S).H	E0, E2, D4, D5, n
MAX.HU	D0, D1, D2	MADDSUR(S).H	D0, D1, D4, D5, n
SAT.H	D0, D1	MSUBADR(S).H	D0, D1, D4, D5, n
SAT.HU	D0, D1		
EQ.H	D0, D1, D2		
LT.H(U)	D0, D1, D2		
EQANY.H	D0, D1, D2		
SH.H	D0, D1, D2		
SHA.H	D0, D1, D2		
CLZ.H	D0, D1		
CLO.H	D0, D1		
CLS.H	D0, D1		
MUL.H	E0, D2, D3, n		
MULM.H	E0, D2, D3, n		
MULR.H	D0, D1, D2, n		
MADD(S).H	E0, E2, D4, D5, n		

Byte Packed Instructions	
ADD.B	D0, D1, D2
ADD.BU	D0, D1, D2
SUB.B	D0, D1, D2
SUB.BU	D0, D1, D2
ABS.B	D0, D1
ABSDIF.B	D0, D1, D2
MIN.B	D0, D1, D2
MIN.BU	D0, D1, D2
MAX.B	D0, D1, D2
MAX.BU	D0, D1, D2
SAT.B	D0, D1
SAT.BU	D0, D1
EQ.B	D0, D1, D2
LT.B(U)	D0, D1, D2
EQANY.B	D0, D1, D2

14.1.35 Constants

Arithmetic	Logic	Branch
ADDI D0, D1, K16S	EQ D0, D1, K9S	JEQ D0, K4, DISP15
ADDIH D0, D1, K16	NE D0, D1, K9S	JGE[.U] D0, K4, DISP15
MOV D0, K16S	LT D0, D1, K9S	JLT[.U] D0, K4, DISP15
MOV.U D0, K16Z	GE D0, D1, K9S	JNE D0, K4, DISP15
MOVH D0, K16	LT.U D0, D1, K9Z	JNED D0, K4, DISP15
MOV D15, K9S	GE.U D0, D1, K9Z	JNEI D0, K4, DISP15
ADD D0, D1, K9S	op.EQ D0, D1, K9S	
ADDS D0, D1, K9S	op.NE D0, D1, K9S	16-bit Instructions
ADDS.U D0, D1, K9S	op.LT D0, D1, K9S	ADD D0, K4
ADDX D0, D1, K9S	op.GE D0, D1, K9S	ADD D15, D1, K4
ADDC D0, D1, K9S	op.LT.U D0, D1, K9Z	ADD D1, D15, K4
RSUB D0, D1, K9S	op.GE.U D0, D1, K9Z	CADD{N} D0, D15, K4
RSUBS D0, D1, K9S	SH.EQ D0, D1, K9S	CMOV{N} D0, D15, K4
RSUBS.U D0, D1, K9S	SH.NE D0, D1, K9S	EQ D15, D1, K4
ABSDIF D0, D1, K9S	SH.LT D0, D1, K9S	MOV D0, K4
ABSDIFS D0, D1, K9S	SH.GE D0, D1, K9S	SH D0, K4
CADD(N) D0, D1, D2, K9S	SH.LT.U D0, D1, K9Z	SHA D0, K4
SEL(N) D0, D1, D2, K9S	SH.GE.U D0, D1, K9Z	AND D15, K8
MAX D0, D1, K9S	EQANY.{B,H} D0, D1, K9S	OR D15, K8
MIN D0, D1, K9S	AND D0, D1, K9Z	
MAX.U D0, D1, K9Z	OR D0, D1, K9Z	8-bit Constant used for Control
MIN.U D0, D1, K9Z	XOR D0, D1, K9Z	SH D0, D1, K8
	NAND D0, D1, K9Z	SHA D0, D1, K8

MAC	NOR D0, D1, K9Z	SHAS D0, D1, K8
MADD(S) D0, D1, D2, K9S	XNOR D0, D1, K9Z	SH.H D0, D1, K8
MADD(S) E0, E2, D4, K9S	ANDN D0, D1, K9Z	SHA.H D0, D1, K8
MADD(S).U D0, D1, D2, K9Z	ORN D0, D1, K9Z	INSERT D0, D1, D2, D3, K8
MADD(S).U E0, E2, D4, K9Z		INSERT D0, D1, D2, K8, K8
MSUB(S) D0, D1, D2, K9S	Address	EXTR [.U] D0, D1, D2, K8
MSUB(S) E0, E2, D4, K9S	ADDIH.A A2, A3, K16 (<i>K16 upper</i>)	EXTR [.U] D0, D1, K8, K8
MSUB(S).U D0, D1, D2, K9Z	MOVH.A A2, K16 (<i>K16 upper</i>)	DEXTR D0, D1, D2, K8
MSUB(S).U E0, E2, D4, K9Z	LEA A2, (K14 + 4bit segment)	IMASK E0, D2, D3, K8
MUL D0, D1, K9S	ADD.A A2, K4	IMASK E0, D2, K8, K8
MUL E0, D2, K9S	MOV.A A2, K4	
MUL.U E0, D2, K9Z	SUB.A SP, K8	
Bit Field	Store Bit	
INSERT D0, D1, K4, E2	IMASK E0, K4, D2, K8	

Note: 2 unsigned instructions (ADDS.U. RSUBS.U) have their constants sign-extended. There is no arithmetic reason for this, but it was a by-product of implementation. The user should consider these 2 instructions as having 8-bit constants.

14.1.36 Instructions Modifying a 64-bit Data Register

MUL E0, D2, D3	BSPLIT e0,d1,d2
MUL.U E0, D2, D3	LD.D E0, <mode>
MUL.H E0, D2, D3, n	DVINIT E0, D2, D3
MADD E0, E2, D4, D5	DVINIT.U E0, D2, D3
MADD(S).U E0, E2, D4, D5	DVINIT.B E0, D2, D3
MADD(S).Q E0, E2, D4, D5	DVINIT.BU E0, D2, D3
MADD(S).H E0, E2, D4, D5, n	DVINIT.H E0, D2, D3
MADDM(S).H E0, E2, D4, D5, n	DVINIT.HU E0, D2, D3
MADDSU(S).H E0, E2, D4, D5, n	DVSTEP E0, E2, D4
MADDSUM(S).H E0, E2, D4, D5, n	DVADJ E0, E2, D4
MSUB E0, E2, D4, D5	IMASK E0, D2,D4,K8
MSUB(S).U E0, E2, D4, D5	
MSUB(S).Q E0, E2, D4, D5	
MSUB(S).H E0, E2, D4, D5, n	
MSUBM(S).H E0, E2, D4, D5, n	
MSUBAD(S).H E0, E2, D4, D5, n	
MSUBADM(S).H E0, E2, D4, D5, n	

15 TriCore 2

TriCore 2 introduces new addressing modes and 15 new instructions. The syntax of each instruction is given below. None of these instructions touch or use the PSW flags.

Name	Syntax
Move 64-bit	mov Ec,Db
	mov Ec,Da,Db
	mov Ec,const16
	mov Ea,const4
xpose.h	xpose.h Ec,Da,Db
xpose.b	xpose.b Ec,Da,Db
ld.dd	ld.dd Ea/Ea+2,<mode>
st.dd	st.dd <mode>, Ea/Ea+2
jeq (5-bit disp)	jeq D15,Db,disp5
	jeq D15,const4,disp5
jne (5-bit disp)	jne D15,Db,disp5
	jne D15,const4,disp5
Fcall	fcall disp24
Fcalla	fcalla disp24
Fcalli	fcalli Aa
Swap mask	swpmask Ea,<mode>
Disable	disable Da
Restore	restore Da
Cachei.w	cachei.w <mode>
Cachei.iw	cachei.iw <mode>

15.1 Index Addressing Mode

The index addressing mode provides the ability to have a variable increment value. This is of benefit during variable stride inner loops.

Example:

```
ld.w      d0, [a0/a1+i]      ;load word indexed addressing mode
st.w      [a12/a13+i], d0    ;store word indexed addressing mode
```

The index addressing mode uses an address register pair to hold the required state.

<i>B</i>		<i>Aodd</i>
<i>M</i>	<i>I</i>	<i>Aeven</i>

- The even register is the base address of the array (B).
- The least-significant half of the odd register is the index into the array (I).
- The most-significant half is the modifier (M) which is added to I after every access.
- The effective address is B+I.
- The index (I) is post incremented and its new value is I + M.

All load and store instructions are able to use the new addressing mode, including the new LD.DD and ST.DD instructions in TC2. The exceptions are ST.T, SWAP.W and LDMST.

15.2 New Instructions

For this optimization guide, only instructions which would fit in their respective chapter are described:

- mov <e register> would go in [Chapter 2 Simple Arithmetic](#)
- xpose.h, xpose.b would go in [Chapter 6 Bit Field Operations](#)
- ld.dd, st.dd, swpmask would go in [Chapter 8 Pointer Arithmetic and Addressing Modes](#)
- jeq, jne (5-bit disp), fcall, fall, fcalli would go in [Chapter 9 Program Control and Context Switch](#)

15.2.1 MOV <E register> - Move 64-bit

The ability to load an immediate or register into an extended (64-bit) Register is useful for initialization, and for data management.

Example:

```

mov      e0,d5           ; e0 = sign_ext64(d5)
mov      e0,d6,d3        ; e0 = d6_d3 or in other words d1=d6 d0=d3
mov      e0,0x1234        ; e0 = 0x00000000000001234
mov      e0,0xE           ; e0 = 0xFFFFFFFFFFFFFFFE

```

15.2.2 XPOSE.H, XPOSE.B

A DSP transpose operation with two 32 bit sources and a 64 bit destination which shuffles half words. This operation can speed up various DSP algorithms. There is a performance gain for FFT, matrix transpose, and other DSP algorithms. This instruction can replace a six-instruction sequence. For compilers, this simplifies the automatic “SIMD-ization” of C loops over arrays of 16-bit values.

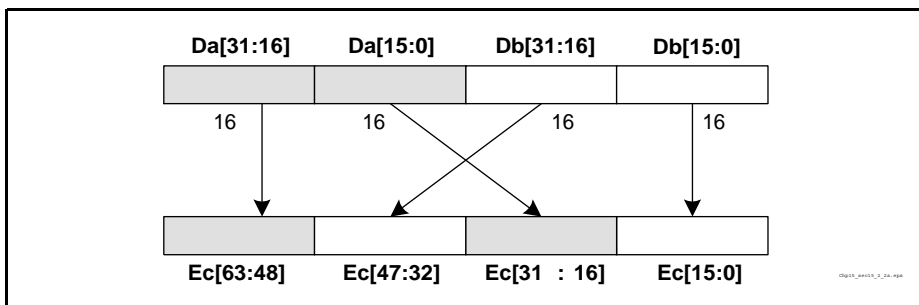
Transpose half words

Takes 2 data registers Da, Db and mixes them into a data register pair Ec. The left source operand supplies the most left 16 bits of the destination.

```

Ec[63:48] = Da[31:16]
Ec[47:32] = Db[31:16]
Ec[31:16] = Da[15:0]
Ec[15:0] = Db[15:0]

```

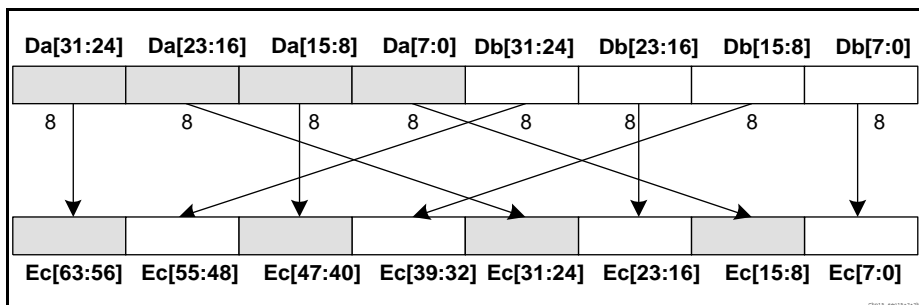


Transpose bytes:

Takes 2 data registers Da, Db, and mixes them into a data register pair Ec. The left source operand supplies the most left 8 bits of the destination.

$Ec[63:32] = \{Da[31:24], Db[31:24], Da[15:8], Db[15:8]\}$

$Ec[31:0] = \{Da[23:16], Db[23:16], Da[7:0], Db[7:0]\}$



Example:

```
xpose.h    e0,d2,d5      ; e0 = { d2[31:16], d5[31:16], d2[15:0], d5[15:0] }
xpose.b    e10,d1,d2     ;
```

15.2.3 LD.DD, ST.DD - Load/Store 128-bit

These instructions respectively load and store 4 data registers. These instructions speed up non-cached external accesses, to peripherals or non-cached memory for example, by using peripheral bus burst transfers.

Example:

```
ld.dd      e2/e4,[a0+]8   ; Load quad word
st.dd      [a0+]8,e8/e10  ; Store quad word
```

The registers Ea and Ea+2 are used for the operation. An operation with Ea = E14 causes a register wrap through E14, E0.

The following restrictions also applied:

- The source/destination registers are a series of 4 consecutive data registers where the index of the first register is a multiple of 2.
- There are some more stringent alignment restrictions that are implementation defined.

Note: These 2 instructions are available in all addressing modes except absolute and base +16-bit offset.

15.2.4 SWAPMSK - Semaphore Management

This instruction has been introduced to improve the ability to control remote semaphores, which was a shortcoming in the TC1.3 ISA definition. For TC 1.3, only the SWAP instruction can be used to create semaphores. Semaphores can only be 32-bit wide, but in most OS code 8 bit semaphores are assumed.

With this instruction, arrays of semaphore bits can be implemented, which allows more efficient packing of semaphores and, more importantly, efficient HW semaphore interaction with peripherals. Up to 32 semaphores (bits) can be tested in parallel.

Example:

```
swapmask    e0, [a0+4]
```

This instruction uses an E register pair to provide both mask and swap data. It swaps through a mask the contents of Ea (lower) with the memory contents. Only those bits are swapped where the corresponding bits in the mask Ea(upper) are set.

15.2.5 JNE,JEQ with 5-bit Branch Offset

The TC2.0 short conditional branches can use a zero-extended 5-bit offset field (TC1.3 was limited to 4-bit). The offset range supported by the 16-bit jump is therefore doubled.

Example:

```
jne         d15,d1,Error_jump
```

15.2.6 FCALL, FCALLA, FCALLI - Fast Call Context

The Fast Call allows a routine with low register pressure to be called with a limited context save set. This allows the context operations to occur more quickly and use less power.

In parallel with the Fast Call's implicit jump, the CPU saves the caller's Stack Pointer (A10), return address (A11), and PSW in an available Context Save Area (CSA), and then sets register A11 to the address of the next instruction beyond the call.

Example:

```
fcall        monitor_channel  
fcalli       a2
```

References

- Infineon Technologies Corporation
'TriCore Architecture Manual V1.3.3' (2002)
- Infineon Technologies Corporation
'TriCore Architecture Overview Handbook' (2001)
- R. Arnold
'RTOS Implementor's Guide for the TriCore Architecture' (1998)
- D. F. Martin & R. E. Owen
'A RISC architecture with uncompromised digital signal processing and microcontroller operations'
ICCASP, Seattle (1998)
- D. F. Martin
'Unique DSP architecture blends into the new Siemens 32-bit μ C/DSP TriCore family'
ICSPAT, Toronto (1998)
- C. Britton Rorabaugh
'DSP Primer' (1998)
- P.M.Embree & D.Dameli
'C++ Algorithms for Digital Signal Processing'
Prentice-Hall (1998)
- Motorola
'Fractional and Integer Arithmetic using the DSP56000 Family of General-Purpose Digital Signal Processors' (1993)
- Intel
'2920 Signal Processor Design Seminar Notebook' (1980)
- A. V. Oppenheim & R. W. Schaffer
'Discrete-time Signal Processing' (1989)
- R. J. Higgins
'Digital signal processing in VLSI' (1990)

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.

Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

<http://www.infineon.com>