

# **TMS320C66x DSP**

## **CPU and Instruction Set**

### **Reference Guide**



Literature Number: SPRUGH7  
November 2010

## Release History

Release	Date	Chapter/Topic	Description/Comments
	November 2010	All	Initial release based on <i>TMS320C674x CPU and Instruction Set Reference Guide</i> (SPRUFE8)

## Contents

---

<i>Release History</i> .....	ø-ii
<i>List of Tables</i> .....	ø-xvii
<i>List of Figures</i> .....	ø-xx
<i>List of Examples</i> .....	ø-xxiv

---

<i>Preface</i> .....	ø-XXV
----------------------	-------

About This Manual.....	ø-XXV
Notational Conventions.....	ø-XXV
Related Documentation from Texas Instruments.....	ø-XXVI
Trademarks.....	ø-XXVI

### Chapter 1

---

<i>Introduction</i> .....	1-1
1.1 DSP Features and Options.....	1-2
1.1.1 4x Multiply.....	1-3
1.1.2 Floating point support.....	1-3
1.1.3 Vector Processing.....	1-3
1.1.4 Complex arithmetic and matrix operations.....	1-4
1.2 DSP Architecture.....	1-5
1.2.1 Central Processing Unit (CPU) .....	1-6
1.2.2 Internal Memory .....	1-6
1.2.3 Memory and Peripheral Options.....	1-6

### Chapter 2

---

<i>CPU Data Paths and Control</i> .....	2-1
2.1 Introduction .....	2-2
2.2 General-Purpose Register Files .....	2-3
2.3 Functional Units .....	2-5
2.4 Register File Cross Paths.....	2-6
2.5 Memory, Load, and Store Paths.....	2-6
2.6 Data Address Paths .....	2-7
2.7 Galois Field .....	2-7
2.7.1 Special Timing Considerations.....	2-9
2.8 Control Register File .....	2-10
2.8.1 Register Addresses for Accessing the Control Registers.....	2-11
2.8.2 Pipeline/Timing of Control Register Accesses .....	2-11
2.8.3 Addressing Mode Register (AMR) .....	2-12
2.8.4 Control Status Register (CSR) .....	2-15
2.8.5 Galois Field Polynomial Generator Function Register (GFPGFR) .....	2-17
2.8.6 Interrupt Clear Register (ICR) .....	2-18
2.8.7 Interrupt Enable Register (IER) .....	2-19
2.8.8 Interrupt Flag Register (IFR) .....	2-20
2.8.9 Interrupt Return Pointer Register (IRP) .....	2-20
2.8.10 Interrupt Set Register (ISR) .....	2-21
2.8.11 Interrupt Service Table Pointer Register (ISTP) .....	2-22
2.8.12 Nonmaskable Interrupt (NMI) Return Pointer Register (NRP).....	2-23
2.8.13 E1 Phase Program Counter (PCE1).....	2-23
2.9 Control Register File Extensions .....	2-24
2.9.1 DSP Core Number Register (DNUM).....	2-24
2.9.2 Exception Clear Register (ECR).....	2-24
2.9.3 Exception Flag Register (EFR).....	2-25

2.9.4 GMPY Polynomial—A Side Register (GPLYA) .....	2-25
2.9.5 GMPY Polynomial—B Side Register (GPLYB) .....	2-26
2.9.6 Internal Exception Report Register (IERR).....	2-27
2.9.7 SPLOOP Inner Loop Count Register (ILC) .....	2-28
2.9.8 Interrupt Task State Register (ITSR) .....	2-28
2.9.9 NMI/Exception Task State Register (NTSR).....	2-29
2.9.10 Restricted Entry Point Register (REP).....	2-29
2.9.11 SPLOOP Reload Inner Loop Count Register (RILC) .....	2-30
2.9.12 Saturation Status Register (SSR) .....	2-30
2.9.13 Time Stamp Counter Registers (TSCL and TSCH) .....	2-31
2.9.13.1 Initialization.....	2-31
2.9.13.2 Enabling Counting .....	2-31
2.9.13.3 Disabling Counting .....	2-32
2.9.13.4 Reading the Counter.....	2-32
2.9.14 Task State Register (TSR) .....	2-33
2.10 Control Register File Extensions for Floating-Point Operations .....	2-34
2.10.1 Floating-Point Adder Configuration Register (FADCR).....	2-35
2.10.2 Floating-Point Auxiliary Configuration Register (FAUCR) .....	2-37
2.10.3 Floating-Point Multiplier Configuration Register (FMCR).....	2-40

## Chapter 3

<b>Instruction Set</b>	<b>3-1</b>
3.1 Instruction Operation and Execution Notation .....	3-2
3.2 Instruction Syntax and Opcode Notations .....	3-4
3.2.1 32-Bit Opcode Maps .....	3-5
3.2.2 16-Bit Opcode Maps .....	3-5
3.3 Overview of IEEE Standard Single- and Double-Precision Formats .....	3-6
3.3.1 Single-Precision Formats .....	3-7
3.3.2 Double-Precision Formats .....	3-8
3.4 Delay Slots.....	3-9
3.5 Parallel Operations.....	3-11
3.5.1 Example Parallel Code.....	3-13
3.5.2 Branching Into the Middle of an Execute Packet.....	3-13
3.6 Conditional Operations .....	3-14
3.7 SPMASKed Operations .....	3-15
3.8 Resource Constraints.....	3-15
3.8.1 Constraints on Instructions Using the Same Functional Unit .....	3-15
3.8.2 Constraints on the Same Functional Unit Writing in the Same Instruction Cycle .....	3-15
3.8.3 Constraints on Cross Paths (1X and 2X) .....	3-16
3.8.3.1 Maximum Number of Accesses to a Register on the Opposite Register File .....	3-16
3.8.4 Cross Path Sharing .....	3-16
3.8.5 Cross Path Stalls .....	3-17
3.8.6 Constraints on Loads and Stores.....	3-17
3.8.7 Constraints on Long (40-Bit) Data.....	3-18
3.8.8 Constraints on Register Reads .....	3-18
3.8.9 Constraints on Register Writes.....	3-18
3.8.10 Constraints on AMR Writes .....	3-19
3.8.11 Constraints on Multicycle NOPs .....	3-19
3.8.12 Constraints on Unitless Instructions .....	3-19
3.8.12.1 MFENCE Restrictions .....	3-19
3.8.12.2 SPLOOP Restrictions .....	3-20
3.8.12.3 BNOP <disp>,n.....	3-20
3.8.12.4 DINT.....	3-20
3.8.12.5 IDLE .....	3-21
3.8.12.6 NOP n .....	3-21
3.8.12.7 RINT .....	3-21
3.8.12.8 SPKERNEL(R).....	3-22

3.8.12.9 SPLOOP(D/W).....	3-22
3.8.12.10 SPMASK(R).....	3-22
3.8.12.11 SWE .....	3-22
3.8.12.12 SWENR .....	3-23
3.8.13 Constraints on Floating-Point Instructions .....	3-23
3.9 Addressing Modes .....	3-24
3.9.1 Linear Addressing Mode .....	3-25
3.9.1.1 LD and ST Instructions .....	3-25
3.9.1.2 ADDA and SUBA Instructions .....	3-25
3.9.2 Circular Addressing Mode .....	3-25
3.9.2.1 LD and ST Instructions .....	3-25
3.9.2.2 ADDA and SUBA Instructions .....	3-26
3.9.2.3 Circular Addressing Considerations with Nonaligned Memory .....	3-26
3.9.3 Syntax for Load/Store Address Generation .....	3-27
3.10 Compact Instructions on the CPU .....	3-29
3.10.1 Compact Instruction Overview .....	3-29
3.10.2 Header Word Format .....	3-30
3.10.2.1 Layout Field in Compact Header Word.....	3-30
3.10.2.2 Expansion Field in Compact Header Word .....	3-31
3.10.2.3 P-bit Field in Compact Header Word .....	3-32
3.10.3 Processing of Fetch Packets .....	3-33
3.10.4 Execute Packet Restrictions .....	3-33
3.10.5 Available Compact Instructions .....	3-34
3.11 Instruction Compatibility .....	3-35

## **Chapter 4**

<i>Instruction Descriptions</i>	4-1
4.1 Example .....	4-2
4.2 ABS.....	4-4
4.3 ABS2 .....	4-7
4.4 ABSDP.....	4-9
4.5 ABSSP .....	4-11
4.6 ADD .....	4-13
4.7 ADDAB .....	4-18
4.8 ADDAD .....	4-21
4.9 ADDAH .....	4-23
4.10 ADDAW .....	4-27
4.11 ADDDP .....	4-30
4.12 ADDK .....	4-32
4.13 ADDKPC.....	4-33
4.14 ADDSP .....	4-35
4.15 ADDSUB .....	4-37
4.16 ADDSUB2 .....	4-39
4.17 ADDU .....	4-41
4.18 ADD2 .....	4-43
4.19 ADD4 .....	4-46
4.20 AND .....	4-49
4.21 ANDN .....	4-51
4.22 AVG2.....	4-54
4.23 AVGU4 .....	4-56
4.24 B .....	4-58
4.25 B .....	4-60
4.26 B IRP .....	4-62
4.27 B NRP .....	4-64
4.28 BDEC .....	4-66

**Contents**

4.29 BITC4.....	4-68
4.30 BITR.....	4-70
4.31 BNOP .....	4-72
4.32 BNOP .....	4-75
4.33 BPOS.....	4-78
4.34 CALLP.....	4-80
4.35 CCMATMPY .....	4-82
4.36 CCMATMPYR1 .....	4-85
4.37 CCMPY32R1 .....	4-87
4.38 CLR .....	4-90
4.39 CMATMPY .....	4-93
4.40 CMATMPYR1 .....	4-95
4.41 CMPEQ.....	4-97
4.42 CMPEQ2.....	4-100
4.43 CMPEQ4.....	4-103
4.44 CMPEQDP .....	4-106
4.45 CMPEQSP .....	4-108
4.46 CMPGT .....	4-110
4.47 CMPGT2.....	4-113
4.48 CMPGTD.....	4-116
4.49 CMPGTSP .....	4-118
4.50 CMPGTU .....	4-120
4.51 CMPGTU4 .....	4-123
4.52 CMPLT .....	4-126
4.53 CMPLT2 .....	4-129
4.54 CMPLTDP .....	4-131
4.55 CMPLTSP .....	4-133
4.56 CMPLTU .....	4-135
4.57 CMPLTU4.....	4-138
4.58 CMPY .....	4-141
4.59 CMPY32R1 .....	4-143
4.60 CMPYR .....	4-146
4.61 CMPYR1 .....	4-149
4.62 CMPYSP .....	4-152
4.63 CROT270 .....	4-155
4.64 CROT90 .....	4-156
4.65 DADD .....	4-157
4.66 DADD2.....	4-159
4.67 DADDSP .....	4-161
4.68 DAPYS2 .....	4-163
4.69 DAVG2 .....	4-165
4.70 DAVGNR2 .....	4-167
4.71 DAVGNRU4 .....	4-169
4.72 DAVGU4.....	4-171
4.73 DCCMPY .....	4-173
4.74 DCCMPYR1 .....	4-174
4.75 DCMPEQ2 .....	4-177
4.76 DCMPEQ4 .....	4-179
4.77 DCMPGT .....	4-181
4.78 DCMPGTU4 .....	4-183
4.79 DCMPY .....	4-185
4.80 DCMPYR1 .....	4-186
4.81 DCROT270 .....	4-188
4.82 DCROT90.....	4-189

4.83 DDOTP4.....	4-190
4.84 DDOTP4H .....	4-192
4.85 DDOTPH2 .....	4-194
4.86 DDOTPH2R.....	4-197
4.87 DDOTPL2.....	4-199
4.88 DDOTPL2R .....	4-202
4.89 DDOTPSU4H .....	4-204
4.90 DEAL.....	4-206
4.91 DINT .....	4-208
4.92 DINTHSP .....	4-209
4.93 DINTHSP .....	4-211
4.94 DINTHSPU.....	4-212
4.95 DINTSPU .....	4-214
4.96 DMAX2.....	4-215
4.97 DMAXU4 .....	4-217
4.98 DMIN2 .....	4-219
4.99 DMINU4 .....	4-221
4.100 DMPY2.....	4-223
4.101 DMPYSP .....	4-225
4.102 DMPYSU4.....	4-230
4.103 DMPYU2 .....	4-231
4.104 DMPYU4 .....	4-233
4.105 DMV .....	4-234
4.106 DMVD.....	4-236
4.107 DOTP2 .....	4-237
4.108 DOTP4H.....	4-241
4.109 DOTPN2.....	4-243
4.110 DOTPNRSU2.....	4-245
4.111 DOTPNRUS2.....	4-248
4.112 DOTPRSU2 .....	4-250
4.113 DOTPRUS2 .....	4-253
4.114 DOTPSU4 .....	4-255
4.115 DOTPSU4H .....	4-257
4.116 DOTPUS4 .....	4-259
4.117 DOTPU4.....	4-260
4.118 DPACK2.....	4-262
4.119 DPACKH2 .....	4-264
4.120 DPACKH4 .....	4-266
4.121 DPACKHL2 .....	4-267
4.122 DPACKL2.....	4-269
4.123 DPACKL4 .....	4-271
4.124 DPACKLH2 .....	4-272
4.125 DPACKLH4 .....	4-274
4.126 DPACKX2 .....	4-275
4.127 DPINT .....	4-277
4.128 DPSP.....	4-279
4.129 DPTRUNC .....	4-281
4.130 DSADD .....	4-283
4.131 DSADD2 .....	4-285
4.132 DSHL .....	4-287
4.133 DSHL2 .....	4-289
4.134 DSHR .....	4-291
4.135 DSHR2 .....	4-293
4.136 DSHRU .....	4-295

4.137 DSHRU2.....	4-297
4.138 DSMPY2.....	4-299
4.139 DSPACKU4.....	4-303
4.140 DSPINT.....	4-305
4.141 DSPINTH.....	4-307
4.142 DSSUB .....	4-309
4.143 DSSUB2.....	4-311
4.144 DSUB .....	4-313
4.145 DSUB2 .....	4-315
4.146 DSUBSP .....	4-317
4.147 DXPNPND2 .....	4-320
4.148 DXPNPND4 .....	4-322
4.149 EXT .....	4-324
4.150 EXTU.....	4-327
4.151 FADDDP .....	4-330
4.152 FADDSP .....	4-334
4.153 FMOPYDP.....	4-337
4.154 FSUBDP .....	4-339
4.155 FSUBSP .....	4-340
4.156 GMOPY .....	4-342
4.157 GMOPY4.....	4-344
4.158 IDLE.....	4-347
4.159 INTDP .....	4-348
4.160 INTDPU .....	4-350
4.161 INTSP .....	4-352
4.162 INTSPU .....	4-354
4.163 LAND .....	4-356
4.164 LANDN .....	4-357
4.165 LDB(U) .....	4-358
4.166 LDB(U) .....	4-362
4.167 LDDW .....	4-364
4.168 LDH(U) .....	4-369
4.169 LDH(U) .....	4-372
4.170 LDNDW .....	4-374
4.171 LDNW .....	4-378
4.172 LDW .....	4-381
4.173 LDW .....	4-385
4.174 LMBD .....	4-387
4.175 LOR .....	4-389
4.176 MAX2 .....	4-390
4.177 MAXU4 .....	4-393
4.178 MFENCE .....	4-395
4.179 MIN2 .....	4-397
4.180 MINU4 .....	4-400
4.181 MPY .....	4-402
4.182 MPY2 .....	4-404
4.183 MPY2IR .....	4-406
4.184 MPY32 .....	4-408
4.185 MPY32 .....	4-409
4.186 MPY32SU .....	4-410
4.187 MPY32U .....	4-411
4.188 MPY32US .....	4-412
4.189 MPYDP .....	4-413
4.190 MPYH .....	4-415

4.191 MPYHI .....	4-417
4.192 MPYHIR .....	4-419
4.193 MPYHL .....	4-421
4.194 MPYHLU .....	4-423
4.195 MPYHSLU .....	4-424
4.196 MPYHSU .....	4-425
4.197 MPYHU .....	4-427
4.198 MPYHULS .....	4-429
4.199 MPYHUS .....	4-430
4.200 MPYI .....	4-431
4.201 MPYID .....	4-433
4.202 MPYIH .....	4-435
4.203 MPYIHR .....	4-436
4.204 MPYIL .....	4-437
4.205 MPYILR .....	4-438
4.206 MPYLH .....	4-439
4.207 MPYLHU .....	4-441
4.208 MPYLI .....	4-442
4.209 MPYLIR .....	4-444
4.210 MPYLSHU .....	4-446
4.211 MPYLUHS .....	4-447
4.212 MPYSP .....	4-448
4.213 MPYSPDP .....	4-450
4.214 MPYSP2DP .....	4-452
4.215 MPYSU .....	4-454
4.216 MPYSU4 .....	4-456
4.217 MPYU .....	4-458
4.218 MPYU2 .....	4-460
4.219 MPYU4 .....	4-462
4.220 MPYUS .....	4-464
4.221 MPYUS4 .....	4-466
4.222 MV .....	4-468
4.223 MVC .....	4-470
4.224 MVD .....	4-474
4.225 MVK .....	4-475
4.226 MVKH/MVKLH .....	4-478
4.227 MVKL .....	4-480
4.228 NEG .....	4-482
4.229 NOP .....	4-484
4.230 NORM .....	4-486
4.231 NOT .....	4-489
4.232 OR .....	4-490
4.233 PACK2 .....	4-492
4.234 PACKH2 .....	4-495
4.235 PACKH4 .....	4-498
4.236 PACKHL2 .....	4-500
4.237 PACKLH2 .....	4-503
4.238 PACKL4 .....	4-506
4.239 QMPY32 .....	4-508
4.240 QMPYSP .....	4-510
4.241 QSMPY32R1 .....	4-512
4.242 RCPDP .....	4-516
4.243 RCPSP .....	4-518
4.244 RINT .....	4-520

## Contents

4.245 ROTL.....	4-521
4.246 RPACK2 .....	4-523
4.247 RSQRDP .....	4-525
4.248 RSQRSP .....	4-527
4.249 SADD .....	4-529
4.250 SADD2 .....	4-533
4.251 SADDSUB .....	4-536
4.252 SADDSUB2 .....	4-538
4.253 SADDSU2 .....	4-540
4.254 SADDUS2 .....	4-542
4.255 SADDU4 .....	4-545
4.256 SAT .....	4-548
4.257 SET .....	4-551
4.258 SHFL .....	4-554
4.259 SHFL3 .....	4-556
4.260 SHL .....	4-558
4.261 SHL2 .....	4-561
4.262 SHLMB .....	4-563
4.263 SHR .....	4-566
4.264 SHR2 .....	4-569
4.265 SHRMB .....	4-572
4.266 SHRU .....	4-575
4.267 SHRU2 .....	4-577
4.268 SMPY .....	4-580
4.269 SMPYH .....	4-582
4.270 SMPYHL .....	4-583
4.271 SMPYLH .....	4-585
4.272 SMPY2 .....	4-587
4.273 SMPY32 .....	4-590
4.274 SPACK2 .....	4-592
4.275 SPACKU4 .....	4-595
4.276 SPDP .....	4-598
4.277 SPINT .....	4-600
4.278 SPKERNEL .....	4-602
4.279 SPKERNELR .....	4-604
4.280 SPLOOP .....	4-605
4.281 SPLOOPD .....	4-606
4.282 SPLOOPW .....	4-607
4.283 SPMASK .....	4-608
4.284 SPMASKR .....	4-610
4.285 SPTRUNC .....	4-612
4.286 SSHL .....	4-614
4.287 SSHVL .....	4-617
4.288 SSHVR .....	4-620
4.289 SSUB .....	4-623
4.290 SSUB2 .....	4-626
4.291 STB .....	4-628
4.292 STB .....	4-631
4.293 STDW .....	4-633
4.294 STH .....	4-637
4.295 STH .....	4-640
4.296 STNDW .....	4-641
4.297 STNW .....	4-644
4.298 STW .....	4-647

4.299 STW.....	4-650
4.300 SUB.....	4-652
4.301 SUBAB .....	4-656
4.302 SUBABS4.....	4-658
4.303 SUBAH.....	4-660
4.304 SUBAW.....	4-661
4.305 SUBC.....	4-663
4.306 SUBDP .....	4-665
4.307 SUBSP.....	4-668
4.308 SUBU .....	4-671
4.309 SUB2.....	4-673
4.310 SUB4.....	4-676
4.311 SWAP2.....	4-678
4.312 SWAP4.....	4-680
4.313 SWE.....	4-682
4.314 SWENR.....	4-683
4.315 UNPKBU4 .....	4-684
4.316 UNPKH2.....	4-686
4.317 UNPKHU2 .....	4-688
4.318 UNPKHU4 .....	4-690
4.319 UNPKLU4 .....	4-693
4.320 XOR.....	4-696
4.321 XORMPY .....	4-699
4.322 XPND2 .....	4-701
4.323 XPND4 .....	4-703
4.324 ZERO.....	4-705

## **Chapter 5**

<i>Pipeline</i>	5-1
5.1 Pipeline Operation Overview .....	5-2
5.1.1 Fetch .....	5-2
5.1.2 Decode .....	5-3
5.1.3 Execute.....	5-4
5.1.4 Pipeline Operation Summary .....	5-5
5.2 Pipeline Execution of Instruction Types .....	5-9
5.2.1 Single-Cycle Instructions .....	5-11
5.2.2 Two-Cycle Instructions and .M Unit Nonmultiply Operations.....	5-12
5.2.3 Store Instructions .....	5-13
5.2.4 Extended Multiply Instructions .....	5-14
5.2.5 Load Instructions .....	5-15
5.2.6 Branch Instructions .....	5-16
5.2.7 Two-Cycle DP Instructions.....	5-17
5.2.8 Three-Cycle Instructions.....	5-18
5.2.9 Four-Cycle Instructions.....	5-18
5.2.10 INTDP Instruction .....	5-19
5.2.11 Double-Precision (DP) Compare Instructions.....	5-19
5.2.12 ADDDP/SUBDP Instructions .....	5-20
5.2.13 MPYI Instruction .....	5-21
5.2.14 MPYID Instruction .....	5-21
5.2.15 MPYDP Instruction.....	5-21
5.2.16 MPYSPDP Instruction .....	5-22
5.2.17 MPYSP2DP Instruction .....	5-23
5.3 Functional Unit Constraints .....	5-24
5.3.1 .S-Unit Constraints .....	5-24
5.3.2 .M-Unit Constraints .....	5-28

5.3.3 L-Unit Constraints .....	.5-36
5.3.4 D-Unit Instruction Constraints .....	.5-40
<b>5.4 Performance Considerations .....</b>	<b>.5-43</b>
5.4.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet .....	.5-43
5.4.2 Multicycle NOPs .....	.5-44
5.4.3 Memory Considerations.....	.5-46

**Chapter 6**

<b><i>Interrupts</i></b>	<b>6-1</b>
6.1 Overview .....	.6-2
6.1.1 Types of Interrupts and Signals Used .....	.6-2
6.1.1.1 Reset (RESET) .....	.6-3
6.1.1.2 Nonmaskable Interrupt (NMI).....	.6-3
6.1.1.3 Maskable Interrupts (INT4-INT15) .....	.6-4
6.1.1.4 Interrupt Service Table (IST).....	.6-4
6.1.1.5 Interrupt Service Fetch Packet (ISFP) .....	.6-5
6.1.1.6 Interrupt Service Table Pointer (ISTP) .....	.6-8
6.1.2 Summary of Interrupt Control Registers.....	.6-9
6.2 Globally Enabling and Disabling Interrupts .....	.6-10
6.3 Individual Interrupt Control .....	.6-13
6.3.1 Enabling and Disabling Interrupts .....	.6-13
6.3.2 Status of Interrupts.....	.6-13
6.3.3 Setting and Clearing Interrupts .....	.6-14
6.3.4 Returning From Interrupt Servicing.....	.6-14
6.3.4.1 CPU State After RESET.....	.6-14
6.3.4.2 Returning From Nonmaskable Interrupts .....	.6-15
6.3.4.3 Returning From Maskable Interrupts.....	.6-15
6.4 Interrupt Detection and Processing .....	.6-16
6.4.1 Setting the Nonreset Interrupt Flag.....	.6-17
6.4.1.1 Detection of Missed Interrupts .....	.6-18
6.4.2 Conditions for Processing a Nonreset Interrupt.....	.6-18
6.4.3 Saving TSR Context in Nonreset Interrupt Processing.....	.6-19
6.4.4 Actions Taken During Nonreset Interrupt Processing .....	.6-20
6.4.5 Conditions for Processing a Nonmaskable Interrupt .....	.6-20
6.4.6 Saving of Context in Nonmaskable Interrupt Processing.....	.6-23
6.4.7 Actions Taken During Nonmaskable Interrupt Processing .....	.6-23
6.4.8 Setting the RESET Interrupt Flag .....	.6-23
6.4.9 Actions Taken During RESET Interrupt Processing .....	.6-24
6.5 Performance Considerations .....	.6-25
6.5.1 General Performance.....	.6-25
6.5.2 Pipeline Interaction .....	.6-25
6.6 Programming Considerations .....	.6-26
6.6.1 Single Assignment Programming.....	.6-26
6.6.2 Nested Interrupts .....	.6-27
6.6.3 Manual Interrupt Processing (polling).....	.6-28
6.6.4 Traps .....	.6-28

**Chapter 7**

<b><i>CPU Exceptions</i></b>	<b>7-1</b>
7.1 Overview .....	.7-2
7.1.1 Types of Exceptions and Signals Used.....	.7-2
7.1.1.1 Reset (RESET) .....	.7-2
7.1.1.2 Nonmaskable Interrupt (NMI).....	.7-2
7.1.1.3 Exception (EXCEP) .....	.7-3
7.1.1.4 Internal Exceptions.....	.7-3

7.1.1.5 Exception Acknowledgment .....	7-3
7.1.2 Exception Service Vector .....	7-3
7.1.3 Summary of Exception Control Registers.....	7-4
7.2 Exception Control.....	7-5
7.2.1 Enabling and Disabling External Exceptions..	7-5
7.2.2 Pending Exceptions.....	7-6
7.2.3 Exception Event Context Saving.....	7-7
7.2.4 Returning From Exception Servicing.....	7-8
7.3 Exception Detection and Processing.....	7-8
7.3.1 Setting the Exception Pending Flag.....	7-8
7.3.2 Conditions for Processing an External Exception .....	7-9
7.3.3 Actions Taken During External Exception (EXCEP) Processing.....	7-11
7.3.4 Nested Exceptions .....	7-12
7.4 Performance Considerations .....	7-12
7.4.1 General Performance.....	7-12
7.4.2 Pipeline Interaction .....	7-12
7.5 Programming Considerations .....	7-15
7.5.1 Internal Exceptions.....	7-15
7.5.2 Internal Exception Report Register (IERR).....	7-15
7.5.3 Software Exception .....	7-16
7.5.3.1 SWE Instruction.....	7-16
7.5.3.2 SWENR Instruction .....	7-16

## **Chapter 8**

<i>Software Pipelined Loop (SPLOOP) Buffer</i>	8-1
8.1 Software Pipelining .....	8-2
8.2 Software Pipelining .....	8-3
8.3 Terminology.....	8-3
8.4 SPLOOP Hardware Support.....	8-3
8.4.1 Loop Buffer .....	8-4
8.4.2 Loop Buffer Count Register (LBC) .....	8-4
8.4.3 Inner Loop Count Register (ILC).....	8-4
8.4.4 Reload Inner Loop Count Register (RILC) .....	8-4
8.4.5 Task State Register (TSR), Interrupt Task State Register (ITSR), and NMI/Exception Task State Register (NTSR)	
8-4	
8.5 SPLOOP-Related Instructions .....	8-5
8.5.1 SPLOOP, SPLOOPD, and SPLOOPW Instructions .....	8-5
8.5.1.1 SPLOOP Instruction .....	8-5
8.5.1.2 SPLOOPD Instruction.....	8-5
8.5.1.3 SPLOOPW Instruction .....	8-6
8.5.2 SPKERNEL and SPKERNELR Instructions .....	8-6
8.5.2.1 SPKERNEL Instruction .....	8-7
8.5.2.2 SPKERNELR Instruction .....	8-7
8.5.3 SPMASK and SPMASKR Instructions.....	8-7
8.6 Basic SPLOOP Example.....	8-9
8.6.1 Some Points About the Basic SPLOOP Example .....	8-10
8.6.2 Same Example Using the SPLOOPW Instruction .....	8-11
8.6.3 Some Points About the SPLOOPW Example .....	8-12
8.7 Loop Buffer.....	8-13
8.7.1 Software Pipeline Execution From the Loop Buffer .....	8-13
8.7.2 Stage Boundary Terminology.....	8-14
8.7.3 Loop Buffer Operation .....	8-14
8.7.3.1 Interrupt During SPLOOP Operation.....	8-15
8.7.3.2 Loop Buffer Active or Idle .....	8-15
8.7.3.3 Loading Instructions into the Loop Buffer.....	8-16
8.7.3.4 Fetching (Dispatching) Instructions from the Loop Buffer.....	8-16

8.7.3.5 Disabling (Draining) Instructions in the Loop Buffer .....	8-17
8.7.3.6 Enabling (Reloading) Instructions in the Loop Buffer .....	8-17
<b>8.8 Execution Patterns.....</b>	<b>8-18</b>
8.8.1 Prolog, Kernel, and Epilog Execution Patterns.....	8-18
8.8.2 Early-Exit Execution Pattern .....	8-19
8.8.3 Reload Execution Pattern.....	8-20
<b>8.9 Loop Buffer Control Using the Unconditional SPLOOP(D) Instruction .....</b>	<b>8-22</b>
8.9.1 Initial Termination Condition Test and ILC Decrement.....	8-22
8.9.2 Stage Boundary Termination Condition Test and ILC Decrement.....	8-22
8.9.3 Using SPLOOPD for Loops with Known Minimum Iteration Counts.....	8-23
8.9.4 Program Memory Fetch Enable Delay During Epilog .....	8-24
8.9.5 Stage Boundary and SPKERNEL(R) Position.....	8-24
8.9.6 Loop Buffer Reload.....	8-25
8.9.6.1 Reload Start.....	8-25
8.9.6.2 Resetting ILC With RILC.....	8-25
8.9.6.3 Program Memory Fetch Disable During Reload .....	8-25
8.9.6.4 Restrictions on Interruptible Loops that Reload .....	8-26
8.9.6.5 Restrictions on Reload Enforced by the Assembler .....	8-26
8.9.7 Restrictions on Accessing ILC and RILC.....	8-27
<b>8.10 Loop Buffer Control Using the SPLOOPW Instruction.....</b>	<b>8-29</b>
8.10.1 Initial Termination Condition Using the SPLOOPW Condition.....	8-29
8.10.2 Stage Boundary Termination Condition Using the SPLOOPW Condition .....	8-29
8.10.3 Interrupting the Loop Buffer When Using SPLOOPW .....	8-29
8.10.4 Under-Execution of Early Stages of SPLOOPW When Termination Condition Becomes True While Interrupt Draining .....	8-31
<b>8.11 Using the SPMASK Instruction.....</b>	<b>8-31</b>
8.11.1 Using SPMASK to Merge Setup Code Example .....	8-32
8.11.2 Some Points About the SPMASK to Merge Setup Code Example .....	8-33
8.11.3 Using SPMASK to Merge Reset Code Example.....	8-33
8.11.4 Some Points About the SPMASK to Merge Reset Code Example .....	8-34
8.11.5 Returning from an Interrupt.....	8-34
<b>8.12 Program Memory Fetch Control.....</b>	<b>8-35</b>
8.12.1 Program Memory Fetch Disable .....	8-35
8.12.2 Program Memory Fetch Enable.....	8-35
<b>8.13 Interrupts .....</b>	<b>8-36</b>
8.13.1 Interrupting the Loop Buffer .....	8-36
8.13.2 Returning to an SPLOOP(D/W) After an Interrupt.....	8-37
8.13.3 Exceptions.....	8-37
8.13.4 Branch to Interrupt, Pipe-Down Sequence .....	8-37
8.13.5 Return from Interrupt, Pipe-Up Sequence.....	8-37
8.13.6 Disabling Interrupts During Loop Buffer Operation.....	8-38
<b>8.14 Branch Instructions .....</b>	<b>8-39</b>
<b>8.15 Instruction Resource Conflicts and SPMASK Operation .....</b>	<b>8-40</b>
8.15.1 Program Memory and Loop Buffer Resource Conflicts .....	8-40
8.15.2 Restrictions on Stall Detection Within SPLOOP Operation .....	8-40
<b>8.16 Restrictions on Crosspath Stalls .....</b>	<b>8-41</b>
<b>8.17 Restrictions on AMR-Related Stalls .....</b>	<b>8-41</b>
<b>8.18 Restrictions on Instructions Placed in the Loop Buffer .....</b>	<b>8-41</b>

## Chapter 9

<b>CPU Privilege</b>	<b>9-1</b>
9.1 Overview .....	9-1
9.2 Execution Modes .....	9-2
9.2.1 Privilege Mode After Reset .....	9-2
9.2.2 Execution Mode Transitions .....	9-2
9.2.3 Supervisor Mode.....	9-2

9.2.4 User Mode .....	9-2
9.2.4.1 Restricted Control Register Access in User Mode .....	9-2
9.2.4.2 Partially Restricted Control Register Access in User Mode.....	9-3
9.2.4.3 Restricted Instruction Execution in User Mode .....	9-3
9.3 Interrupts and Exception Handling .....	9-4
9.3.1 Inhibiting Interrupts in User Mode .....	9-4
9.3.2 Privilege and Interrupts .....	9-4
9.3.3 Privilege and Exceptions .....	9-4
9.3.4 Privilege and Memory Protection .....	9-4
9.4 Operating System Entry .....	9-5
9.4.1 Entering User Mode from Supervisor Mode .....	9-5
9.4.2 Entering Supervisor Mode from User Mode .....	9-6

**Appendix A**

<i>Instruction Compatibility</i>	A-1
----------------------------------	-----

**Appendix B**

<i>Mapping Between Instruction and Functional Unit</i>	B-1
--	-----

**Appendix C**

<i>.D Unit Instructions and Opcode Maps</i>	C-1
C.1 Instructions Executing in the .D Functional Unit.....	C-1
C.2 Opcode Map Symbols and Meanings.....	C-2
C.3 32-Bit Opcode Maps .....	C-3
C.4 16-Bit Opcode Maps .....	C-4

**Appendix D**

<i>.L Unit Instructions and Opcode Maps</i>	D-1
D.1 Instructions Executing in the .L Functional Unit.....	D-1
D.2 Opcode Map Symbols and Meanings.....	D-1
D.3 32-Bit Opcode Maps .....	D-2
D.4 16-Bit Opcode Maps .....	D-3

**Appendix E**

<i>.M Unit Instructions and Opcode Maps</i>	E-1
E.1 Instructions Executing in the .M Functional Unit .....	E-1
E.2 Opcode Map Symbols and Meanings .....	E-2
E.3 32-Bit Opcode Maps .....	E-2
E.4 16-Bit Opcode Maps .....	E-3

**Appendix F**

<i>.S Unit Instructions and Opcode Maps</i>	F-1
F.1 Instructions Executing in the .S Functional Unit .....	F-1
F.2 Opcode Map Symbols and Meanings .....	F-1
F.3 32-Bit Opcode Maps .....	F-2
F.4 16-Bit Opcode Maps .....	F-4

**Appendix G**

<i>.D, .L, or .S Unit Opcode Maps</i>	G-1
---------------------------------------	-----

**Contents**

---

G.1	Opcode Map Symbols and Meanings.....	G-1
G.2	32-Bit Opcode Maps .....	G-2
G.3	16-Bit Opcode Maps .....	G-2

**Appendix H**

---

<i>No Unit Specified Instructions and Opcode Maps</i>	H-1
H.1 Instructions Executing With No Unit Specified .....	H-2
H.2 Opcode Map Symbols and Meanings.....	H-2
H.3 32-Bit Opcode Maps .....	H-3
H.4 16-Bit Opcode Maps .....	H-3

## List of Tables

Table 1-1	Raw Performance Comparison Between the C674x and the C66x.....	1-4
Table 2-1	64-bit Register Pairs .....	2-4
Table 2-2	128-bit Register Quadruplets .....	2-4
Table 2-3	Modulo 2 Arithmetic .....	2-7
Table 2-4	Modulo 5 Arithmetic .....	2-8
Table 2-5	Modulo Arithmetic for Field GF( $2^3$ ).....	2-9
Table 2-6	Control Registers .....	2-10
Table 2-7	Addressing Mode Register (AMR) Field Descriptions .....	2-12
Table 2-8	Block Size Calculations.....	2-13
Table 2-9	Control Status Register (CSR) Field Descriptions .....	2-15
Table 2-10	Galois Field Polynomial Generator Function Register (GFPGFR) Field Descriptions .....	2-17
Table 2-11	Interrupt Clear Register (ICR) Field Descriptions.....	2-18
Table 2-12	Interrupt Enable Register (IER) Field Descriptions .....	2-19
Table 2-13	Interrupt Flag Register (IFR) Field Descriptions .....	2-20
Table 2-14	Interrupt Set Register (ISR) Field Descriptions.....	2-21
Table 2-15	Interrupt Service Table Pointer Register (ISTP) Field Descriptions.....	2-22
Table 2-16	Control Register File Extensions .....	2-24
Table 2-17	Exception Flag Register (EFR) Field Descriptions .....	2-25
Table 2-18	Internal Exception Report Register (IERR) Field Descriptions.....	2-27
Table 2-19	NMI/Exception Task State Register (NTSR) Field Descriptions.....	2-29
Table 2-20	Saturation Status Register Field Descriptions .....	2-30
Table 2-21	Task State Register (TSR) Field Descriptions.....	2-33
Table 2-22	Control Register File Extensions for Floating-Point Operations .....	2-34
Table 2-23	Floating-Point Adder Configuration Register (FADCR) Field Descriptions .....	2-35
Table 2-24	Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions .....	2-38
Table 2-25	Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions .....	2-40
Table 3-1	Instruction Operation and Execution Notations.....	3-2
Table 3-2	Instruction Syntax and Opcode Notations .....	3-4
Table 3-3	IEEE Floating-Point Notations .....	3-6
Table 3-4	Special Single-Precision Values .....	3-7
Table 3-5	Hexadecimal and Decimal Representation for Selected Single-Precision Values.....	3-7
Table 3-6	Special Double-Precision Values .....	3-8
Table 3-7	Hexadecimal and Decimal Representation for Selected Double-Precision Values .....	3-8
Table 3-8	Delay Slot and Functional Unit Latency.....	3-10
Table 3-9	Registers That Can Be Tested by Conditional Operations.....	3-14
Table 3-10	Indirect Address Generation for Load/Store .....	3-28
Table 3-11	Address Generator Options for Load/Store .....	3-28
Table 3-12	Layout Field Description in Compact Instruction Packet Header.....	3-30
Table 3-13	Expansion Field Description in Compact Instruction Packet Header .....	3-31
Table 3-14	LD/ST Data Size Selection.....	3-32
Table 3-15	P-bits Field Description in Compact Instruction Packet Header.....	3-32
Table 3-16	Available Compact Instructions .....	3-34
Table 4-1	Relationships Between Operands, Operand Size, Functional Units, and Opfields for Example Instruction (ADD) .....	4-3
Table 4-2	Program Counter Values for Branch Using a Displacement Example .....	4-59
Table 4-3	Program Counter Values for Branch Using a Register Example .....	4-61
Table 4-4	Program Counter Values for B IRP Instruction Example.....	4-63
Table 4-5	Program Counter Values for B NRP Instruction Example.....	4-65
Table 4-6	Data Types Supported by LDB(U) Instruction .....	4-358
Table 4-7	Data Types Supported by LDB(U) Instruction (15-Bit Offset).....	4-362
Table 4-8	Data Types Supported by LDH(U) Instruction .....	4-369
Table 4-9	Data Types Supported by LDH(U) Instruction (15-Bit Offset).....	4-372
Table 4-10	Register Addresses for Accessing the Control Registers .....	4-472

*List of Tables*

Table 4-11	Field Allocation in stg/cyc Field .....	4-603
Table 4-12	Bit Allocations to Stage and Cycle in stg/cyc Field .....	4-603
Table 5-1	Operations Occurring During Pipeline Phases .....	5-5
Table 5-2	Execution Stage Length Description for Each Instruction Type - Part A .....	5-9
Table 5-3	Execution Stage Length Description for Each Instruction Type - Part B .....	5-10
Table 5-4	Execution Stage Length Description for Each Instruction Type - Part C .....	5-10
Table 5-5	Execution Stage Length Description for Each Instruction Type - Part D .....	5-11
Table 5-6	Single-Cycle Instruction Execution .....	5-11
Table 5-7	Multiply Instruction Execution .....	5-12
Table 5-8	Store Instruction Execution .....	5-13
Table 5-9	Extended Multiply Instruction Execution .....	5-14
Table 5-10	Load Instruction Execution .....	5-15
Table 5-11	Branch Instruction Execution .....	5-16
Table 5-12	Two-Cycle DP Instruction Execution .....	5-17
Table 5-13	Three-Cycle Instruction Execution .....	5-18
Table 5-14	Four-Cycle Instruction Execution .....	5-19
Table 5-15	INTDP Instruction Execution .....	5-19
Table 5-16	DP Compare Instruction Execution .....	5-20
Table 5-17	ADDDP/SUBDP Instruction Execution .....	5-20
Table 5-18	MPYI Instruction Execution .....	5-21
Table 5-19	MPYID Instruction Execution .....	5-21
Table 5-20	MPYDP Instruction Execution .....	5-22
Table 5-21	MPYSPDP Instruction Execution .....	5-22
Table 5-22	MPYSP2DP Instruction Execution .....	5-23
Table 5-23	Single-Cycle .S-Unit Instruction Constraints .....	5-24
Table 5-24	DP Compare .S-Unit Instruction Constraints .....	5-25
Table 5-25	2-Cycle DP .S-Unit Instruction Constraints .....	5-26
Table 5-26	ADDSP/SUBSP .S-Unit Instruction Constraints .....	5-26
Table 5-27	ADDDP/SUBDP .S-Unit Instruction Constraints .....	5-27
Table 5-28	Branch .S-Unit Instruction Constraints .....	5-27
Table 5-29	16 × 16 Multiply .M-Unit Instruction Constraints .....	5-28
Table 5-30	4-Cycle .M-Unit Instruction Constraints .....	5-29
Table 5-31	MPYI .M-Unit Instruction Constraints .....	5-30
Table 5-32	MPYID .M-Unit Instruction Constraints .....	5-31
Table 5-33	MPYDP .M-Unit Instruction Constraints .....	5-32
Table 5-34	MPYSP .M-Unit Instruction Constraints .....	5-33
Table 5-35	MPYSPDP .M-Unit Instruction Constraints .....	5-34
Table 5-36	MPYSP2DP .M-Unit Instruction Constraints .....	5-35
Table 5-37	Single-Cycle .L-Unit Instruction Constraints .....	5-36
Table 5-38	4-Cycle .L-Unit Instruction Constraints .....	5-37
Table 5-39	INTDP .L-Unit Instruction Constraints .....	5-38
Table 5-40	ADDDP/SUBDP .L-Unit Instruction Constraints .....	5-39
Table 5-41	Load .D-Unit Instruction Constraints .....	5-40
Table 5-42	Store .D-Unit Instruction Constraints .....	5-41
Table 5-43	Single-Cycle .D-Unit Instruction Constraints .....	5-42
Table 5-44	LDDW Instruction With Long Write Instruction Constraints .....	5-42
Table 5-45	Program Memory Accesses Versus Data Load Accesses .....	5-47
Table 6-1	Interrupt Priorities .....	6-3
Table 6-2	Interrupt Control Registers .....	6-9
Table 6-3	TSR Field Behavior When an Interrupt is Taken .....	6-20
Table 6-4	TSR Field Behavior When an NMI Interrupt is Taken .....	6-23
Table 7-1	Exception-Related Control Registers .....	7-4
Table 7-2	NTSR Field Behavior When an Exception is Taken .....	7-7
Table 7-3	TSR Field Behavior When an Exception is Taken (EXC = 0) .....	7-10

Table 8-1	SPLOOP Instruction Flow for <a href="#">Example 8-4</a> and <a href="#">Example 8-5</a> .....	.8-10
Table 8-2	SPLOOPW Instruction Flow for <a href="#">Example 8-7</a> .....	.8-11
Table 8-3	Software Pipeline Instruction Flow Using the Loop Buffer.....	.8-14
Table 8-4	SPLOOPD Minimum Loop Iterations.....	.8-23
Table 8-5	SPLOOP Instruction Flow for First Three Cycles of <a href="#">Example 8-15</a> .....	.8-32
Table 8-6	SPLOOP Instruction Flow for <a href="#">Example 8-15</a> .....	.8-34
Table A-1	Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs.....	A-1
Table B-1	Instruction to Functional Unit Mapping .....	B-1
Table C-1	Instructions Executing in the .D Functional Unit .....	C-1
Table C-2	.D Unit Opcode Map Symbol Definitions .....	C-2
Table C-3	Address Generator Options for Load/Store .....	C-3
Table D-1	Instructions Executing in the .L Functional Unit.....	D-1
Table D-2	.L Unit Opcode Map Symbol Definitions .....	D-2
Table E-1	Instructions Executing in the .M Functional Unit.....	E-1
Table E-2	.M Unit Opcode Map Symbol Definitions .....	E-2
Table F-1	Instructions Executing in the .S Functional Unit.....	F-1
Table F-2	.S Unit Opcode Map Symbol Definitions.....	F-2
Table G-1	.D, .L, and .S Units Opcode Map Symbol Definitions.....	G-1
Table H-1	Instructions Executing With No Unit Specified .....	H-2
Table H-2	No Unit Specified Instructions Opcode Map Symbol Definitions.....	H-2

## List of Figures

---

Figure 1-1	QMPY32 - Example of vector instruction.....	1-4
Figure 1-2	TMS320C66x DSP Block Diagram.....	1-5
Figure 2-1	CPU Data Paths.....	2-3
Figure 2-2	Addressing Mode Register (AMR) .....	2-12
Figure 2-3	Control Status Register (CSR).....	2-15
Figure 2-4	PWRD Field of Control Status Register (CSR) .....	2-15
Figure 2-5	Galois Field Polynomial Generator Function Register (GFPGFR).....	2-17
Figure 2-6	Interrupt Clear Register (ICR).....	2-18
Figure 2-7	Interrupt Enable Register (IER) .....	2-19
Figure 2-8	Interrupt Flag Register (IFR).....	2-20
Figure 2-9	Interrupt Return Pointer Register (IRP).....	2-21
Figure 2-10	Interrupt Set Register (ISR).....	2-21
Figure 2-11	Interrupt Service Table Pointer Register (ISTP) .....	2-22
Figure 2-12	NMI Return Pointer Register (NRP) .....	2-23
Figure 2-13	E1 Phase Program Counter (PCE1).....	2-23
Figure 2-14	DSP Core Number Register (DNUM) .....	2-24
Figure 2-15	Exception Flag Register (EFR) .....	2-25
Figure 2-16	GMPY Polynomial A-Side Register (GPLYA) .....	2-25
Figure 2-17	GMPY Polynomial B-Side (GPLYB) .....	2-26
Figure 2-18	Internal Exception Report Register (IERR) .....	2-27
Figure 2-19	Inner Loop Count Register (ILC) .....	2-28
Figure 2-20	Interrupt Task State Register (ITSR).....	2-28
Figure 2-21	NMI/Exception Task fState Register (NTSR) .....	2-29
Figure 2-22	Reload Inner Loop Count Register (RILC) .....	2-30
Figure 2-23	Saturation Status Register (SSR) .....	2-30
Figure 2-24	Time Stamp Counter Register - Low Half (TSCL).....	2-31
Figure 2-25	Time Stamp Counter Register - High Half (TSCH).....	2-31
Figure 2-26	Task State Register (TSR) .....	2-33
Figure 2-27	Floating-Point Adder Configuration Register (FADCR) .....	2-35
Figure 2-28	Floating-Point Auxiliary Configuration Register (FAUCR) .....	2-37
Figure 2-29	Floating-Point Multiplier Configuration Register (FMCR) .....	2-40
Figure 3-1	Single-Precision Floating-Point Fields .....	3-7
Figure 3-2	Double-Precision Floating-Point Fields .....	3-8
Figure 3-3	Basic Format of a Fetch Packet .....	3-11
Figure 3-4	Examples of the Detectability of Write Conflicts by the Assembler.....	3-19
Figure 3-5	CPU Fetch Packet Types .....	3-29
Figure 3-6	Compact Instruction Header Format .....	3-30
Figure 3-7	Layout Field in Compact Header Word .....	3-30
Figure 3-8	Expansion Field in Compact Header Word.....	3-31
Figure 3-9	P-bits Field in Compact Header Word .....	3-32
Figure 5-1	Pipeline Stages.....	5-2
Figure 5-2	Fetch Phases of the Pipeline .....	5-3
Figure 5-3	Decode Phases of the Pipeline .....	5-4
Figure 5-4	Execute Phases of the Pipeline .....	5-4
Figure 5-5	Pipeline Phases.....	5-5
Figure 5-6	Pipeline Operation: One Execute Packet per Fetch Packet.....	5-5
Figure 5-7	Pipeline Phases Block Diagram.....	5-7
Figure 5-8	Single-Cycle Instruction Phases .....	5-11
Figure 5-9	Single-Cycle Instruction Execution Block Diagram .....	5-11
Figure 5-10	Two-Cycle Instruction Phases .....	5-12
Figure 5-11	Single 16 × 16 Multiply Instruction Execution Block Diagram .....	5-12
Figure 5-12	Store Instruction Phases .....	5-13

Figure 5-13	Store Instruction Execution Block Diagram .....	.5-13
Figure 5-14	Extended Multiply Instruction Phases .....	.5-14
Figure 5-15	Extended Multiply Instruction Execution Block Diagram .....	.5-14
Figure 5-16	Load Instruction Phases .....	.5-15
Figure 5-17	Load Instruction Execution Block Diagram .....	.5-15
Figure 5-18	Branch Instruction Phases .....	.5-16
Figure 5-19	Branch Instruction Execution Block Diagram .....	.5-17
Figure 5-20	Two-Cycle DP Instruction Phases.....	.5-17
Figure 5-21	Three-Cycle Instruction Phases.....	.5-18
Figure 5-22	Four-Cycle Instruction Phases.....	.5-19
Figure 5-23	INTDP Instruction Phases .....	.5-19
Figure 5-24	DP Compare Instruction Phases .....	.5-20
Figure 5-25	ADDDP/SUBDP Instruction Phases .....	.5-20
Figure 5-26	MPYI Instruction Phases .....	.5-21
Figure 5-27	MPYID Instruction Phases.....	.5-21
Figure 5-28	MPYDP Instruction Phases .....	.5-22
Figure 5-29	MPYSPDP Instruction Phases.....	.5-22
Figure 5-30	MPYSP2DP Instruction Phases .....	.5-23
Figure 5-31	Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets.....	.5-44
Figure 5-32	Multicycle NOP in an Execute Packet.....	.5-45
Figure 5-33	Branching and Multicycle NOPs.....	.5-46
Figure 5-34	Pipeline Phases Used During Memory Accesses .....	.5-46
Figure 5-35	Program and Data Memory Stalls .....	.5-47
Figure 6-1	Interrupt Service Table .....	6-5
Figure 6-2	Interrupt Service Fetch Packet .....	6-6
Figure 6-3	Interrupt Service Table With Branch to Additional Interrupt Service Code Located Outside the IST .....	6-7
Figure 6-4	Nonreset Interrupt Detection and Processing: Pipeline Operation .....	6-17
Figure 6-5	Return from Interrupt Execution and Processing: Pipeline Operation .....	6-19
Figure 6-6	CPU Nonmaskable Interrupt Detection and Processing: Pipeline Operation.....	6-21
Figure 6-7	CPU Return from Nonmaskable Interrupt Execution and Processing: Pipeline Operation .....	6-22
Figure 6-8	<u>RESET</u> Interrupt Detection and Processing: Pipeline Operation .....	6-24
Figure 7-1	Interrupt Service Table With Branch to Additional Exception Service Code Located Outside the IST .....	7-5
Figure 7-2	External Exception (EXCEP) Detection and Processing: Pipeline Operation.....	7-9
Figure 7-3	Return from Exception Processing: Pipeline Operation.....	7-11
Figure 7-4	NMI Exception Detection and Processing: Pipeline Operation .....	7-13
Figure 7-5	Double Exception Detection and Processing: Pipeline Operation.....	7-14
Figure 8-1	Software Pipelined Execution Flow.....	8-2
Figure 8-2	General Prolog, Kernel, and Epilog Execution Pattern .....	8-18
Figure 8-3	Single Kernel Stage Execution Pattern.....	8-19
Figure 8-4	Early-Exit Execution Pattern.....	8-19
Figure 8-5	Single Loop Iteration Execution Pattern .....	8-20
Figure 8-6	Reload Execution Pattern .....	8-20
Figure 8-7	Reload Early-Exit Execution Pattern .....	8-21
Figure 8-8	Instruction Flow Using Reload .....	8-27
Figure 8-9	Instruction Flow for <code>strncpy()</code> of Null String .....	8-31
Figure C-1	1 or 2 Sources Instruction Format .....	C-3
Figure C-2	Extended .D Unit 1 or 2 Sources Instruction Format.....	C-3
Figure C-3	Load/Store Basic Operations .....	C-3
Figure C-4	Load Nonaligned Doubleword Instruction Format.....	C-4
Figure C-5	Store Nonaligned Doubleword Instruction Format.....	C-4
Figure C-6	Load/Store Long-Immediate Operations .....	C-4
Figure C-7	ADDA Long-Immediate Operations .....	C-4
Figure C-8	Doff4 Instruction Format .....	C-5
Figure C-9	Doff4DW Instruction Format.....	C-5

*List of Figures*

Figure C-10	Dind Instruction Format .....	C-5
Figure C-11	DindDW Instruction Format.....	C-6
Figure C-12	Dinc Instruction Format .....	C-6
Figure C-13	DincDW Instruction Format.....	C-7
Figure C-14	Ddec Instruction Format .....	C-7
Figure C-15	DdecDW Instruction Format .....	C-8
Figure C-16	Dstk Instruction Format .....	C-8
Figure C-17	Dx2op Instruction Format .....	C-8
Figure C-18	Dx5 Instruction Format .....	C-9
Figure C-19	Dx5p Instruction Format.....	C-9
Figure C-20	Dx1 Instruction Format .....	C-9
Figure C-21	Dpp Instruction Format.....	C-10
Figure D-1	1 or 2 Sources Instruction Format .....	D-2
Figure D-2	1 or 2 Sources, Nonconditional Instruction Format.....	D-2
Figure D-3	Unary Instruction Format .....	D-3
Figure D-4	L3 Instruction Format.....	D-3
Figure D-5	L3i Instruction Format .....	D-3
Figure D-6	Ltbd Instruction Format .....	D-4
Figure D-7	L2c Instruction Format.....	D-4
Figure D-8	Lx5 Instruction Format .....	D-4
Figure D-9	Lx3c Instruction Format .....	D-5
Figure D-10	Lx1c Instruction Format .....	D-5
Figure D-11	Lx1 Instruction Format .....	D-5
Figure E-1	Extended M-Unit with Compound Operations .....	E-2
Figure E-2	Extended .M Unit 1 or 2 Sources, Nonconditional Instruction Format.....	E-2
Figure E-3	Extended .M-Unit Unary Instruction Format .....	E-3
Figure E-4	M3 Instruction Format .....	E-3
Figure F-1	1 or 2 Sources Instruction Format .....	F-2
Figure F-2	Extended .S Unit 1 or 2 Sources Instruction Format .....	F-2
Figure F-3	Extended .S Unit 1 or 2 Sources, Nonconditional Instruction Format.....	F-3
Figure F-4	Unary Instruction Format .....	F-3
Figure F-5	Extended .S Unit Branch Conditional, Immediate Instruction Format .....	F-3
Figure F-6	Call Unconditional, Immediate with Implied NOP 5 Instruction Format.....	F-3
Figure F-7	Branch with NOP Constant Instruction Format.....	F-3
Figure F-8	Branch with NOP Register Instruction Format.....	F-4
Figure F-9	Branch Instruction Format .....	F-4
Figure F-10	MVK Instruction Format .....	F-4
Figure F-11	Field Operations.....	F-4
Figure F-12	Sbs7 Instruction Format .....	F-5
Figure F-13	Sbu8 Instruction Format .....	F-5
Figure F-14	Scs10 Instruction Format .....	F-5
Figure F-15	Sbs7c Instruction Format .....	F-5
Figure F-16	Sbu8c Instruction Format.....	F-5
Figure F-17	S3 Instruction Format .....	F-6
Figure F-18	S3i Instruction Format .....	F-6
Figure F-19	Smvk8 Instruction Format .....	F-7
Figure F-20	Ssh5 Instruction Format .....	F-7
Figure F-21	S2sh Instruction Format .....	F-7
Figure F-22	Sc5 Instruction Format .....	F-8
Figure F-23	S2ext Instruction Format .....	F-8
Figure F-24	Sx2op Instruction Format.....	F-8
Figure F-25	Sx5 Instruction Format .....	F-9
Figure F-26	Sx1 Instruction Format .....	F-9
Figure F-27	Sx1b Instruction Format .....	F-9

---

Figure G-1	LSDmvto Instruction Format .....	G-2
Figure G-2	LSDmvfr Instruction Format .....	G-2
Figure G-3	LSDx1c Instruction Format.....	G-2
Figure G-4	LSDx1 Instruction Format.....	G-3
Figure H-1	Loop Buffer Instruction Format .....	H-3
Figure H-2	NOP and IDLE Instruction Format .....	H-3
Figure H-3	Uspl Instruction Format.....	H-3
Figure H-4	Uspldr Instruction Format .....	H-4
Figure H-5	Uspk Instruction Format .....	H-4
Figure H-6	Uspm Instruction Format .....	H-4
Figure H-7	Unop Instruction Format .....	H-4

## List of Examples

Example 2-1	Code to Read the 64-Bit TSC Value in Branch Delay Slot .....	2-32
Example 2-2	Code to Read the 64-Bit TSC Value Using DINT/RINT .....	2-32
Example 3-1	Fully Serial p-Bit Pattern in a Fetch Packet .....	3-12
Example 3-2	Fully Parallel p-Bit Pattern in a Fetch Packet .....	3-12
Example 3-3	Partially Serial p-Bit Pattern in a Fetch Packet .....	3-13
Example 3-4	LDW Instruction in Circular Mode .....	3-25
Example 3-5	ADDAH Instruction in Circular Mode .....	3-26
Example 3-6	LDNW in Circular Mode .....	3-27
Example 5-1	Execute Packet in <a href="#">Figure 5-7</a> .....	5-8
Example 6-1	Relocation of Interrupt Service Table .....	6-8
Example 6-2	Interrupts Versus Writes to GIE .....	6-10
Example 6-3	Code Sequence to Disable Maskable Interrupts Globally .....	6-11
Example 6-4	Code Sequence to Enable Maskable Interrupts Globally .....	6-11
Example 6-5	Code Sequence with Disable Global Interrupt Enable .....	6-12
Example 6-6	Code Sequence with Restore Global Interrupt Enable .....	6-12
Example 6-7	Code Sequence with Disable Reenable Interrupt Sequence .....	6-12
Example 6-8	Code Sequence to Enable an Individual Interrupt (INT9) .....	6-13
Example 6-9	Code Sequence to Disable an Individual Interrupt (INT9) .....	6-13
Example 6-10	Code to Set an Individual Interrupt (INT6) and Read the Flag Register .....	6-14
Example 6-11	Code to Clear an Individual Interrupt (INT6) and Read the Flag Register .....	6-14
Example 6-12	Code to Return From NMI .....	6-15
Example 6-13	Code to Return from a Maskable Interrupt .....	6-15
Example 6-14	Code Without Single Assignment: Multiple Assignment of A1 .....	6-26
Example 6-15	Code Using Single Assignment .....	6-26
Example 6-16	Manual Interrupt Processing .....	6-28
Example 6-17	Code Sequence to Invoke a Trap .....	6-28
Example 6-18	Code Sequence for Trap Return .....	6-29
Example 7-1	Code to Return From Exception .....	7-8
Example 7-2	Code to Quickly Detect OS Service Request .....	7-16
Example 8-1	SPMASK Using Unit Mask to Indicate Masked Unit .....	8-8
Example 8-2	SPMASK Using Caret to Indicate Masked Unit .....	8-8
Example 8-3	Copy Loop Coded as C Fragment .....	8-9
Example 8-4	SPLOOP Implementation of Copy Loop .....	8-9
Example 8-5	SPLOOPD Implementation of Copy Loop .....	8-9
Example 8-6	Example C Coded Loop .....	8-11
Example 8-7	SPLOOPW Implementation of C Coded Loop .....	8-11
Example 8-8	SPLOOP, SPLOOP Body, and SPKERNEL .....	8-13
Example 8-9	Using ILC With the SPLOOP Instruction .....	8-23
Example 8-10	Using ILC With a SPLOOPD Instruction .....	8-24
Example 8-11	Using ILC With a SPLOOPD Instruction .....	8-26
Example 8-12	Using the SPLOOPW Instruction .....	8-30
Example 8-13	strcpy() Using the SPLOOPW Instruction .....	8-30
Example 8-14	Using the SPMASK Instruction to Merge Setup Code with SPLOOPW .....	8-32
Example 8-15	Using the SPMASK Instruction to Merge Reset Code with SPLOOP .....	8-33
Example 8-16	Initiating a Branch Prior to SPLOOP Body .....	8-39

# Preface

## About This Manual

The TMS320C66x is the next-generation fixed and floating-point DSP. The new DSP enhances the TMS320C674x, which merged the TMS320C67x+ floating point and the TMS320C64x+ fixed point instruction set architectures. This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C66x DSP.

## Notational Conventions

This document uses the following conventions:

- Commands and keywords are in **boldface** font.
- Arguments for which you supply values are in *italic* font.
- Terminal sessions and information the system displays are in `screen` font.
- Information you must enter is in **boldface screen font**.
- Elements in square brackets ([ ]) are optional.

Notes use the following conventions:



**Note**—Means reader take note. Notes contain helpful suggestions or references to material not covered in the publication.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.



**CAUTION**—Indicates the possibility of service interruption if precautions are not taken.



**WARNING**—Indicates the possibility of damage to equipment if precautions are not taken.

## Related Documentation from Texas Instruments

[\*C66x CorePac User Guide\*](#)  
[\*C66x DSP Cache User Guide\*](#)

SPRUGW0  
SPRUGY8

## Trademarks

VLYNQ, PIQUA Software, wONE, PBCC, Uni-DSL, Dynamic Adaptive Equalization, Telinnovation, TurboDSL Packet Accelerator, interOps Test Labs, TurboDOX, and INCA are trademarks of Texas Instruments Incorporated.

All other brand names and trademarks mentioned in this document are the property of Texas Instruments Incorporated or their respective owners, as applicable.

# Introduction

The TMS320C66x is the next-generation fixed and floating-point DSP. The new DSP enhances the TMS320C674x, which merged the TMS320C67x+ floating point and the TMS320C64x+™fixed point instruction set architectures.

This document describes the CPU architecture, pipeline, instruction set, and interrupts of the C66x DSP. The C66x CorePac is the name used to designate the CPU together with the hardware providing memory, bandwidth management, interrupt, memory protection, and power-down support. The C66x CorePac is not described in this document because it is fully covered in the [C66x CorePac User Guide](#) (SPRUGW0).

- 1.1 "DSP Features and Options" on page 1-2
- 1.2 "DSP Architecture" on page 1-5

## 1.1 DSP Features and Options

The TMS320C66x Instruction Set Architecture is the latest for the C6000 family. As with its predecessors, the C66x is an advanced VLIW architecture with 8 functional units (two multipliers unit and six arithmetic units) that operate in parallel. The C66x CPU consists of 64 general-purpose 32-bit registers and eight functional units. These eight functional units contain:

- Two multipliers
- Six ALUs

The C6000 generation has a complete set of optimized development tools, including an efficient C compiler, an assembly optimizer for simplified assembly-language programming and scheduling, and a Windows® operating system-based debugger interface for visibility into source code execution characteristics. A hardware emulation board, compatible with the TI XDS510™ and XDS560™ emulator interface, is also available. This tool complies with IEEE Standard 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture. Features of the C6000 devices include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units
  - Executes up to eight instructions per cycle
  - Allows designers to develop highly effective RISC-like code for fast development time
- Instruction packing
  - Gives code size equivalence for eight instructions executed serially or in parallel
  - Reduces code size, program fetches, and power consumption
- Conditional execution of most instructions
  - Reduces costly branching
  - Increases parallelism for higher sustained performance
- Efficient code execution on independent functional units
  - Industry's most efficient C compiler on DSP benchmark suite
  - Industry's first assembly optimizer for fast development and improved parallelization
- 8-/16-/32-bit/64-bit data support, providing efficient memory support for a variety of applications
- 40-bit arithmetic options add extra precision for vocoders and other computationally intensive applications
- Saturation and normalization provide support for key arithmetic operations
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

The C64x, C64x+, C674x, and C66x devices include these additional features:

- Each multiplier can perform two  $16 \times 16$ -bit or four  $8 \times 8$  bit multiplies every clock cycle.
- Quad 8-bit and dual 16-bit instruction set extensions with data flow support
- Support for non-aligned 32-bit (word) and 64-bit (double word) memory accesses
- Special communication-specific instructions have been added to address common operations in error-correcting codes.
- Bit count and rotate hardware extends support for bit-level algorithms.

The C64x+, C674x, and C66x devices include these additional features:

- Compact instructions: Common instructions (AND, ADD, LD, MPY) have 16-bit versions to reduce code size.
- Protected mode operation: A two-level system of privileged program execution to support higher-capability operating systems and system features such as memory protection.
- Exceptions support for error detection and program redirection to provide robust code execution
- Hardware support for modulo loop operation to reduce code size and allow interrupts during fully-pipelined code
- Each multiplier can perform  $32 \times 32$  bit multiplies
- Additional instructions to support complex multiplies allowing up to eight 16-bit multiply/add/subtracts per clock cycle

The C66x has the following key improvements to the ISA:

- 4x Multiply Accumulate improvement for both fixed and floating point
- Improvement of the floating point arithmetic
- Enhancement of the vector processing capability for fixed and floating point
- Addition of domain-specific instructions for complex arithmetic and matrix operations.

### 1.1.1 4x Multiply

The new C66x Core ISA significantly improves the maximum number multiply operations that can be executed per cycle. The core can now execute up to 32 (16x16-bit) multiplies per cycle or up to 8 single-precision floating-point multiplies per cycle.

### 1.1.2 Floating point support

The C66x ISA implements native support for IEEE 754 single-precision and double-precision instructions. Floating point improvements relative to the C674x include:

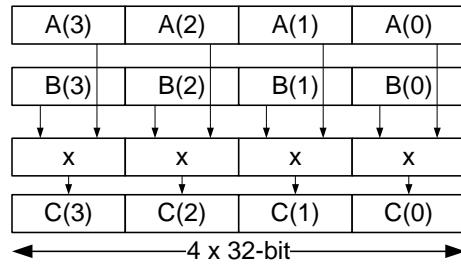
- Fast implementation of all basic floating point operations,
- SIMD support for floating point operations,
- Single-Precision Complex multiply
- Additional resource flexibility (e.g. the conversion from/to integer to/from single-precision floating operations can now be executed on .L and .S units).

The improved performance of the floating point processing capability and the ability to execute both fixed-point and floating-point math adds a new dimension to the capabilities available to DSP algorithm programmers. By selectively using advanced fixed-point instructions and floating-point instructions within a single instruction set, the programmer can achieve significantly higher performance for their algorithm (in term of data precision and cycle count) than ever before.

### 1.1.3 Vector Processing

On the C66x ISA, the vector processing capability is improved by extending the width of the SIMD instructions. The C674x DSP supports 2-way SIMD operations for 16-bit data and 4-way SIMD operations for 8-bit data. C66x enhances this capabilities with the addition of SIMD instructions for 32-bit data allowing operation on 128-bit vectors. For example the QMPY32 instruction is able to perform the element to element multiplication between two vectors of four 32-bit data each.

**Figure 1-1 QMPY32 - Example of vector instruction**



#### 1.1.4 Complex arithmetic and matrix operations

Communication signal processing requires an extensive use of complex arithmetic functions and linear algebra (matrix computation). C66x ISA includes a set of specific instructions to handle complex arithmetic and matrix operations.

For example, the C66x can now perform up to two multiplications of a [1x2] complex vector by a [2x2] complex matrix per cycle and provides a set of instructions that operates either on scalar or vector numbers to perform complex multiplications, conjugate of a complex number, multiplication by the conjugate, rotations of complex numbers, ....

Table 1-1 provides a comparison of the raw performance between the C674x and the C66x ISA.

**Table 1-1 Raw Performance Comparison Between the C674x and the C66x**

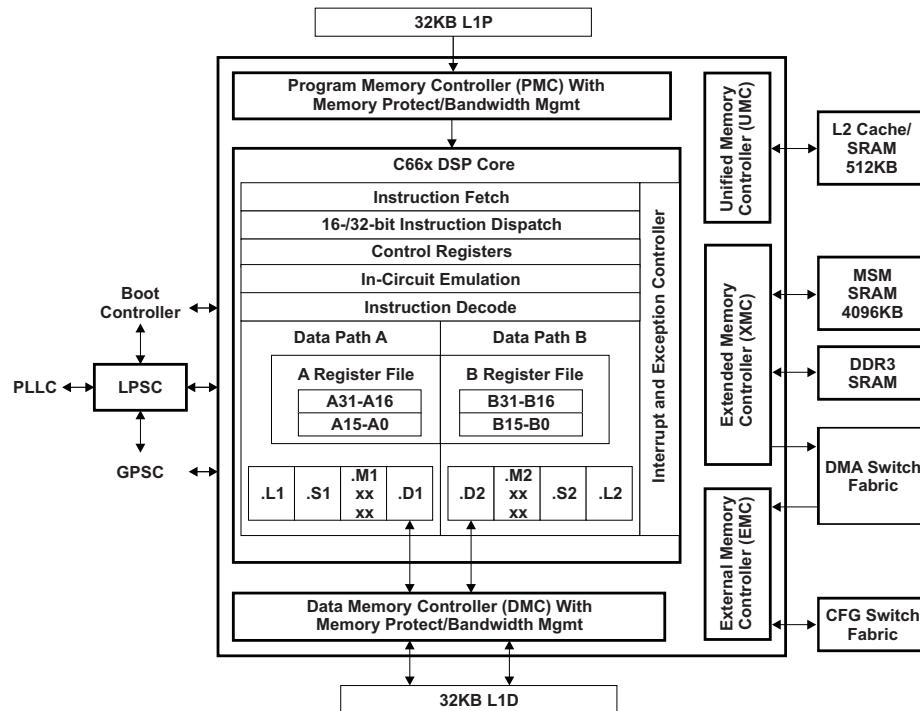
	C674x	C66x
Fixed point 16x16 MACs per cycle	8	32
Fixed point 32x32 MACs per cycle	2	8
Floating point single precision MACs per cycle	2	8
Arithmetic floating point operations per cycle	6 <sup>1</sup>	16 <sup>2</sup>
Arithmetic floating point operations per cycle	6 <sup>3</sup>	16 <sup>4</sup>
Load/store width	2 x 64-bit	2 x 64-bit
Vector size (SIMD capability)	32-bit (2 x 16-bit, 4x-8bits)	128-bit (4 x 32-bit, 4 x 16-bit, 4x-8bits)

1. One operation per .L, .S, .M units for each side (A and B)
2. 2-way SIMD on .L and .S units (e.g. 8 SP operations for A and B) and 4 SP multiply on one .M unit (e.g. 8 SP operations for A and B)
3. One operation per .L, .S, .M units for each side (A and B)
4. 2-way SIMD on .L and .S units (e.g. 8 SP operations for A and B) and 4 SP multiply on one .M unit (e.g. 8 SP operations for A and B)

## 1.2 DSP Architecture

Figure 1-2 is the block diagram for the C66x CorePac DSP. All C66x devices include L1 program and data memories, each of which can be configured as cache and/or SRAM. The devices also have varying sizes of local and shared L2 cache/SRAM as well as external memory interface(s). The number of instances, memory sizes and locations, as well as a full list of peripherals and chip-level connectivity are fully documented in the device data sheet.

**Figure 1-2 TMS320C66x CorePac DSP Block Diagram**



### 1.2.1 Central Processing Unit (CPU)

The C66x CPU, in [Figure 1-2](#), contains:

- Program fetch unit
- 16-/32-bit instruction dispatch unit, advanced instruction packing
- Instruction decode unit
- Two data paths, each with four functional units
- 64 32-bit registers
- Control registers
- Control logic
- Test, emulation, and interrupt logic
- Internal DMA (IDMA) for transfers between internal memories

The program fetch, instruction dispatch, and instruction decode units can deliver up to fifteen 16- and/or 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 32 32-bit general-purpose registers. The data paths are described in more detail in Chapter 2 [“CPU Data Paths and Control”](#) on page 2-1. A control register file provides the means to configure and control various processor operations. To understand how instructions are fetched, dispatched, decoded, and executed in the data path, see Chapter 5 [“Pipeline”](#) on page 5-1.

### 1.2.2 Internal Memory

The DSP has a 32-bit, byte-addressable address space. L1 memory for each DSP CPU memory is organized in separate data and program spaces, with unified memory for L2 and higher.

The DSP has a 256-bit read-only port to access internal program memory and two 256-bit ports (read and write) to access internal data memory.

### 1.2.3 Memory and Peripheral Options

For an overview of the peripherals available on the C66x DSPs Applications Processors, refer to the device-specific data manual and user guides.

# CPU Data Paths and Control

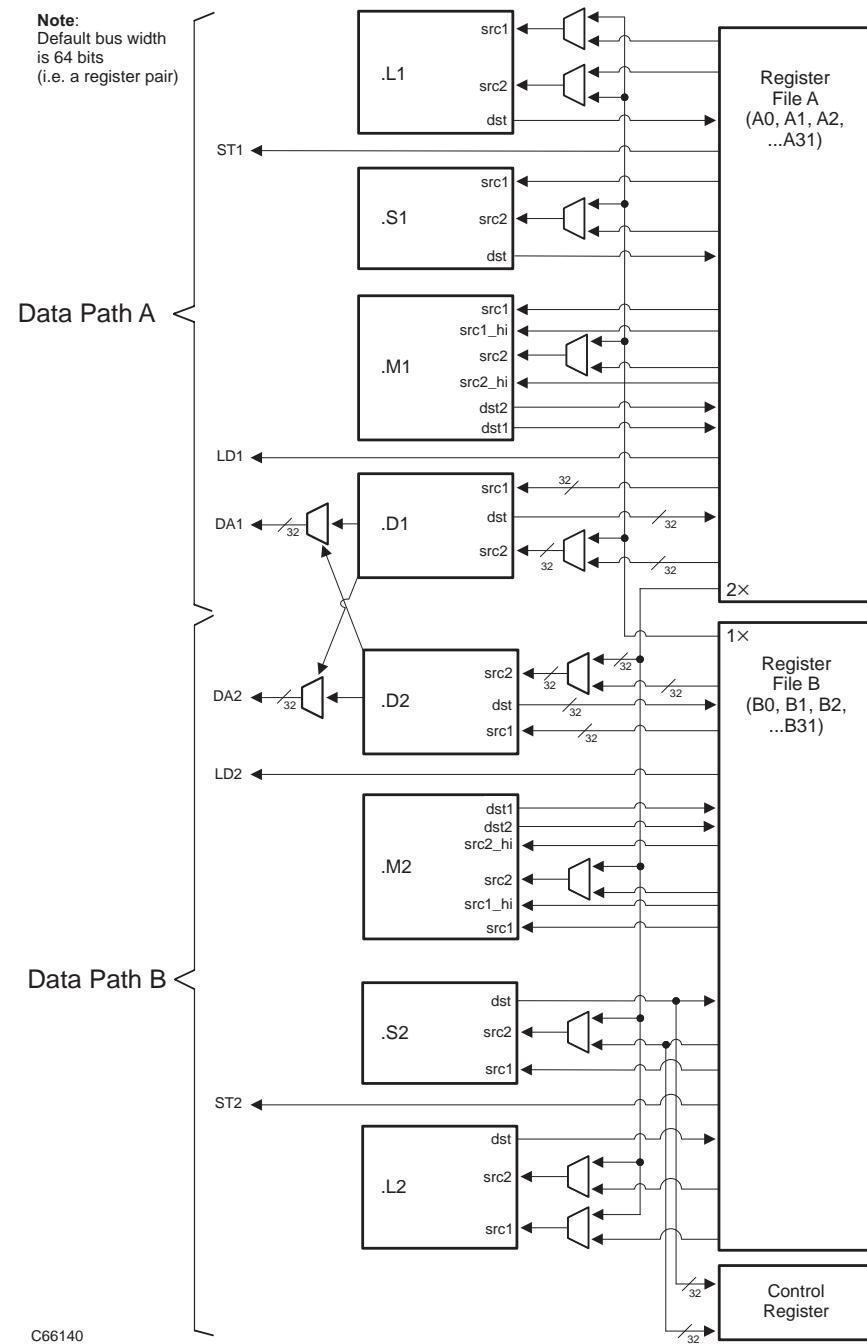
This chapter focuses on the CPU, providing information about the data paths and control registers. The two register files and the data cross paths are described.

- 2.1 ["Introduction" on page 2-2](#)
- 2.2 ["General-Purpose Register Files" on page 2-3](#)
- 2.3 ["Functional Units" on page 2-5](#)
- 2.4 ["Register File Cross Paths" on page 2-6](#)
- 2.5 ["Memory, Load, and Store Paths" on page 2-6](#)
- 2.6 ["Data Address Paths" on page 2-7](#)
- 2.7 ["Galois Field" on page 2-7](#)
- 2.8 ["Control Register File" on page 2-10](#)
- 2.9 ["Control Register File Extensions" on page 2-24](#)
- 2.10 ["Control Register File Extensions for Floating-Point Operations" on page 2-34](#)

## 2.1 Introduction

The components of the data path for the CPU are shown in [Figure 2-1](#) on page 2-3. These components consist of:

- Two general-purpose register files (A and B)
- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
- Two load-from-memory data paths (LD1 and LD2)
- Two store-to-memory data paths (ST1 and ST2)
- Two data address paths (DA1 and DA2)
- Two register file data cross paths (1X and 2X)

**Figure 2-1 CPU Data Paths**


## 2.2 General-Purpose Register Files

As on the TMS320C64x+ and TMS320C674x, the C66x CPU contains two general purpose register files (A and B). Each of these files contains thirty-two 32-bit registers (A0-A31 for file A and B0-B31 for file B).

The DSP general purpose register files support data ranging in size from packed 8-bit through 128-bit fixed point data.

Values larger than 32 bits (such as 40-bit values and 64-bit quantities) are stored in register pairs. Values larger than 64-bits (such as 128-bit quantities) are stored in a pair of register pair (a quadruplet of registers).

Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register, eight 8-bit values or four 16-bit values or two 32-bit value in a single 64-bit register pair, and eight 16-bit values or four 32-bit values or two 64-bit values in a single 128-bit register quadruplet.

**Table 2-1 64-bit Register Pairs**

Register File	
A	B
A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14
A17:A16	B17:B16
A19:A18	B19:B18
A21:A20	B21:B20
A23:A22	B23:B22
A25:A24	B25:B24
A27:A26	B27:B26
A29:A28	B29:B28
A31:A30	B31:B30

**Table 2-2 128-bit Register Quadruplets**

Register File	
A	B
A3:A2:A1:A0	B3:B2:B1:B0
A7:A6:A5:A4	B7:B6:B5:B4
A11:A10:A9:A8	B11:B10:B9:B8
A15:A14:A13:A12	B15:B14:B13:B12
A19:A18:A17:A16	B19:B18:B17:B16
A23:A22:A21:A20	B23:B22:B21:B20
A27:A26:A25:A24	B27:B26:B25:B24
A31:A30:A29:A28	B31:B30:B29:B28

## 2.3 Functional Units

Except for the .D unit which remains unchanged vs. the C64x+ and C674x, most data lines for the .L/.S/.M in the CPU are increased.

The .L/.S units now support up to 64-bit operands. The src1 and dst ports are now 64-bit wide; src2 is also 64-bit and can directly transport long 40-bit data (the long src expansion port has been removed).

The .M units now support up to 128-bit source operands. Each source operands can use up to two 64-bit read ports (src1 and src1\_hi for the first operand, src2 and src2\_hi for the second operand). The src2 input of the .M unit is selectable between the cross path and the same-side register file.

Since the .M unit can return up to 128-bit results, it includes two 64-bit write ports (dst1 and dst2) to the register file. The dst1 and dst2 ports can be used in combination to transport a 128-bit results generated by one 4-cycle instruction or to transport two independent results (one result of 64 bits or less from a 2-cycle instruction and another result of 64 bits or less from a 4-cycle instruction).

## 2.4 Register File Cross Paths

As on previous architecture C674x and C64x+, each functional unit reads directly from and writes directly to the register file within its own data path.

On the C66x, the 1x and 2x cross paths allow functional units from one data path to access a 64-bit operand from the opposite side register file. On C64x+ and C674x, the cross paths are limited to 32-bit operand.

Therefore the C66x enables new combinations of operands for existing C64x+/C674x instructions. The list of C64x+/C674x instructions that can now use the crosspath for long 40-bit data is:

ABS	.Lx	xlong,	slong	
ADD	.Lx	scst5,	xlong,	slong
CMPEQ	.Lx	scst5,	xlong,	uint
CMPGT	.Lx	scst5,	xlong,	uint
CMPGTU	.Lx	ucst5,	xulong,	uint
CMPLT	.Lx	scst5,	xlong,	uint
CMPLTU	.Lx	ucst5,	xlong,	uint
NORM	.Lx	xlong,	uint	
SADD	.Lx	scst5,	xlong,	slong
SAT	.Lx	xlong,	slong	
SHL	.Sx	xlong,	(uint or ucst5),	slong
SHR	.Sx	xlong,	(uint or ucst5),	slong
SHRU	.Sx	xulong,	(uint or ucst5),	ulong
SSUB	.Lx	scst5,	xlong,	slong
SUB	.Lx	scst5,	xlong,	slong

## 2.5 Memory, Load, and Store Paths

The DSP supports doubleword loads and stores. There are four 32-bit paths for loading data from memory to the register file. For side A, LD1a is the load path for the 32 LSBs and LD1b is the load path for the 32 MSBs. For side B, LD2a is the load path for the 32 LSBs and LD2b is the load path for the 32 MSBs. There are also four 32-bit paths for storing register values to memory from each register file. For side A, ST1a is the write path for the 32 LSBs and ST1b is the write path for the 32 MSBs. For side B, ST2a is the write path for the 32 LSBs and ST2b is the write path for the 32 MSBs.

On the C6000 architecture, some of the ports for long and doubleword operands are shared between functional units. This places a constraint on which long or doubleword operations can be scheduled on a data path in the same execute packet. See Section [3.8.7](#) on page 3-18.

## 2.6 Data Address Paths

The data address paths (DA1 and DA2) are each connected to the .D units in both data paths. This allows data addresses generated by any one path to access data to or from any register.

The DA1 and DA2 resources and their associated data paths are specified as T1 and T2, respectively. T1 consists of the DA1 address path and the LD1 and ST1 data paths. For the DSP, LD1 is comprised of LD1a and LD1b to support 64-bit loads; ST1 is comprised of ST1a and ST1b to support 64-bit stores. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. For the DSP, LD2 is comprised of LD2a and LD2b to support 64-bit loads; ST2 is comprised of ST2a and ST2b to support 64-bit stores.

The T1 and T2 designations appear in the functional unit fields for load and store instructions. For example, the following load instruction uses the .D1 unit to generate the address but is using the LD2 path resource from DA2 to place the data in the B register file. The use of the DA2 resource is indicated with the T2 designation.

```
LDW .D1T2 *A0 [3], B1
```

## 2.7 Galois Field

Modern digital communication systems typically make use of error correction coding schemes to improve system performance under imperfect channel conditions. The scheme most commonly used is the Reed-Solomon code, due to its robustness against burst errors and its relative ease of implementation.

The DSP contains Galois field multiply hardware that is used for Reed-Solomon encode and decode functions. To understand the relevance of the Galois field multiply hardware, it is necessary to first define some mathematical terms.

Two kinds of number systems that are common in algorithm development are integers and real numbers. For integers, addition, subtraction, and multiplication operations can be performed. Division can also be performed, if a nonzero remainder is allowed. For real numbers, all four of these operations can be performed, even if there is a nonzero remainder for division operations.

Real numbers can belong to a mathematical structure called a field. A field consists of a set of data elements along with addition, subtraction, multiplication, and division. A field of integers can also be created if modulo arithmetic is performed.

An example is doing arithmetic using integers modulo 2. Perform the operations using normal integer arithmetic and then take the result modulo 2. [Table 2-3](#) illustrates addition, subtraction, and multiplication modulo 2.

**Table 2-3 Modulo 2 Arithmetic**

Addition			Subtraction			Multiplication		
+	0	1	-	0	1	x	0	1
0	0	1	0	0	1	0	0	0
1	1	0	1	1	0	1	0	1

Note that addition and subtraction results are the same, and in fact are equivalent to the XOR (exclusive-OR) operation in binary. Also, the multiplication result is equal to the AND operation in binary. These properties are unique to modulo 2 arithmetic, but modulo 2 arithmetic is used extensively in error correction coding. Another more general property is that division by any nonzero element is now defined. Division can always be performed, if every element other than zero has a multiplicative inverse:

$$x \times x^{-1} = 1$$

Another example, arithmetic modulo 5, illustrates this concept more clearly. The addition, subtraction, and multiplication tables are given in [Table 2-4](#).

**Table 2-4 Modulo 5 Arithmetic**

Addition						Subtraction						Multiplication					
+	0	1	2	3	4	-	0	1	2	3	4	×	0	1	2	3	4
0	0	1	2	3	4	0	0	4	3	2	1	0	0	0	0	0	0
1	1	2	3	4	0	1	1	0	4	3	2	1	0	1	2	3	4
2	2	3	4	0	1	2	2	1	0	4	3	2	0	2	4	1	3
3	3	4	0	1	2	3	3	2	1	0	4	3	0	3	1	4	2
4	4	0	1	2	3	4	4	3	2	1	0	4	0	4	3	2	1

In the rows of the multiplication table, element 1 appears in every nonzero row and column. Every nonzero element can be multiplied by at least one other element for a result equal to 1. Therefore, division always works and arithmetic over integers modulo 5 forms a field. Fields generated in this manner are called finite fields or Galois fields and are written as GF(X), such as GF(2) or GF(5). They only work when the arithmetic performed is modulo a prime number.

Galois fields can also be formed where the elements are vectors instead of integers if polynomials are used. Finite fields, therefore, can be found with a number of elements equal to any power of a prime number. Typically, we are interested in implementing error correction coding systems using binary arithmetic. All of the fields that are dealt with in Reed Solomon coding systems are of the form GF( $2^m$ ). This allows performing addition using XORs on the coefficients of the vectors, and multiplication using a combination of ANDs and XORs.

A final example considers the field GF( $2^3$ ), which has 8 elements. This can be generated by arithmetic modulo the (irreducible) polynomial  $P(x) = x^3 + x + 1$ . Elements of this field look like vectors of three bits. [Table 2-5](#) shows the addition and multiplication tables for field GF( $2^3$ ).

Note that the value 1 (001) appears in every nonzero row of the multiplication table, which indicates that this is a valid field.

The channel error can now be modeled as a vector of bits, with a one in every bit position that an error has occurred, and a zero where no error has occurred. Once the error vector has been determined, it can be subtracted from the received message to determine the correct code word.

The Galois field multiply hardware on the DSP is named GMPY4. The **GMPY4** instruction performs four parallel operations on 8-bit packed data on the .M unit. The Galois field multiplier can be programmed to perform all Galois multiplies for fields of the form  $GF(2^m)$ , where  $m$  can range between 1 and 8 using any generator polynomial. The field size and the polynomial generator are controlled by the Galois field polynomial generator function register (GFPGR).

In addition to the **GMPY4** instruction, the C674x DSP has the **GMPY** instruction that uses either the GPLYA or GPLYB control register as a source for the polynomial (depending on whether the A or B side functional unit is used) and produces a 32-bit result.

The GFPGR, shown in [Figure 2-5](#) and described in [Table 2-10](#), contains the Galois field polynomial generator and the field size control bits. These bits control the operation of the **GMPY4** instruction. GFPGR can only be set via the **MVC** instruction. The default function after reset for the **GMPY4** instruction is field size = 7h and polynomial = 1Dh.

### 2.7.1 Special Timing Considerations

If the next execute packet after an **MVC** instruction that changes the GFPGR value contains a **GMPY4** instruction, then the **GMPY4** is controlled by the newly loaded GFPGR value.

**Table 2-5 Modulo Arithmetic for Field  $GF(2^3)$**

		Addition							
+		000	001	010	011	100	101	110	111
000	000	001	010	011	100	101	110	111	
001	001	000	011	010	101	100	111	110	
010	010	011	000	001	110	111	100	101	
011	011	010	001	000	111	110	101	100	
100	100	101	110	111	000	001	010	011	
101	101	100	111	110	001	000	011	010	
110	110	111	100	101	010	011	000	001	
111	111	110	101	100	011	010	001	000	

		Multiplication							
x		000	001	010	011	100	101	110	111
000	000	000	000	000	000	000	000	000	000
001	000	001	010	011	100	101	110	111	
010	000	010	100	110	011	001	111	101	
011	000	011	110	101	111	100	001	010	
100	000	100	011	111	110	010	101	001	
101	000	101	001	100	010	111	011	110	
110	000	110	111	001	101	011	010	100	
111	000	111	101	010	001	110	100	011	

## 2.8 Control Register File

Table 2-6 lists the control registers contained in the control register file.

**Table 2-6 Control Registers**

Acronym	Register Name	Section
AMR	Addressing mode register	2.8.3 <a href="#">"Addressing Mode Register (AMR)" on page 2-12</a>
CSR	Control status register	2.8.4 <a href="#">"Control Status Register (CSR)" on page 2-15</a>
GFPGFR	Galois field multiply control register	2.8.5 <a href="#">"Galois Field Polynomial Generator Function Register (GFPGFR)" on page 2-17</a>
ICR	Interrupt clear register	2.8.6 <a href="#">"Interrupt Clear Register (ICR)" on page 2-18</a>
IER	Interrupt enable register	2.8.7 <a href="#">"Interrupt Enable Register (IER)" on page 2-19</a>
IFR	Interrupt flag register	2.8.8 <a href="#">"Interrupt Flag Register (IFR)" on page 2-20</a>
IRP	Interrupt return pointer register	2.8.9 <a href="#">"Interrupt Return Pointer Register (IRP)" on page 2-20</a>
ISR	Interrupt set register	2.8.10 <a href="#">"Interrupt Set Register (ISR)" on page 2-21</a>
ISTP	Interrupt service table pointer register	2.8.11 <a href="#">"Interrupt Service Table Pointer Register (ISTP)" on page 2-22</a>
NRP	Nonmaskable interrupt return pointer register	2.8.12 <a href="#">"Nonmaskable Interrupt (NMI) Return Pointer Register (NRP)" on page 2-23</a>
PCE1	Program counter, E1 phase	2.8.13 <a href="#">"E1 Phase Program Counter (PCE1)" on page 2-23</a>
<b>Control Register File Extensions</b>		
DNUM	DSP core number register	2.9.1 <a href="#">"DSP Core Number Register (DNUM)" on page 2-24</a>
ECR	Exception clear register	2.9.2 <a href="#">"Exception Clear Register (ECR)" on page 2-24</a>
EFR	Exception flag register	2.9.3 <a href="#">"Exception Flag Register (EFR)" on page 2-25</a>
GPLYA	GMPY A-side polynomial register	2.9.4 <a href="#">"GMPY Polynomial—A Side Register (GPLYA)" on page 2-25</a>
GPLYB	GMPY B-side polynomial register	2.9.5 <a href="#">"GMPY Polynomial—B Side Register (GPLYB)" on page 2-26</a>
IERR	Internal exception report register	2.9.6 <a href="#">"Internal Exception Report Register (IERR)" on page 2-27</a>
ILC	Inner loop count register	2.9.7 <a href="#">"SPLOOP Inner Loop Count Register (ILC)" on page 2-28</a>
ITSR	Interrupt task state register	2.9.8 <a href="#">"Interrupt Task State Register (ITSR)" on page 2-28</a>
NTSR	NMI/Exception task state register	2.9.9 <a href="#">"NMI/Exception Task State Register (NTSR)" on page 2-29</a>
REP	Restricted entry point address register	2.9.10 <a href="#">"Restricted Entry Point Register (REP)" on page 2-29</a>
RILC	Reload inner loop count register	2.9.11 <a href="#">"SPLOOP Reload Inner Loop Count Register (RILC)" on page 2-30</a>
SSR	Saturation status register	2.9.12 <a href="#">"Saturation Status Register (SSR)" on page 2-30</a>
TSCH	Time-stamp counter (high 32) register	2.9.13 <a href="#">"Time Stamp Counter Registers (TSCL and TSCH)" on page 2-31</a>
TSCL	Time-stamp counter (low 32) register	2.9.13 <a href="#">"Time Stamp Counter Registers (TSCL and TSCH)" on page 2-31</a>
TSR	Task state register	2.9.14 <a href="#">"Task State Register (TSR)" on page 2-33</a>
<b>Control Register File Extensions for Floating-point Operations</b>		
FADCR	Floating-point adder configuration register	2.10.1 <a href="#">"Floating-Point Adder Configuration Register (FADCR)" on page 2-35</a>
FAUCR	Floating-point auxiliary configuration register	2.10.2 <a href="#">"Floating-Point Auxiliary Configuration Register (FAUCR)" on page 2-37</a>
FMCR	Floating-point multiplier configuration register	2.10.3 <a href="#">"Floating-Point Multiplier Configuration Register (FMCR)" on page 2-40</a>

### 2.8.1 Register Addresses for Accessing the Control Registers

[Table 4-10](#) on page 4-472 lists the register addresses for accessing the control register file. One unit (.S2) can read from and write to the control register file. Each control register is accessed by the **MVC** instruction. See the **MVC** instruction description (see [MVC](#)) for information on how to use this instruction.

Additionally, some of the control register bits are specially accessed in other ways. For example, arrival of a maskable interrupt on an external interrupt pin, INT $m$ , triggers the setting of flag bit IFR $m$ . Subsequently, when that interrupt is processed, this triggers the clearing of IFR $m$  and the clearing of the global interrupt enable bit, GIE. Finally, when that interrupt processing is complete, the **B IRP** instruction in the interrupt service routine restores the pre-interrupt value of the GIE. Similarly, saturating instructions like **SADD** set the SAT (saturation) bit in the control status register (CSR).

On the CPU, access to some of the registers is restricted when in User mode. See Chapter 9 “[CPU Privilege](#)” on page 9-1 for more information.

### 2.8.2 Pipeline/Timing of Control Register Accesses

All **MVC** instructions are single-cycle instructions that complete their access of the explicitly named registers in the E1 pipeline phase. This is true whether **MVC** is moving a general register to a control register, or conversely. In all cases, the source register content is read, moved through the .S2 unit, and written to the destination register in the E1 pipeline phase.

Pipeline Stage	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.S2

Even though **MVC** modifies the particular target control register in a single cycle, it can take extra clocks to complete modification of the non-explicitly named register. For example, the **MVC** cannot modify bits in the IFR directly. Instead, **MVC** can only write 1's into the ISR or the ICR to specify setting or clearing, respectively, of the IFR bits. **MVC** completes this ISR/ICR write in a single (E1) cycle but the modification of the IFR bits occurs one clock later. For more information on the manipulation of ISR, ICR, and IFR, see section 2.8.10 “[Interrupt Set Register \(ISR\)](#)” on page 2-21, section 2.8.6 “[Interrupt Clear Register \(ICR\)](#)” on page 2-18, and section 2.8.8 “[Interrupt Flag Register \(IFR\)](#)” on page 2-20.

Saturating instructions, such as **SADD**, set the saturation flag bit (SAT) in CSR indirectly. As a result, several of these instructions update the SAT bit one full clock cycle after their primary results are written to the register file. For example, the **SMPY** instruction writes its result at the end of pipeline stage E2; its primary result is available after one delay slot. In contrast, the SAT bit in CSR is updated one cycle later than the result is written; this update occurs after two delay slots. (For the specific behavior of an instruction, refer to the description of that individual instruction).

The **B IRP** and **B NRP** instructions directly update the GIE and NMIE bits, respectively. Because these branches directly modify CSR and IER, respectively, there are no delay slots between when the branch is issued and when the control register updates take effect.

### 2.8.3 Addressing Mode Register (AMR)

For each of the eight registers (A4-A7, B4-B7) that can perform linear or circular addressing, the addressing mode register (AMR) specifies the addressing mode. A 2-bit field for each register selects the address modification mode: linear (the default) or circular mode. With circular addressing, the field also specifies which BK (block size) field to use for a circular buffer. In addition, the buffer must be aligned on a byte boundary equal to the block size. The mode select fields and block size fields are shown in [Figure 2-2](#) and described in [Table 2-7](#).

**Figure 2-2 Addressing Mode Register (AMR)**

31	26	25	21	20	16
Reserved		BK1			BK0
R-0			R/W-0		
15	14	13	12	11	10
B7 MODE	B6 MODE	B5 MODE	B4 MODE	A7 MODE	A6 MODE
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
9	8	7	6	5	4
A5 MODE	A4 MODE			3	2
R/W-0	R/W-0			R/W-0	R/W-0
1	0				

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-7 Addressing Mode Register (AMR) Field Descriptions (Part 1 of 2)**

Bit	Field	Value	Description
31-26	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
25-21	BK1	0-1Fh	Block size field 1. A 5-bit value used in calculating block sizes for circular addressing. <a href="#">Table 2-5</a> on page 2-9 shows block size calculations for all 32 possibilities. <i>Block size (in bytes) = 2<sup>(N+1)</sup>, where N is the 5-bit value in BK1</i>
20-16	BK0	0-1Fh	Block size field 0. A 5-bit value used in calculating block sizes for circular addressing. <a href="#">Table 2-5</a> on page 2-9 shows block size calculations for all 32 possibilities. <i>Block size (in bytes) = 2<sup>(N+1)</sup>, where N is the 5-bit value in BK0</i>
15-14	B7 MODE	0-3h	Address mode selection for register file B7. 0 Linear modification (default at reset) 1h Circular addressing using the BK0 field 2h Circular addressing using the BK1 field 3h Reserved
13-12	B6 MODE	0-3h	Address mode selection for register file B6. 0 Linear modification (default at reset) 1h Circular addressing using the BK0 field 2h Circular addressing using the BK1 field 3h Reserved
11-10	B5 MODE	0-3h	Address mode selection for register file B5. 0 Linear modification (default at reset) 1h Circular addressing using the BK0 field 2h Circular addressing using the BK1 field 3h Reserved

**Table 2-7 Addressing Mode Register (AMR) Field Descriptions (Part 2 of 2)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
9-8	B4 MODE	0-3h	Address mode selection for register file B4.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
7-6	A7 MODE	0-3h	Address mode selection for register file A7.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
5-4	A6 MODE	0-3h	Address mode selection for register file A6.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
3-2	A5 MODE	0-3h	Address mode selection for register file a5.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved
1-0	A4 MODE	0-3h	Address mode selection for register file A4.
		0	Linear modification (default at reset)
		1h	Circular addressing using the BK0 field
		2h	Circular addressing using the BK1 field
		3h	Reserved

**Table 2-8 Block Size Calculations (Part 1 of 2)**

<b>BKn Value</b>	<b>Block Size</b>	<b>BKn Value</b>	<b>Block Size</b>
00000	2	10000	131072
00001	4	10001	262144
00010	8	10010	524288
00011	16	10011	1048576
00100	32	10100	2097152
00101	64	10101	4194304
00110	128	10110	8388608
00111	256	10111	16777216
01000	512	11000	33554432
01001	1024	11001	67108864
01010	2048	11010	134217728
01011	4096	11011	268435456
01100	8192	11100	536870912

**Table 2-8 Block Size Calculations (Part 2 of 2)**

<b>BKn Value</b>	<b>Block Size</b>	<b>BKn Value</b>	<b>Block Size</b>
01101	16384	11101	1073741824
01110	32768	11110	2147483648
01111	65536	11111	4294967296

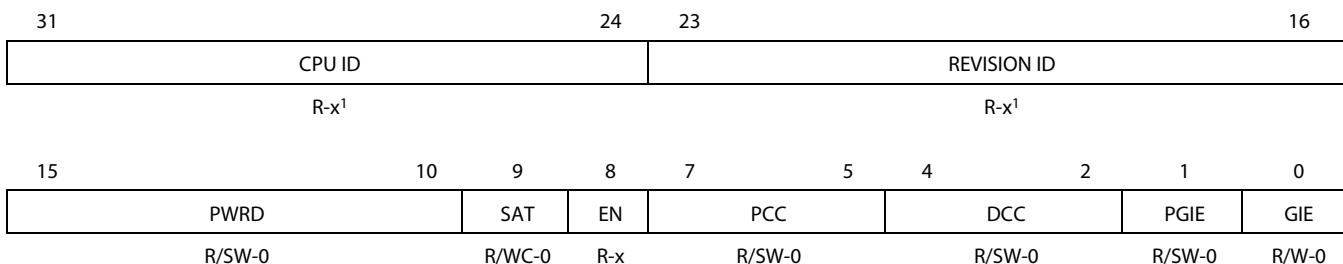
### 2.8.4 Control Status Register (CSR)

The control status register (CSR) contains control and status bits. The CSR is shown in [Figure 2-3](#) and described in [Table 2-9](#). For the PWRD, EN, PCC, and DCC fields, see the device-specific data sheet to see if it supports the options that these fields control. The PCC and DCC fields are ignored on the C674x CPU.

The power-down modes and their wake-up methods are programmed by the PWRD field (bits 15-10) of CSR. The PWRD field of CSR is shown in. When writing to CSR, all bits of the PWRD field should be configured at the same time. A logic 0 should be used when writing to the reserved bit (bit 15) of the PWRD field.

The PWRD, PCC, DCC, and PGIE fields cannot be written in User mode. The PCC and DCC fields can only be modified in Supervisor mode. See Chapter 9 “[CPU Privilege](#)” on page 9-1 for more information.

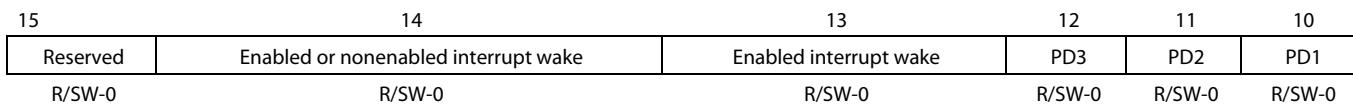
**Figure 2-3 Control Status Register (CSR)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; SW = Writable by the **MVC** instruction only in supervisor mode; WC = Bit is cleared on write; -n = value after reset; -x = value is indeterminate after reset

1. See the device-specific data sheet for the default value of this field.

**Figure 2-4 PWRD Field of Control Status Register (CSR)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset; SW = Writable by the **MVC** instruction only in supervisor mode; -n = value after reset

**Table 2-9 Control Status Register (CSR) Field Descriptions (Part 1 of 2)**

Bit	Field	Value	Description
31-24	CPU ID	0-FFh 0-13h 14h 15h-FFh	Identifies the CPU of the device. Not writable by the <b>MVC</b> instruction. Reserved C674x CPU Reserved
23-16	REVISION ID	0-FFh	Identifies silicon revision of the CPU. For the most current silicon revision information, see the device-specific data sheet. Not writable by the <b>MVC</b> instruction.

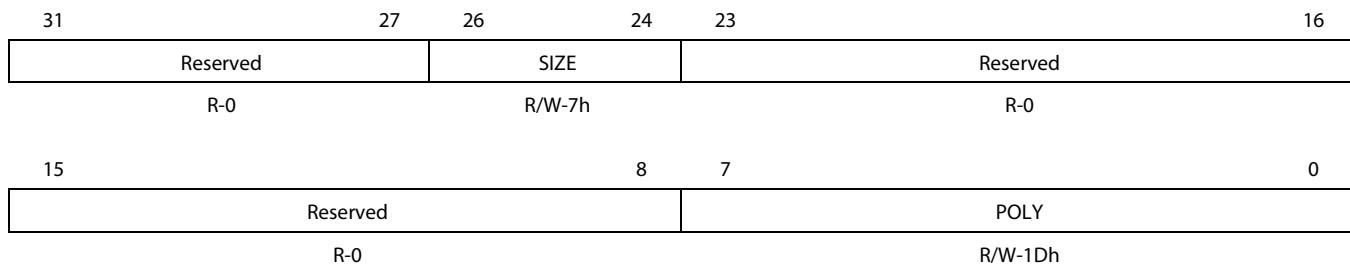
**Table 2-9 Control Status Register (CSR) Field Descriptions (Part 2 of 2)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
15-10	PWRD	0-3Fh 0 1h-8h 9h Ah-10h 11h 12h-19h 1Ah 1Bh 1Ch 1D-3Fh	Power-down mode field. See <a href="#">Figure 2-4</a> . Writable by the <b>MVC</b> instruction only in Supervisor mode. No power-down. Reserved Power-down mode PD1; wake by an enabled interrupt. Reserved Power-down mode PD1; wake by an enabled or nonenabled interrupt. Reserved Power-down mode PD2; wake by a device reset. Reserved Power-down mode PD3; wake by a device reset. Reserved
9	SAT	0 1	Saturate bit. Can be cleared only by the <b>MVC</b> instruction and can be set only by a functional unit. The set by a functional unit has priority over a clear (by the <b>MVC</b> instruction), if they occur on the same cycle. The SAT bit is set one full cycle (one delay slot) after a saturate occurs. The SAT bit will not be modified by a conditional instruction whose condition is false. No functional units generated saturated results. One or more functional units performed an arithmetic operation which resulted in saturation.
8	EN	0 1	Endian mode. Not writable by the <b>MVC</b> instruction. Big endian Little endian
7-5	PCC	0-7h 0-7h	Program cache control mode. This field is ignored on the C674x CPU. Reserved
4-2	DCC	0-7h 0-7h	Data cache control mode. This field is ignored on the C674x CPU. Reserved
1	PGIE	0 1	Previous GIE (global interrupt enable). This bit contains a copy of the GIE bit at the point when interrupt is taken. It is physically the same bit as GIE bit in the interrupt task state register (ITSR). Writable by the <b>MVC</b> instruction only in Supervisor mode; not writable in User mode. Interrupts will be disabled after return from interrupt. Interrupts will be enabled after return from interrupt.
0	GIE	0 1	Global interrupt enable. Physically the same bit as GIE bit in the task state register (TSR). Writable by the <b>MVC</b> instruction in Supervisor and User mode. See section <a href="#">6.2</a> on page 6-10 for details on how the GIE bit affects interruptibility. Disables all interrupts, except the reset interrupt and NMI (nonmaskable interrupt). Enables all interrupts.

### 2.8.5 Galois Field Polynomial Generator Function Register (GFPGFR)

The Galois field polynomial generator function register (GFPGFR) controls the field size and the Galois field polynomial generator of the Galois field multiply hardware. The GFPGFR is shown in [Figure 2-5](#) and described in [Table 2-10](#). The Galois field is described in [2.7](#) on page 2-7.

**Figure 2-5 Galois Field Polynomial Generator Function Register (GFPGFR)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-10 Galois Field Polynomial Generator Function Register (GFPGFR) Field Descriptions**

Bit	Field	Value	Description
31-27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
26-24	SIZE	0-7h	Field size.
23-8	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
7-0	POLY	0-FFh	Polynomial generator.

## 2.8.6 Interrupt Clear Register (ICR)

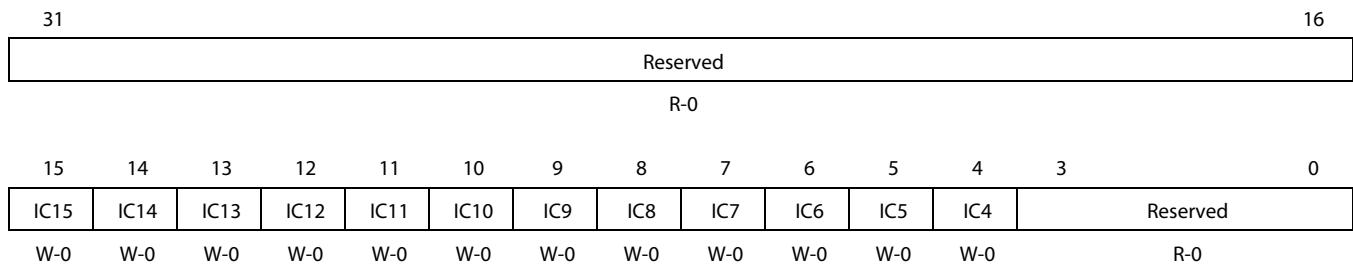
The interrupt clear register (ICR) allows you to manually clear the maskable interrupts (INT15-INT4) in the interrupt flag register (IFR). Writing a 1 to any of the bits in ICR causes the corresponding interrupt flag ( $IF_n$ ) to be cleared in IFR. Writing a 0 to any bit in ICR has no effect. Incoming interrupts have priority and override any write to ICR. You cannot set any bit in ICR to affect NMI or reset. The ISR is shown in [Figure 2-6](#) and described in [Table 2-11](#). See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts.



**Note**—Any write to ICR (by the **MVC** instruction) effectively has one delay slot because the results cannot be read (by the **MVC** instruction) in IFR until two cycles after the write to ICR.

Any write to ICR is ignored by a simultaneous write to the same bit in the interrupt set register (ISR).

**Figure 2-6**    **Interrupt Clear Register (ICR)**



LEGEND: R = Read only; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-11**    **Interrupt Clear Register (ICR) Field Descriptions**

Bit	Field	Value	Description
31-16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15-4	IC $n$		Interrupt clear.
		0	Corresponding interrupt flag ( $IF_n$ ) in IFR is not cleared.
		1	Corresponding interrupt flag ( $IF_n$ ) in IFR is cleared.
3-0	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

### 2.8.7 Interrupt Enable Register (IER)

The interrupt enable register (IER) enables and disables individual interrupts. The IER is shown in [Figure 2-7](#) and described in [Table 2-12](#).

The IER is not accessible in User mode. See [9.2.4.1](#) on page 9-2 for more information. See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts.

**Figure 2-7** Interrupt Enable Register (IER)

31	Reserved															16
R-0																
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
IE15	IE14	IE13	IE12	IE11	IE10	IE9	IE8	IE7	IE6	IE5	IE4	Reserved	NMIE	1		

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-12** Interrupt Enable Register (IER) Field Descriptions

Bit	Field	Value	Description
31-16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15-4	IEn	0 1	Interrupt enable. An interrupt triggers interrupt processing only if the corresponding bit is set to 1. Interrupt is disabled. Interrupt is enabled.
3-2	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	NMIE	0 1	Nonmaskable interrupt enable. An interrupt triggers interrupt processing only if the bit is set to 1. The NMIE bit is cleared at reset. After reset, you must set the NMIE bit to enable the NMI and to allow INT15-INT4 to be enabled by the GIE bit in CSR and the corresponding IER bit. You cannot manually clear the NMIE bit; a write of 0 has no effect. The NMIE bit is also cleared by the occurrence of an NMI. All nonreset interrupts are disabled. All nonreset interrupts are enabled. The NMIE bit is set only by completing a <b>B NRP</b> instruction or by a write of 1 to the NMIE bit.
0	1	1	Reset interrupt enable. You cannot disable the reset interrupt.

### 2.8.8 Interrupt Flag Register (IFR)

The interrupt flag register (IFR) contains the status of INT4-INT15 and NMI interrupt. Each corresponding bit in the IFR is set to 1 when that interrupt occurs; otherwise, the bits are cleared to 0. If you want to check the status of interrupts, use the **MVC** instruction to read the IFR. (See the **MVC** instruction description (“**MVC**”) for information on how to use this instruction.) The IFR is shown in [Figure 2-8](#) and described in [Table 2-13](#). See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts.

**Figure 2-8**    **Interrupt Flag Register (IFR)**

31	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	16
Reserved																
R-0																
IF15	IF14	IF13	IF12	IF11	IF10	IF9	IF8	IF7	IF6	IF5	IF4	Reserved	NMIF	0		
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	

LEGEND: R = Readable by the **MVC** instruction; -n = value after reset

**Table 2-13**    **Interrupt Flag Register (IFR) Field Descriptions**

Bit	Field	Value	Description
31-16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15-4	IF $n$	0 1	Interrupt flag. Indicates the status of the corresponding maskable interrupt. An interrupt flag may be manually set by setting the corresponding bit (IS $n$ ) in the interrupt set register (ISR) or manually cleared by setting the corresponding bit (IC $n$ ) in the interrupt clear register (ICR).  0: Interrupt has not occurred. 1: Interrupt has occurred.
3-2	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
1	NMIF	0 1	Nonmaskable interrupt flag.  0: Interrupt has not occurred. 1: Interrupt has occurred.
0	0	0	Reset interrupt flag.

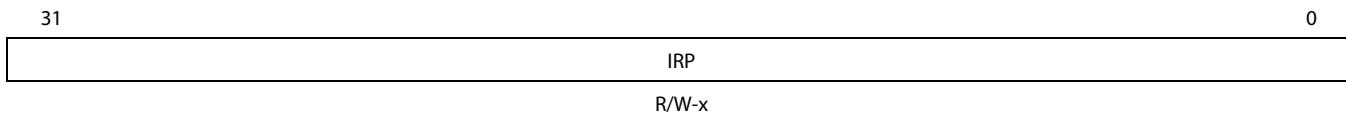
### 2.8.9 Interrupt Return Pointer Register (IRP)

The interrupt return pointer register (IRP) contains the return pointer that directs the CPU to the proper location to continue program execution after processing a maskable interrupt. A branch using the address in IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. The IRP is shown in [Figure 2-9](#).

The IRP contains the 32-bit address of the first execute packet in the program flow that was not executed because of a maskable interrupt. Although you can write a value to IRP, any subsequent interrupt processing may overwrite that value.

See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts.

**Figure 2-9     Interrupt Return Pointer Register (IRP)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -x = value is indeterminate after reset

### 2.8.10 Interrupt Set Register (ISR)

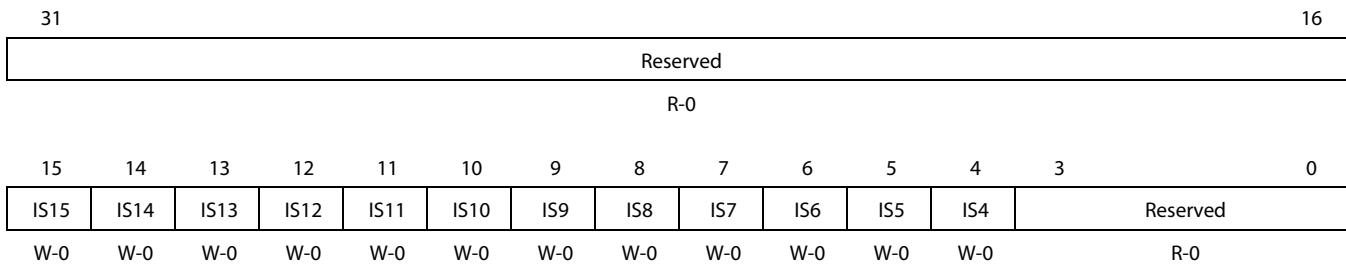
The interrupt set register (ISR) allows you to manually set the maskable interrupts (INT15-INT4) in the interrupt flag register (IFR). Writing a 1 to any of the bits in ISR causes the corresponding interrupt flag (IF $n$ ) to be set in IFR. Writing a 0 to any bit in ISR has no effect. You cannot set any bit in ISR to affect NMI or reset. The ISR is shown in [Figure 2-10](#) and described in [Table 2-14](#). See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts.



**Note**—Any write to ISR (by the **MVC** instruction) effectively has one delay slot because the results cannot be read (by the **MVC** instruction) in IFR until two cycles after the write to ISR.

Any write to the interrupt clear register (ICR) is ignored by a simultaneous write to the same bit in ISR.

**Figure 2-10     Interrupt Set Register (ISR)**



LEGEND: R = Read only; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-14     Interrupt Set Register (ISR) Field Descriptions**

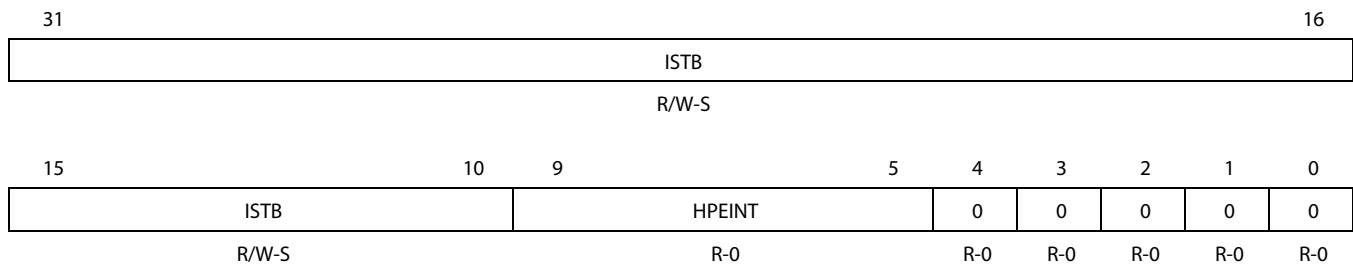
Bit	Field	Value	Description
31-16	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
15-4	IS $n$	0	Interrupt set.
		1	Corresponding interrupt flag (IF $n$ ) in IFR is not set.
3-0	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

### 2.8.11 Interrupt Service Table Pointer Register (ISTP)

The interrupt service table pointer register (ISTP) is used to locate the interrupt service routine (ISR). The ISTB field identifies the base portion of the address of the interrupt service table (IST) and the HPEINT field identifies the specific interrupt and locates the specific fetch packet within the IST. The ISTP is shown in [Figure 2-11](#) and described in [Table 2-15](#). See [6.1.1.6](#) on page 6-8 for a discussion of the use of the ISTP.

The ISTP is not accessible in User mode. See [9.2.4.1](#) on page 9-2 for more information. See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts.

**Figure 2-11** Interrupt Service Table Pointer Register (ISTP)



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset; S = See the device-specific data manual for the default value of this field after reset

**Table 2-15** Interrupt Service Table Pointer Register (ISTP) Field Descriptions

Bit	Field	Value	Description
31-10	ISTB	0-3FFFFh	Interrupt service table base portion of the IST address. This field is cleared to a device-specific default value on reset; therefore, upon startup the IST must reside at this specific address. See the device-specific data manual for more information. After reset, you can relocate the IST by writing a new value to ISTB. If relocated, the first ISFP (corresponding to <u>RESET</u> ) is never executed via interrupt processing, because reset clears the ISTB to its default value. See <a href="#">Example 6-1</a> on page 6-8.
9-5	HPEINT	0-1Fh	Highest priority enabled interrupt that is currently pending. This field indicates the number (related bit position in the IFR) of the highest priority interrupt (as defined in <a href="#">Table 6-1</a> on page 6-3) that is enabled by its bit in the IER. Thus, the ISTP can be used for manual branches to the highest priority enabled interrupt. If no interrupt is pending and enabled, HPEINT contains the value 0. The corresponding interrupt need not be enabled by NMIE (unless it is NMI) or by GIE.
4-0	0	0	Cleared to 0 (fetch packets must be aligned on 8-word (32-byte) boundaries).

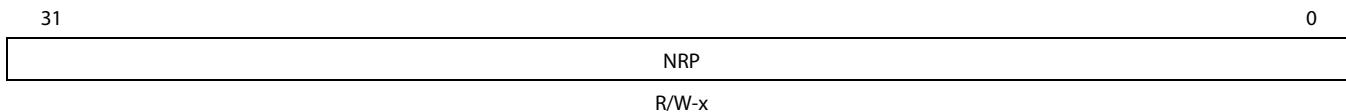
### 2.8.12 Nonmaskable Interrupt (NMI) Return Pointer Register (NRP)

The NMI return pointer register (NRP) contains the return pointer that directs the CPU to the proper location to continue program execution after NMI processing. A branch using the address in NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. The NRP is shown in [Figure 2-12](#).

The NRP contains the 32-bit address of the first execute packet in the program flow that was not executed because of a nonmaskable interrupt. Although you can write a value to NRP, any subsequent interrupt processing may overwrite that value.

See Chapter 6 “[Interrupts](#)” on page 6-1 for more information on interrupts. See Chapter 7 “[CPU Exceptions](#)” on page 7-1 for more information on exceptions.

**Figure 2-12**    **NMI Return Pointer Register (NRP)**

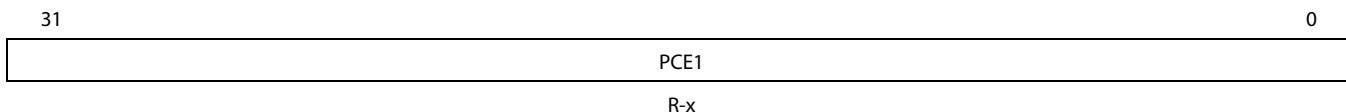


LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -x = value is indeterminate after reset

### 2.8.13 E1 Phase Program Counter (PCE1)

The E1 phase program counter (PCE1), shown in [Figure 2-13](#), contains the 32-bit address of the fetch packet in the E1 pipeline phase.

**Figure 2-13**    **E1 Phase Program Counter (PCE1)**



LEGEND: R = Readable by the **MVC** instruction; -x = value is indeterminate after reset

## 2.9 Control Register File Extensions

Table 2-16 on page 2-24 lists the additional control registers in the DSP.

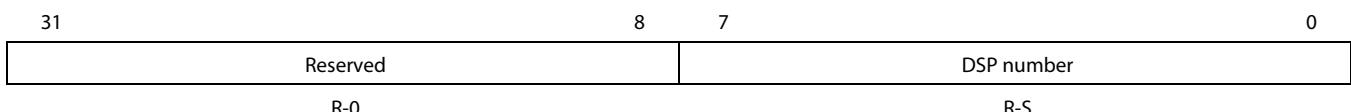
**Table 2-16 Control Register File Extensions**

Acronym	Register Name	Section
DNUM	DSP core number register	2.9.1 "DSP Core Number Register (DNUM)" on page 2-24
ECR	Exception clear register	2.9.2 "Exception Clear Register (ECR)" on page 2-24
EFR	Exception flag register	2.9.3 "Exception Flag Register (EFR)" on page 2-25
GPLYA	GMPY A-side polynomial register	2.9.4 "GMPY Polynomial—A Side Register (GPLYA)" on page 2-25
GPLYB	GMPY B-side polynomial register	2.9.5 "GMPY Polynomial—B Side Register (GPLYB)" on page 2-26
IERR	Internal exception report register	2.9.6 "Internal Exception Report Register (IERR)" on page 2-27
ILC	Inner loop count register	2.9.7 "SPLOOP Inner Loop Count Register (ILC)" on page 2-28
ITSR	Interrupt task state register	2.9.8 "Interrupt Task State Register (ITSR)" on page 2-28
NTSR	NMI/Exception task state register	2.9.9 "NMI/Exception Task State Register (NTSR)" on page 2-29
REP	Restricted entry point address register	2.9.10 "Restricted Entry Point Register (REP)" on page 2-29
RILC	Reload inner loop count register	2.9.11 "SPLOOP Reload Inner Loop Count Register (RILC)" on page 2-30
SSR	Saturation status register	2.9.12 "Saturation Status Register (SSR)" on page 2-30
TSCH	Time-stamp counter (high 32) register	2.9.13 "Time Stamp Counter Registers (TSCL and TSCH)" on page 2-31
TSCL	Time-stamp counter (low 32) register	2.9.13 "Time Stamp Counter Registers (TSCL and TSCH)" on page 2-31
TSR	Task state register	2.9.14 "Task State Register (TSR)" on page 2-33

### 2.9.1 DSP Core Number Register (DNUM)

Multiple CPUs may be used in a system. The DSP core number register (DNUM), provides an identifier to shared resources in the system which identifies which CPU is accessing those resources. The contents of this register are set to a specific value (depending on the device) at reset. See your device-specific data manual for the reset value of this register. The DNUM is shown in Figure 2-14.

**Figure 2-14 DSP Core Number Register (DNUM)**



LEGEND: R = Readable by the **MVC** instruction; -n = value after reset; S = See the device-specific data manual for the default value of this field after reset

### 2.9.2 Exception Clear Register (ECR)

The exception clear register (ECR) is used to clear individual bits in the exception flag register (EFR). Writing a 1 to any bit in ECR clears the corresponding bit in EFR.

The ECR is not accessible in User mode. See 9.2.4.1 "Restricted Control Register Access in User Mode" on page 9-2 for more information. See Chapter 7 "CPU Exceptions" on page 7-1 for more information on exceptions.

### 2.9.3 Exception Flag Register (EFR)

The exception flag register (EFR) contains bits that indicate which exceptions have been detected. Clearing the EFR bits is done by writing a 1 to the corresponding bit position in the exception clear register (ECR). Writing a 0 to the bits in this register has no effect. The EFR is shown in [Figure 2-15](#) and described in [Table 2-17](#).

The EFR is not accessible in User mode. See [9.2.4.1](#) on page 9-2 for more information. See Chapter 7 “[CPU Exceptions](#)” on page 7-1 for more information on exceptions.

**Figure 2-15**    **Exception Flag Register (EFR)**

31	30	29	16
NXF	EXF		Reserved
R/W-0	R/W-0		R-0
15			2      1      0
		Reserved	IXF      SXF
		R-0	R/W-0      R/W-0

LEGEND: R = Readable by the **MVC** EFR instruction only in Supervisor mode; W = Clearable by the **MVC** ECR instruction only in Supervisor mode; -n = value after reset

**Table 2-17**    **Exception Flag Register (EFR) Field Descriptions**

Bit	Field	Value	Description
31	NXF	0	NMI exception flag. NMI exception has not been detected.
		1	NMI exception has been detected.
30	EXF	0	EXCEP flag. Exception has not been detected.
		1	Exception has been detected.
29-2	Reserved	0	Reserved. Read as 0.
1	IXF	0	Internal exception flag. Internal exception has not been detected.
		1	Internal exception has been detected.
0	SXF	0	Software exception flag (set by <b>SWE</b> or <b>SWENR</b> instructions). Software exception has not been detected.
		1	Software exception has been detected.

### 2.9.4 GMPY Polynomial—A Side Register (GPLYA)

The **GMPY** instruction (see [GMPY](#)) uses the 32-bit polynomial in the GMPY polynomial—A side register (GPLYA), [Figure 2-16](#), when the instruction is executed on the M1 unit.

**Figure 2-16**    **GMPY Polynomial Af-Side Register (GPLYA)**

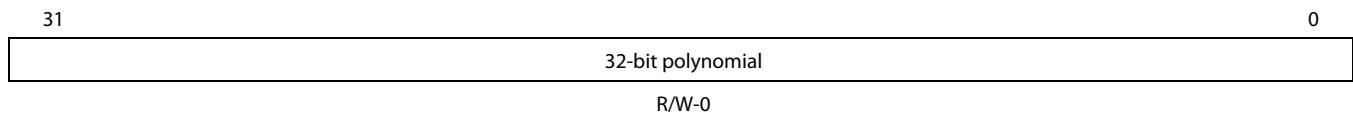
31	0
32-bit polynomial	
	R/W-0

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

### 2.9.5 GMPY Polynomial—B Side Register (GPLYB)

The GMPY instruction (see [GMPY](#)) uses the 32-bit polynomial in the GMPY polynomial—B side register (GPLYB), [Figure 2-17](#), when the instruction is executed on the M2 unit.

**Figure 2-17 GMPY Polynomial B-Side (GPLYB)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

## 2.9.6 Internal Exception Report Register (IERR)

The internal exception report register (IERR) contains flags that indicate the cause of the internal exception. In the case of simultaneous internal exceptions, the same flag may be set by different exception sources. In this case, it may not be possible to determine the exact causes of the individual exceptions. The IERR is shown in Figure 2-18 and described in Table 2-18.

The IERR is not accessible in User mode. See 9.2.4.1 on page 9-2 for more information. See Chapter 7 “CPU Exceptions” on page 7-1 for more information on exceptions.

**Figure 2-18 Internal Exception Report Register (IERR)**

31	9	8	7	6	5	4	3	2	1	0
Reserved	MSX	LBX	PRX	RAX	RCX	OPX	EPX	FPX	IFX	
R-0	R/W-0									

LEGEND: R = Readable by the **MVC** instruction only in Supervisor mode; W = Writable by the **MVC** instruction only in Supervisor mode; -n = value after reset

**Table 2-18 Internal Exception Report Register (IERR) Field Descriptions (Part 1 of 2) (Part 1 of 2)**

Bit	Field	Value	Description
31-9	Reserved	0	Reserved. Read as 0.
8	MSX	0	Missed stall exception
		1	Missed stall exception is not the cause.
		1	Missed stall exception is the cause.
7	LBX	0	SPLOOP buffer exception
		1	SPLOOP buffer exception is not the cause.
		1	SPLOOP buffer exception is the cause.
6	PRX	0	Privilege exception
		1	Privilege exception is not the cause.
		1	Privilege exception is the cause.
5	RAX	0	Resource access exception
		1	Resource access exception is not the cause.
		1	Resource access exception is the cause.
4	RCX	0	Resource conflict exception
		1	Resource conflict exception is not the cause.
		1	Resource conflict exception is the cause.
3	OPX	0	Opcode exception
		1	Opcode exception is not the cause.
		1	Opcode exception is the cause.
2	EPX	0	Execute packet exception
		1	Execute packet exception is not the cause.
		1	Execute packet exception is the cause.
1	FPX	0	Fetch packet exception
		1	Fetch packet exception is not the cause.
		1	Fetch packet exception is the cause.

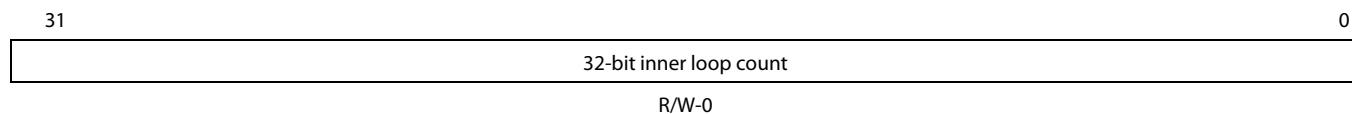
**Table 2-18 Internal Exception Report Register (IERR) Field Descriptions (Part 2 of 2) (Part 2 of 2)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
0	IFX	0	Instruction fetch exception
			Instruction fetch exception is not the cause.
		1	Instruction fetch exception is the cause.

### 2.9.7 SPLOOP Inner Loop Count Register (ILC)

The **SPLOOP** or **SPLOOPD** instructions use the SPLOOP inner loop count register (ILC), Figure 2-19, as the count of the number of iterations left to perform. The ILC content is decremented at each stage boundary until the ILC content reaches 0.

**Figure 2-19**      **Inner Loop Count Register (ILC)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

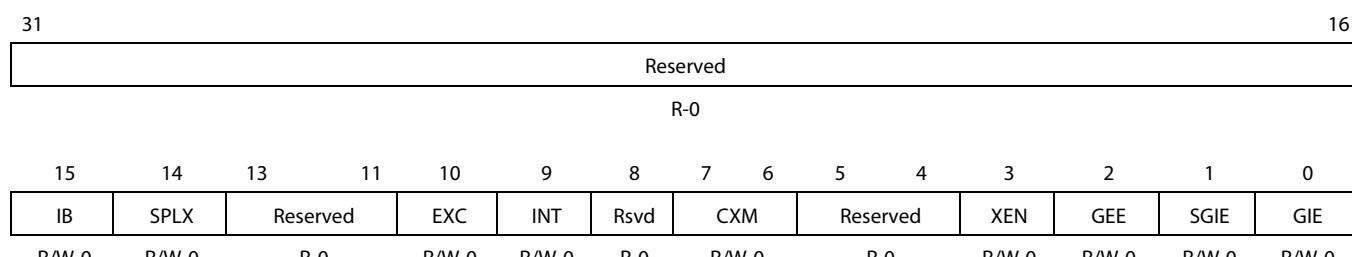
### 2.9.8 Interrupt Task State Register (ITSR)

The interrupt task state register (ITSR) is used to store the contents of the task state register (TSR) in the event of an interrupt. The ITSR is shown in Figure 2-20 and described in 2.9.14 “[Task State Register \(TSR\)](#)” on page 2-33.

The GIE bit in ITSR is physically the same bit as the PGIE bit in CSR.

The ITSР is not accessible in User mode. See [9.2.4.1](#) on page 9-2 for more information.

**Figure 2-20** Interrupt Task State Register (ITSR)



LEGEND: R = Readable by the **MVC** instruction only in Supervisor mode; W = Writable by the **MVC** instruction only in Supervisor mode; -n = value after reset.

### 2.9.9 NMI/Exception Task State Register (NTSR)

The NMI/exception task state register (NTSR) is used to store the contents of the task state register (TSR) and the conditions under which an exception occurred in the event of a nonmaskable interrupt (NMI) or an exception. The NTSR is shown in [Figure 2-21](#) and described in [Table 2-19](#). For detailed bit descriptions (except for the HWE bit), see [2.9.14 “Task State Register \(TSR\)”](#) on page 2-33. The HWE bit is set by taking a hardware exception (NMI, EXCEP, or internal) and is cleared by either **SWE** or **SWENR** instructions.

The NTSR is not accessible in User mode. See [9.2.4.1](#) on page 9-2 for more information.

**Figure 2-21** NMI/Exception Task State Register (NTSR)

31	Reserved														17	16
R-0															R/W-0	
15	14	13	11	10	9	8	7	6	5	4	3	2	1	0		
IB	SPLX	Reserved	EXC	INT	Rsvd	CXM	Reserved	XEN	GEE	SGIE	GIE					

LEGEND: R = Readable by the **MVC** instruction only in Supervisor mode; W = Writable by the **MVC** instruction only in Supervisor mode; -n = value after reset

**Table 2-19** NMI/Exception Task State Register (NTSR) Field Descriptions

Bit	Field	Description
31-17	Reserved	Reserved. Read as 0.
16	HWE	Hardware exception taken (NMI, EXCEP, or internal).
15	IB	Exception occurred while interrupts were blocked.
14	SPLX	Exception occurred during an SPLOOP.
13-11	Reserved	Reserved. Read as 0.
10	EXC	Contains EXC bit value in TSR at point exception taken.
9	INT	Contains INT bit value in TSR at point exception taken.
8	Reserved	Reserved. Read as 0.
7-6	CXM	Contains CXM bit value in TSR at point exception taken.
5-4	Reserved	Reserved. Read as 0.
3	XEN	Contains XEN bit value in TSR at point exception taken.
2	GEE	Contains GEE bit value in TSR at point exception taken.
1	SGIE	Contains SGIE bit value in TSR at point exception taken.
0	GIE	Contains GIE bit value in TSR at point exception taken.

### 2.9.10 Restricted Entry Point Register (REP)

The restricted entry point register (REP) is used by the **SWENR** instruction as the target of the change of control when an **SWENR** instruction is issued. The contents of REP should be preinitialized by the processor in Supervisor mode before any **SWENR** instruction is issued. See [9.2.4.1](#) on page 9-2 for more information. REP cannot be modified in User mode.

### 2.9.11 SPLOOP Reload Inner Loop Count Register (RILC)

Predicated SPLOOP or SPLOOPD instructions used in conjunction with a SPMASKR or SPKERNELR instruction use the SPLOOP reload inner loop count register (RILC), Figure 2-22, as the iteration count value to be written to the SPLOOP inner loop count register (ILC) in the cycle before the reload operation begins. See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more information.

**Figure 2-22** Reload Inner Loop Count Register (RILC)

31	32-bit inner loop count reload	0
R/W-0		

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

### 2.9.12 Saturation Status Register (SSR)

The saturation status register (SSR) provides saturation flags for each functional unit, making it possible for the program to distinguish between saturations caused by different instructions in the same execute packet. There is no direct connection to the SAT bit in the control status register (CSR); writes to the SAT bit have no effect on SSR and writes to SSR have no effect on the SAT bit. Care must be taken when restoring SSR and the SAT bit when returning from a context switch. Since the SAT bit cannot be written to a value of 1 using the **MVC** instruction, restoring the SAT bit to a 1 must be done by executing an instruction that results in saturation. The saturating instruction would affect SSR; therefore, SSR must be restored after the SAT bit has been restored. The SSR is shown in Figure 2-23 and described in Table 2-20.

Instructions resulting in saturation set the appropriate unit flag in SSR in the cycle following the writing of the result to the register file. The setting of the flag from a functional unit takes precedence over a write to the bit from an **MVC** instruction. If no functional unit saturation has occurred, the flags may be set to 0 or 1 by the **MVC** instruction, unlike the SAT bit in CSR.

The bits in SSR can be set by the **MVC** instruction or by a saturation in the associated functional unit. The bits are cleared only by a reset or by the **MVC** instruction. The bits are not cleared by the occurrence of a nonsaturating instruction.

**Figure 2-23** Saturation Status Register (SSR)

31	5	4	3	2	1	0
Reserved	M2	M1	S2	S1	L2	L1
R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-20** Saturation Status Register Field Descriptions (Part 1 of 2)

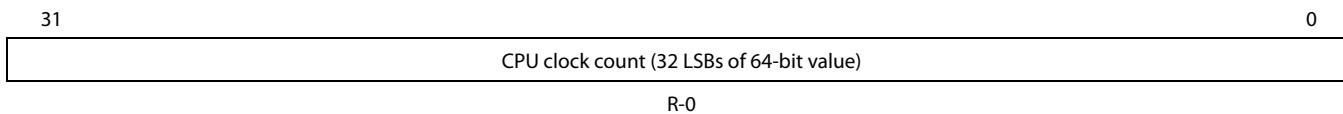
Bit	Field	Value	Description
31-6	Reserved	0	Reserved. Read as 0.
5	M2	0	M2 unit. Saturation did not occur on M2 unit.
		1	Saturation occurred on M2 unit.

**Table 2-20 Saturation Status Register Field Descriptions (Part 2 of 2)**

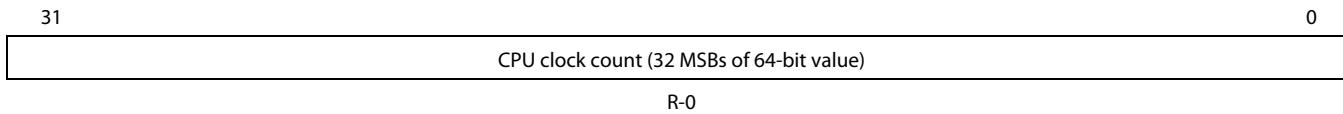
Bit	Field	Value	Description
4	M1	0	M1 unit.
		1	Saturation did not occur on M1 unit.
		1	Saturation occurred on M1 unit.
3	S2	0	S2 unit.
		1	Saturation did not occur on S2 unit.
		1	Saturation occurred on S2 unit.
2	S1	0	S1 unit.
		1	Saturation did not occur on S1 unit.
		1	Saturation occurred on S1 unit.
1	L2	0	L2 unit.
		1	Saturation did not occur on L2 unit.
		1	Saturation occurred on L2 unit.
0	L1	0	L1 unit.
		1	Saturation did not occur on L1 unit.
		1	Saturation occurred on L1 unit.

### 2.9.13 Time Stamp Counter Registers (TSCL and TSCH)

The CPU contains a free running 64-bit counter that advances each CPU clock under normal operation. The counter is accessed as two 32-bit read-only control registers, TSCL (Figure 2-24) and TSCH (Figure 2-25).

**Figure 2-24 Time Stamp Counter Register - Low Half (TSCL)**

LEGEND: R = Readable by the **MVC** instruction; -n = value after reset

**Figure 2-25 Time Stamp Counter Register - High Half (TSCH)**

LEGEND: R = Readable by the **MVC** instruction; -n = value after reset

#### 2.9.13.1 Initialization

The counter is cleared to 0 after reset, and counting is disabled.

#### 2.9.13.2 Enabling Counting

The counter is enabled by writing to TSCL. The value written is ignored. Counting begins in the cycle after the **MVC** instruction executes. If executed with the count disabled, the following code sequence shows the timing of the count starting (assuming no stalls occur in the three cycles shown).

```

MVC      A0, TSCL      ; Start TSC
MVC      TSCL,A0      ; A0 = 0
MVC      TSCL,A1      ; A1 = 1
  
```

### 2.9.13.3 Disabling Counting

Once enabled, counting cannot be disabled under program control. Counting is disabled in the following cases:

- After exiting the reset state.
- When the CPU is fully powered down.

### 2.9.13.4 Reading the Counter

Reading the full 64-bit count takes two sequential **MVC** instructions. A read from **TSCL** causes the upper 32 bits of the count to be copied into **TSCH**. In normal operation, only this snapshot of the upper half of the 64-bit count is available to the programmer. The value read will always be the value copied at the cycle of the last **MVC TSCL**, **reg** instruction. If it is read with no **TSCL** reads having taken place since reset, then the reset value of 0 is read.

When reading the full 64-bit value, it must be ensured that no interrupts are serviced between the two **MVC** instructions if an ISR is allowed to make use of the time stamp counter. There is no way for an ISR to restore the previous value of **TSCH** (snapshot) if it reads **TSCL**, since a new snapshot is performed.

Two methods for reading the 64-bit count value in an uninterruptible manner are shown in [Example 2-1](#) and [Example 2-2](#). [Example 2-1](#) uses the fact that interrupts are automatically disabled in the delay slots of a branch to prevent an interrupt from happening between the **TSCL** read and the **TSCH** read. [Example 2-2](#) accomplishes the same task by explicitly disabling interrupts.

#### Example 2-1 Code to Read the 64-Bit TSC Value in Branch Delay Slot

```

BNOP      TSC_Read_Done, 3
MVC      TSCL,A0      ; Read the low half first; high half copied to TSCH
MVC      TSCH,A1      ; Read the snapshot of the high half
TSC_Read_Done:
  
```

**End of Example 2-1**

#### Example 2-2 Code to Read the 64-Bit TSC Value Using DINT/RINT

```

DINT
|| MVC      TSCL,A0      ; Read the low half first; high half copied to TSCH
RINT
|| MVC      TSCH,A1      ; Read the snapshot of the high half
TSC_Read_Done:
  
```

**End of Example 2-2**

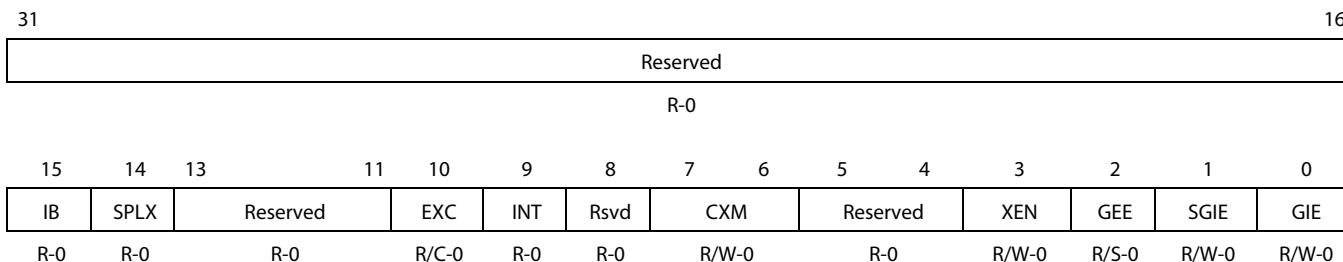
### 2.9.14 Task State Register (TSR)

The task state register (TSR) contains all of the status bits that determine or indicate the current execution environment. TSR is saved in the event of an interrupt or exception to the ITSR or NTSR, respectively. All bits are readable by the **MVC** instruction. The TSR is shown in [Figure 2-26](#) and described in [Table 2-21](#). The SGIE bit in TSR is used by the **DINT** and **RINT** instructions to globally disable and reenable interrupts.

The GIE and SGIE bits may be written in both User mode and Supervisor mode. The remaining bits all have restrictions on how they are written. See 9.2.4.2 “[Partially Restricted Control Register Access in User Mode](#)” on page 9-3 for more information.

The GIE bit in TSR is physically the same bit as the GIE bit in CSR. It is retained in CSR for compatibility reasons, but placed in TSR so that it will be copied in the event of either an exception or an interrupt.

**Figure 2-26 Task State Register (TSR)**



LEGEND: R = Readable by the **MVC** instruction; W = Writable in Supervisor mode; C = Clearable in Supervisor mode; S = Can be set in Supervisor mode;  
 $-n$  = value after reset

**Table 2-21 Task State Register (TSR) Field Descriptions (Part 1 of 2)**

Bit	Field	Value	Description
31-16	Reserved	0	Reserved. Read as 0.
15	IB	0	Interrupts blocked. Not writable by the <b>MVC</b> instruction; set only by hardware.
		1	Interrupts not blocked in previous cycle (interruptible point).
			Interrupts were blocked in previous cycle.
14	SPLX	0	SPLOOP executing. Not writable by the <b>MVC</b> instruction; set only by hardware.
		1	Not currently executing SPLOOP
			Currently executing SPLOOP
13-11	Reserved	0	Reserved. Read as 0.
10	EXC	0	Exception processing. Clearable by the <b>MVC</b> instruction in Supervisor mode. Not clearable by the <b>MVC</b> instruction in User mode.
		1	Not currently processing an exception.
			Currently processing an exception.
9	INT	0	Interrupt processing. Not writable by the <b>MVC</b> instruction.
		1	Not currently processing an interrupt.
			Currently processing an interrupt.
8	Reserved	0	Reserved. Read as 0.

**Table 2-21 Task State Register (TSR) Field Descriptions (Part 2 of 2)**

Bit	Field	Value	Description
7-6	CXM	0-3h	Current execution mode. Not writable by the <b>MVC</b> instruction; these bits reflect the current execution mode of the execute pipeline. CXM is set to 1 when you begin executing the first instruction in User mode. See <a href="#">Chapter 7</a> on page 7-1 for more information.
		0	Supervisor mode
		1h	User mode
		2h-3h	Reserved (an attempt to set these values is ignored)
5-4	Reserved	0	Reserved. Read as 0.
3	XEN	0	Maskable exception enable. Writable only in Supervisor mode.
		0	Disables all maskable exceptions.
		1	Enables all maskable exceptions.
2	GEE	0	Global exception enable. Can be set to 1 only in Supervisor mode. Once set, cannot be cleared except by reset.
		0	Disables all exceptions except the reset interrupt.
		1	Enables all exceptions.
1	SGIE	0	Saved global interrupt enable. Contains previous state of GIE bit after execution of a <b>DINT</b> instruction. Writable in Supervisor and User mode.
		0	Global interrupts remain disabled by the <b>RINT</b> instruction.
		1	Global interrupts are enabled by the <b>RINT</b> instruction.
0	GIE	0	Global interrupt enable. Same physical bit as the GIE bit in the control status register (CSR). Writable in Supervisor and User mode. See <a href="#">6.2</a> on page 6-10 for details on how the GIE bit affects interruptibility.
		0	Disables all interrupts except the reset interrupt and NMI (nonmaskable interrupt).
		1	Enables all interrupts.

## 2.10 Control Register File Extensions for Floating-Point Operations

The C66x DSP has three configuration registers to support floating-point operations. The registers specify the desired floating-point rounding mode for the .L, .S and .M units. They also contain fields to warn if src1 and src2 are NaN or denormalized numbers, and if the result overflows, underflows, is inexact, infinite, or invalid. There are also fields to warn if a divide by 0 was performed, or if a compare was attempted with a NaN source.

The OVER, UNDER, INEX, INVAL, DENN, NANN, INFO, UNORD and DIV0 bits within these registers will not be modified by a conditional instruction whose condition is false.

[Table 2-22](#) lists the additional registers used.

**Table 2-22 Control Register File Extensions for Floating-Point Operations**

Acronym	Register Name	Section
FADCR	Floating-point adder configuration register	<a href="#">2.10.1 "Floating-Point Adder Configuration Register (FADCR)" on page 2-35</a>
FAUCR	Floating-point auxiliary configuration register	<a href="#">2.10.2 "Floating-Point Auxiliary Configuration Register (FAUCR)" on page 2-37</a>
FMCR	Floating-point multiplier configuration register	<a href="#">2.10.3 "Floating-Point Multiplier Configuration Register (FMCR)" on page 2-40</a>

FMCR specifies the desired floating-point rounding mode and contains the warning bits for the instructions that use the .M units.

FAUCR specifies the desired floating-point rounding mode and contains the warning bits for the instructions that use the .S units.

FADCR specifies the desired floating-point rounding mode and contains the warning bits for the instructions that use the .L units and for instructions that can be executed on both .L and .S units. The warning bits in FADCR are the logical-OR of the warnings produced on the .L functional unit and the warnings produced by the instructions that can be executed on both .L and .S unit.

Therefore the following instructions executing in the .S functional unit use the rounding mode from and set the warning bits in FADCR (and not in FAUCR as other .S unit instructions do):

- ADDSP / SUBDP
- ADDDP / SUBDP
- FADDSP / FSUBSP
- FADDDP / FSUBDP
- DINTHSP
- DINTHSPU
- DSPINTH
- DSPINT
- INTSPU
- INTSP
- SPINT

### 2.10.1 Floating-Point Adder Configuration Register (FADCR)

The floating-point adder configuration register (FADCR) contains fields that specify underflow or overflow, the rounding mode, NaNs, denormalized numbers, and inexact results for instructions that use the .L functional units. FADCR has a set of fields specific to each of the .L units: .L2 uses bits 31-16 and .L1 uses bits 15-0. FADCR is shown in [Figure 2-27](#) and described in [Table 2-23](#).

**Figure 2-27 Floating-Point Adder Configuration Register (FADCR)**

31	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	RMODE	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1		
R-0	R/W-0											
15	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	RMODE	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1		
R-0	R/W-0											

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-23 Floating-Point Adder Configuration Register (FADCR) Field Descriptions (Part 1 of 3)**

Bit	Field	Value	Description
31-27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.

**Table 2-23 Floating-Point Adder Configuration Register (FADCR) Field Descriptions (Part 2 of 3)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
26-25	RMODE	0-3h 0 1h 2h 3h	Rounding mode select for .L2.  0 Round toward nearest representable floating-point number 1h Round toward 0 (truncate) 2h Round toward infinity (round up) 3h Round toward negative infinity (round down)
24	UNDER	0 1	Result underflow status for .L2.  0 Result does not underflow. 1 Result underflows.
23	INEX	0 1	Inexact results status for .L2.  0 Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL. 1
22	OVER	0 1	Result overflow status for .L2.  0 Result does not overflow. 1 Result overflows.
21	INFO	0 1	Signed infinity for .L2.  0 Result is not signed infinity. 1 Result is signed infinity.
20	INVAL	0 1	A signed NaN (SNaN) is not a source.  1 A signed NaN (SNaN) is a source. NaN is a source in a floating-point to integer conversion or when infinity is subtracted from infinity.
19	DEN2	0 1	Denormalized number select for .L2 src2.  0 src2 is not a denormalized number. 1 src2 is a denormalized number.
18	DEN1	0 1	Denormalized number select for .L2 src1.  0 src1 is not a denormalized number. 1 src1 is a denormalized number.
17	NAN2	0 1	NaN select for .L2 src2.  0 src2 is not NaN. 1 src2 is NaN.
16	NAN1	0 1	NaN select for .L2 src1.  0 src1 is not NaN. 1 src1 is NaN.
15-11	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10-9	RMODE	0-3h 0 1h 2h 3h	Rounding mode select for .L1.  0 Round toward nearest representable floating-point number 1h Round toward 0 (truncate) 2h Round toward infinity (round up) 3h Round toward negative infinity (round down)
8	UNDER	0 1	Result underflow status for .L1.  0 Result does not underflow. 1 Result underflows.

**Table 2-23 Floating-Point Adder Configuration Register (FADCR) Field Descriptions (Part 3 of 3)**

Bit	Field	Value	Description
7	INEX	0	Inexact results status for .L1.
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
6	OVER	0	Result overflow status for .L1.
		1	Result does not overflow. Result overflows.
5	INFO	0	Signed infinity for .L1.
		1	Result is not signed infinity. Result is signed infinity.
4	INVAL	0	A signed NaN (SNaN) is not a source.
		1	A signed NaN (SNaN) is a source. NaN is a source in a floating-point to integer conversion or when infinity is subtracted from infinity.
3	DEN2	0	Denormalized number select for .L1 src2. src2 is not a denormalized number.
		1	src2 is a denormalized number.
2	DEN1	0	Denormalized number select for .L1 src1. src1 is not a denormalized number.
		1	src1 is a denormalized number.
1	NAN2	0	Nan select for .L1 src2. src2 is not NaN.
		1	src2 is NaN.
0	NAN1	0	Nan select for .L1 src1. src1 is not NaN.
		1	src1 is NaN.

## 2.10.2 Floating-Point Auxiliary Configuration Register (FAUCR)

The floating-point auxiliary register (FAUCR) contains fields that specify underflow or overflow, the rounding mode, NaNs, denormalized numbers, and inexact results for instructions that use the .S functional units. FAUCR has a set of fields specific to each of the .S units: .S2 uses bits 31-16 and .S1 uses bits 15-0. FAUCR is shown in [Figure 2-28](#) and described in [Table 2-24](#).

**Figure 2-28 Floating-Point Auxiliary Configuration Register (FAUCR)**

31	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	DIV0	UNORD	UND	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1	
R-0	R/W-0											
15	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	DIV0	UNORD	UND	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1	
R-0	R/W-0											

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-24 Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions (Part 1 of 2)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
31-27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
26	DIV0	0	Source to reciprocal operation for .S2. 0 is not source to reciprocal operation.
			0 is source to reciprocal operation.
25	UNORD	0	Source to a compare operation for .S2 NaN is not a source to a compare operation.
			NaN is a source to a compare operation.
24	UND	0	Result underflow status for .S2. Result does not underflow.
			Result underflows.
23	INEX	0	Inexact results status for .S2.
			Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
22	OVER	0	Result overflow status for .S2. Result does not overflow.
			Result overflows.
21	INFO	0	Signed infinity for .S2. Result is not signed infinity.
			Result is signed infinity.
20	INVAL	0	A signed NaN (SNaN) is not a source.
			A signed NaN (SNaN) is a source. NaN is a source in a floating-point to integer conversion or when infinity is subtracted from infinity.
19	DEN2	0	Denormalized number select for .S2 src2. src2 is not a denormalized number.
			src2 is a denormalized number.
18	DEN1	0	Denormalized number select for .S2 src1. src1 is not a denormalized number.
			src1 is a denormalized number.
17	NAN2	0	NaN select for .S2 src2. src2 is not NaN.
			src2 is NaN.
16	NAN1	0	NaN select for .S2 src1. src1 is not NaN.
			src1 is NaN.
15-11	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10	DIV0	0	Source to reciprocal operation for .S1. 0 is not source to reciprocal operation.
			0 is source to reciprocal operation.
9	UNORD	0	Source to a compare operation for .S1 NaN is not a source to a compare operation.
			NaN is a source to a compare operation.

**Table 2-24 Floating-Point Auxiliary Configuration Register (FAUCR) Field Descriptions (Part 2 of 2)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
8	UND	0	Result underflow status for .S1. Result does not underflow.
		1	Result underflows.
7	INEX	0	Inexact results status for .S1.
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
6	OVER	0	Result overflow status for .S1. Result does not overflow.
		1	Result overflows.
5	INFO	0	Signed infinity for .S1. Result is not signed infinity.
		1	Result is signed infinity.
4	INVAL	0	A signed NaN (SNaN) is not a source.
		1	A signed NaN (SNaN) is a source. NaN is a source in a floating-point to integer conversion or when infinity is subtracted from infinity.
3	DEN2	0	Denormalized number select for .S1 src2. src2 is not a denormalized number.
		1	src2 is a denormalized number.
2	DEN1	0	Denormalized number select for .S1 src1. src1 is not a denormalized number.
		1	src1 is a denormalized number.
1	NAN2	0	NaN select for .S1 src2. src2 is not NaN.
		1	src2 is NaN.
0	NAN1	0	NaN select for .S1 src1. src1 is not NaN.
		1	src1 is NaN.

### 2.10.3 Floating-Point Multiplier Configuration Register (FMCR)

The floating-point multiplier configuration register (FMCR) contains fields that specify underflow or overflow, the rounding mode, NaNs, denormalized numbers, and inexact results for instructions that use the .M functional units. FMCR has a set of fields specific to each of the .M units: .M2 uses bits 31-16 and .M1 uses bits 15-0. FMCR is shown in [Figure 2-29](#) and described in [Table 2-25](#).

**Figure 2-29 Floating-Point Multiplier Configuration Register (FMCR)**

31	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	RMODE	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1		
R-0	R/W-0											
15	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	RMODE	UNDER	INEX	OVER	INFO	INVAL	DEN2	DEN1	NAN2	NAN1		
R-0	R/W-0											

LEGEND: R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; -n = value after reset

**Table 2-25 Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions (Part 1 of 2)**

Bit	Field	Value	Description
31-27	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
26-25	RMODE	0-3h	Rounding mode select for .M2.
		0	Round toward nearest representable floating-point number
		1h	Round toward 0 (truncate)
		2h	Round toward infinity (round up)
		3h	Round toward negative infinity (round down)
24	UNDER	0	Result underflow status for .M2.
		1	Result does not underflow.
			Result underflows.
23	INEX	0	Inexact results status for .M2.
		1	Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL.
22	OVER	0	Result overflow status for .M2.
		1	Result does not overflow.
			Result overflows.
21	INFO	0	Signed infinity for .M2.
		1	Result is not signed infinity.
			Result is signed infinity.
20	INVAL	0	A signed NaN (SNaN) is not a source.
		1	A signed NaN (SNaN) is a source. NaN is a source in a floating-point to integer conversion or when infinity is subtracted from infinity.
19	DEN2	0	Denormalized number select for .M2 src2.
		1	src2 is not a denormalized number.
			src2 is a denormalized number.
18	DEN1	0	Denormalized number select for .M2 src1.
		1	src1 is not a denormalized number.
			src1 is a denormalized number.

**Table 2-25 Floating-Point Multiplier Configuration Register (FMCR) Field Descriptions (Part 2 of 2)**

Bit	Field	Value	Description
17	NAN2	0 1	NaN select for .M2 src2. <i>src2</i> is not NaN. <i>src2</i> is NaN.
16	NAN1	0 1	NaN select for .M2 src1. <i>src1</i> is not NaN. <i>src1</i> is NaN.
15-11	Reserved	0	Reserved. The reserved bit location is always read as 0. A value written to this field has no effect.
10-9	RMODE	0-3h 0 1h 2h 3h	Rounding mode select for .M1. 0 Round toward nearest representable floating-point number 1h Round toward 0 (truncate) 2h Round toward infinity (round up) 3h Round toward negative infinity (round down)
8	UNDER	0 1	Result underflow status for .M1. 0 Result does not underflow. 1 Result underflows.
7	INEX	0 1	Inexact results status for .M1. 0 Result differs from what would have been computed had the exponent range and precision been unbounded; never set with INVAL. 1
6	OVER	0 1	Result overflow status for .M1. 0 Result does not overflow. 1 Result overflows.
5	INFO	0 1	Signed infinity for .M1. 0 Result is not signed infinity. 1 Result is signed infinity.
4	INVAL	0 1	A signed NaN (SNaN) is not a source. A signed NaN (SNaN) is a source. NaN is a source in a floating-point to integer conversion or when infinity is subtracted from infinity.
3	DEN2	0 1	Denormalized number select for .M1 src2. 0 <i>src2</i> is not a denormalized number. 1 <i>src2</i> is a denormalized number.
2	DEN1	0 1	Denormalized number select for .M1 src1. 0 <i>src1</i> is not a denormalized number. 1 <i>src1</i> is a denormalized number.
1	NAN2	0 1	NaN select for .M1 src2. 0 <i>src2</i> is not NaN. 1 <i>src2</i> is NaN.
0	NAN1	0 1	NaN select for .M1 src1. 0 <i>src1</i> is not NaN. 1 <i>src1</i> is NaN.



# Instruction Set

This chapter describes the assembly language instructions of the TMS320C66x DSP. Also described are parallel operations, conditional operations, resource constraints, and addressing modes.

The C66x DSP uses all of the instructions available to the TMS320C62x, TMS320C64x, TMS320C64x+, and TMS320C674x+ DSPs. The C664x DSP instructions include 8-bit and 16-bit extensions, nonaligned word loads and stores, data packing/unpacking operations.

- 3.1 "Instruction Operation and Execution Notation" on page 3-2
- 3.2 "Instruction Syntax and Opcode Notations" on page 3-4
- 3.3 "Overview of IEEE Standard Single- and Double-Precision Formats" on page 3-6
- 3.4 "Delay Slots" on page 3-9
- 3.5 "Parallel Operations" on page 3-11
- 3.6 "Conditional Operations" on page 3-14
- 3.7 "SPMASKed Operations" on page 3-15
- 3.8 "Resource Constraints" on page 3-15
- 3.9 "Addressing Modes" on page 3-24
- 3.10 "Compact Instructions on the CPU" on page 3-29
- 3.11 "Instruction Compatibility" on page 3-35

### 3.1 Instruction Operation and Execution Notation

Table 3-1 explains the symbols used in the instruction descriptions.

**Table 3-1 Instruction Operation and Execution Notations (Part 1 of 3)**

Symbol	Meaning
abs(x)	Absolute value of x
and	Bitwise AND
-a	Perform 2s-complement subtraction using the addressing mode defined by the AMR
+a	Perform 2s-complement addition using the addressing mode defined by the AMR
b <sub>i</sub>	Select bit i of source/destination b
bit_count	Count the number of bits that are 1 in a specified byte
bit_reverse	Reverse the order of bits in a 32-bit register
byte0	8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
byte1	8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
byte2	8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
byte3	8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
bv2	Bit vector of two flags for s2 or u2 data type
bv4	Bit vector of four flags for s4 or u4 data type
b <sub>y..z</sub>	Selection of bits y through z of bit string b
cond	Check for either creg equal to 0 or creg not equal to 0
creg	3-bit field specifying a conditional register, see Section 3.6 on page 3-14
cstn	n-bit constant field (for example, cst5)
dint	64-bit integer value (two registers)
dst_e	lsb32 of 64-bit dst (placed in even-numbered register of a 64-bit register pair)
dst_h	msb8 of 40-bit dst (placed in odd-numbered register of 64-bit register pair)
dst_l	lsb32 of 40-bit dst (placed in even-numbered register of a 64-bit register pair)
dst_o	msb32 of 64-bit dst (placed in odd-numbered register of 64-bit register pair)
dst_0 or src_0	32-bit value in the least-significant position in 128-bit quad register
dst_1 or src_1	32-bit value in the next to least-significant 32-bit word position in 128-bit quad register
dst_2 or src_2	32-bit value in the next to most-significant 32-bit word position in 128-bit quad register
dst_3 or src_3	32-bit value in the least-significant position in 128-bit quad register
dwdst	64-bit register pair result
dwop	64-bit register pair operand
dws4	Four packed signed 16-bit integers in a 64-bit register pair
dwu4	Four packed unsigned 16-bit integers in a 64-bit register pair
gmpy	Galois Field Multiply
i2	Two packed 16-bit integers in a single 32-bit register
i4	Four packed 8-bit integers in a single 32-bit register
int	32-bit integer value
lmb0(x)	Leftmost 0 bit search of x
lmb1(x)	Leftmost 1 bit search of x
long	40-bit integer value
lsbn or LSBn	n least-significant bits (for example, lsb16)
msbn or MSBn	n most-significant bits (for example, msb16)
nop	No operation
norm(x)	Leftmost nonredundant sign bit of x

**Table 3-1 Instruction Operation and Execution Notations (Part 2 of 3)**

<b>Symbol</b>	<b>Meaning</b>
not	Bitwise logical complement
op	Opfields
or	Bitwise OR
R	Any general-purpose register
ROTL	Rotate left
sat	Saturate
sbyte0	Signed 8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
sbyte1	Signed 8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
sbyte2	Signed 8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
sbyte3	Signed 8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
scstn	n-bit signed constant field
se	Sign-extend
sint	Signed 32-bit integer value
slong	Signed 40-bit integer value
sllong	Signed 64-bit integer value
slsb16	Signed 16-bit integer value in lower half of 32-bit register
smsb16	Signed 16-bit integer value in upper half of 32-bit register
src1_e or src2_e	lsb32 of 64-bit src (placed in even-numbered register of a 64-bit register pair)
src1_h or src2_h	msb8 of 40-bit src (placed in odd-numbered register of 64-bit register pair)
src1_l or src2_l	lsb32 of 40-bit src (placed in even-numbered register of a 64-bit register pair)
src1_o or src2_o	msb32 of 64-bit src (placed in odd-numbered register of 64-bit register pair)
s2	Two packed signed 16-bit integers in a single 32-bit register
s4	Four packed signed 8-bit integers in a single 32-bit register
-s	Perform 2s-complement subtraction and saturate the result to the result size, if an overflow occurs
+s	Perform 2s-complement addition and saturate the result to the result size, if an overflow occurs
ubyte0	Unsigned 8-bit value in the least-significant byte position in 32-bit register (bits 0-7)
ubyte1	Unsigned 8-bit value in the next to least-significant byte position in 32-bit register (bits 8-15)
ubyte2	Unsigned 8-bit value in the next to most-significant byte position in 32-bit register (bits 16-23)
ubyte3	Unsigned 8-bit value in the most-significant byte position in 32-bit register (bits 24-31)
ucstn	n-bit unsigned constant field (for example, ucst5)
uint	Unsigned 32-bit integer value
ulong	Unsigned 40-bit integer value
ullong	Unsigned 64-bit integer value
ulsb16	Unsigned 16-bit integer value in lower half of 32-bit register
umsb16	Unsigned 16-bit integer value in upper half of 32-bit register
u2	Two packed unsigned 16-bit integers in a single 32-bit register
u4	Four packed unsigned 8-bit integers in a single 32-bit register
x clear b,e	Clear a field in x, specified by b (beginning bit) and e (ending bit)
x ext l,r	Extract and sign-extend a field in x, specified by l (shift left value) and r (shift right value)
x extu l,r	Extract an unsigned field in x, specified by l (shift left value) and r (shift right value)
x set b,e	Set field in x to all 1s, specified by b (beginning bit) and e (ending bit)
xdwop	64-bit register pair operand that can optionally use cross path
xint	32-bit integer value that can optionally use cross path
xor	Bitwise exclusive-ORs

## 3.2 Instruction Syntax and Opcode Notations

## Chapter 3—Instruction Set

**Table 3-1 Instruction Operation and Execution Notations (Part 3 of 3)**

<b>Symbol</b>	<b>Meaning</b>
xsint	Signed 32-bit integer value that can optionally use cross path
xslsb16	Signed 16 LSB of register that can optionally use cross path
xsmsb16	Signed 16 MSB of register that can optionally use cross path
xs2	Two packed signed 16-bit integers in a single 32-bit register that can optionally use cross path
xs4	Four packed signed 8-bit integers in a single 32-bit register that can optionally use cross path
xuint	Unsigned 32-bit integer value that can optionally use cross path
xulsb16	Unsigned 16 LSB of register that can optionally use cross path
xumsb16	Unsigned 16 MSB of register that can optionally use cross path
xu2	Two packed unsigned 16-bit integers in a single 32-bit register that can optionally use cross path
xu4	Four packed unsigned 8-bit integers in a single 32-bit register that can optionally use cross path
→	Assignment
+	Addition
++	Increment by 1
×	Multiplication
-	Subtraction
==	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<<	Shift left
>>	Shift right
>>s	Shift right with sign extension
>>z	Shift right with a zero fill
~	Logical inverse
&	Logical AND

**3.2 Instruction Syntax and Opcode Notations**

Table 3-2 explains the syntaxes and opcode fields used in the instruction descriptions.

**Table 3-2 Instruction Syntax and Opcode Notations (Part 1 of 2)**

<b>Symbol</b>	<b>Meaning</b>
baseR	base address register
creg	3-bit field specifying a conditional register, see Section 3.6 on page 3-14
cst	constant
csta	constant a
cstb	constant b
cstn	n-bit constant field
dst	destination
dw	doubleword; 0 = word, 1 = doubleword
fcyc	SPLOOP fetch cycle
fstg	SPLOOP fetch stage
h	MVK or MVKH instruction

**Table 3-2** Instruction Syntax and Opcode Notations (Part 2 of 2)

Symbol	Meaning
$ii_n$	bit n of the constant $ii$
$ld/st$	load or store; 0 = store, 1 = load
<i>mode</i>	addressing mode, see Section 3.9 on page 3-24
<i>na</i>	nonaligned; 0 = aligned, 1 = nonaligned
<i>N3</i>	3-bit field
<i>offsetR</i>	register offset
<i>op</i>	opfield; field within opcode that specifies a unique instruction
$op_n$	bit n of the opfield
<i>p</i>	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
<i>ptr</i>	offset from either A4-A7 or B4-B7 depending on the value of the s bit. The <i>ptr</i> field is the 2 least-significant bits of the <i>src2</i> ( <i>baseR</i> ) field—bit 2 of register address is forced to 1.
<i>r</i>	LDDW/LDNDW/LDNW instruction
<i>rsv</i>	reserved
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B.
<i>sc</i>	scaling mode; 0 = nonscaled, <i>offsetR/ucst5</i> is not shifted; 1 = scaled, <i>offsetR/ucst5</i> is shifted
<i>scstn</i>	n-bit signed constant field
$scst_n$	bit n of the signed constant field
<i>sn</i>	sign
<i>src</i>	source
<i>src1</i>	source 1
<i>src2</i>	source 2
$stg_n$	bit n of the constant $stg$
<i>sz</i>	data size select; 0 = primary size, 1 = secondary size (see “ <a href="#">Expansion Field in Compact Header Word</a> ” on page 3-31)
<i>t</i>	side of source/destination ( <i>src/dst</i> ) register; 0 = side A, 1 = side B
<i>ucstn</i>	n-bit unsigned constant field
$ucst_n$	bit n of the unsigned constant field
<i>unit</i>	unit decode
<i>x</i>	cross path for <i>src2</i> ; 0 = do not use cross path, 1 = use cross path
<i>y</i>	.D1 or .D2 unit; 0 = .D1 unit, 1 = .D2 unit
<i>z</i>	test for equality with zero or nonzero

### 3.2.1 32-Bit Opcode Maps

The 32-bit opcodes are mapped in Appendix C “[.D Unit Instructions and Opcode Maps](#)” on page C-1 through Appendix H “[No Unit Specified Instructions and Opcode Maps](#)” on page H-1.

### 3.2.2 16-Bit Opcode Maps

The 16-bit opcodes used for compact instructions are mapped in Appendix C “[.D Unit Instructions and Opcode Maps](#)” on page C-1 through Appendix H “[No Unit Specified Instructions and Opcode Maps](#)” on page H-1. See Section 3.10 on page 3-29 for more information about compact instructions.

### 3.3 Overview of IEEE Standard Single- and Double-Precision Formats

Floating-point operands are classified as single-precision (SP) and double-precision (DP). Single-precision floating-point values are 32-bit values stored in a single register. Double-precision floating-point values are 64-bit values stored in a register pair. The register pair consists of consecutive even and odd registers from the same register file. The 32 least-significant-bits are loaded into the even register; the 32 most-significant-bits containing the sign bit and exponent are loaded into the next register (that is always the odd register). The register pair syntax places the odd register first, followed by a colon, then the even register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

Instructions that use DP sources fall in two categories: instructions that read the upper and lower 32-bit words on separate cycles, and instructions that read both 32-bit words on the same cycle. All instructions that produce a double-precision result write the low 32-bit word one cycle before writing the high 32-bit word. If an instruction that writes a DP result is followed by an instruction that uses the result as its DP source and it reads the upper and lower words on separate cycles, then the second instruction can be scheduled on the same cycle that the high 32-bit word of the result is written. The lower result is written on the previous cycle. This is because the second instruction reads the low word of the DP source one cycle before the high word of the DP source.

IEEE floating-point numbers consist of normal numbers, denormalized numbers, NaNs (not a number), and infinity numbers. Denormalized numbers are nonzero numbers that are smaller than the smallest nonzero normal number. Infinity is a value that represents an infinite floating-point number. NaN values represent results for invalid operations, such as (+infinity + (-infinity)).

Normal single-precision values are always accurate to at least six decimal places, sometimes up to nine decimal places. Normal double-precision values are always accurate to at least 15 decimal places, sometimes up to 17 decimal places.

[Table 3-3](#) shows notations used in discussing floating-point numbers.

**Table 3-3 IEEE Floating-Point Notations**

Symbol	Meaning
s	Sign bit
e	Exponent field
f	Fraction (mantissa) field
x	Can have value of 0 or 1 (don't care)
NaN	Not-a-Number (SNaN or QNaN)
SNaN	Signal NaN
QNaN	Quiet NaN
NaN_out	QNaN with all bits in the f field = 1
Inf	Infinity
LFPN	Largest floating-point number
SFPN	Smallest floating-point number
LDFPN	Largest denormalized floating-point number
SDFPN	Smallest denormalized floating-point number
signed Inf	+infinity or -infinity
signed NaN_out	NaN_out with s = 0 or 1

### 3.3.1 Single-Precision Formats

Figure 3-1 shows the fields of a single-precision floating-point number represented within a 32-bit register.

**Figure 3-1 Single-Precision Floating-Point Fields**

31	30	23	22	0
s	e			f

LEGEND: s = sign bit (0 = positive, 1 = negative); e = 8-bit exponent (0 < e < 255); f = 23-bit fraction (0 < f <  $1 \times 2^{-1} + 1 \times 2^{-2} + \dots + 1 \times 2^{-23}$  or  $0 < f < ((2^{23}) - 1)/(2^{23})$ )

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 255) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a single-precision floating-point number.

$$\text{Normalized: } -1^s \times 2^{(e - 127)} \times 1.f \quad 0 < e < 255$$

$$\text{Denormalized (Subnormal): } -1^s \times 2^{-126} \times 0.f \quad e = 0; f \text{ is nonzero}$$

Table 3-4 shows the s, e, and f values for special single-precision floating-point numbers.

**Table 3-4 Special Single-Precision Values**

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
-0	1	0	0
+Inf	0	255	0
-Inf	1	255	0
NaN	x	255	nonzero
QNaN	x	255	1xx..x
SNaN	x	255	0xx..x and nonzero

Table 3-5 shows hexadecimal and decimal values for some single-precision floating-point numbers.

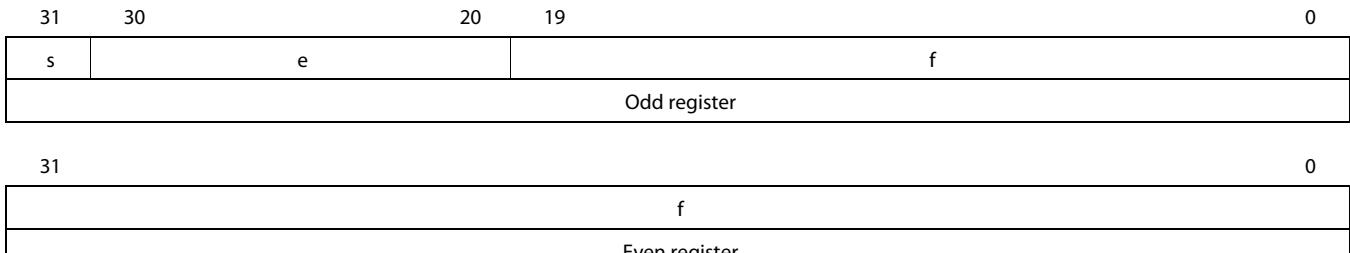
**Table 3-5 Hexadecimal and Decimal Representation for Selected Single-Precision Values**

Symbol	Hex Value	Decimal Value
NaN_out	7FFFFFFF	QNaN
0	00000000	0.0
-0	80000000	-0.0
1	3F800000	1.0
2	40000000	2.0
LFPN	7F7FFFFF	3.40282347e+38
SFPN	00800000	1.17549435e-38
LDFPN	007FFFFF	1.17549421e-38
SDFPN	00000001	1.40129846e-45

### 3.3.2 Double-Precision Formats

[Figure 3-2](#) shows the fields of a double-precision floating-point number represented within a pair of 32-bit registers.

**Figure 3-2 Double-Precision Floating-Point Fields**



LEGEND: s = sign bit (0 = positive, 1 = negative); e = 11-bit exponent ( $0 < e < 2047$ ); f = 52-bit fraction ( $0 < f < 1 \times 2^1 + 1 \times 2^{-2} + \dots + 1 \times 2^{-52}$  or  $0 < f < ((2^{52}) - 1)/(2^{52})$ )

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 2047) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a double-precision floating-point number.

$$\text{Normalized: } -1^s \times 2^{(e - 1023)} \times 1.f \quad 0 < e < 2047$$

$$\text{Denormalized (Subnormal): } -1^s \times 2^{-1022} \times 0.f \quad e = 0; f \text{ is nonzero}$$

[Table 3-6](#) shows the s, e, and f values for special double-precision floating-point numbers.

**Table 3-6 Special Double-Precision Values**

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
-0	1	0	0
+Inf	0	2047	0
-Inf	1	2047	0
NaN	x	2047	nonzero
QNaN	x	2047	1xx..x
SNaN	x	2047	0xx..x and nonzero

[Table 3-7](#) shows hexadecimal and decimal values for some double-precision floating-point numbers.

**Table 3-7 Hexadecimal and Decimal Representation for Selected Double-Precision Values**

Symbol	Hex Value	Decimal Value
NaN_out	7FFFFFFFFFFFFF	QNaN
0	0000000000000000	0.0
-0	8000000000000000	-0.0
1	3FF0000000000000	1.0
2	4000000000000000	2.0
LFPN	7FEFFFFFFFFFFFFF	1.7976931348623157e+308

**Table 3-7     Hexadecimal and Decimal Representation for Selected Double-Precision Values**

Symbol	Hex Value	Decimal Value
SFPN	0010000000000000	2.2250738585072014e-308
LDFPN	000FFFFFFFFFFFFF	2.2250738585072009e-308
SDFPN	0000000000000001	4.9406564584124654e-324

## 3.4 Delay Slots

As described in section 3.4 of [1], the execution of fixed-point and floating-point instructions can be defined in terms of delay slots and functional unit latency.

The number of delay slots is equivalent to the number of additional cycles required after the source operands are read for the result to be available for reading.

The functional unit latency is equivalent to the number of cycles that must pass before the functional unit can start executing the next instruction.

The C66x is fully binary compatible with the C64x+/C674x. Therefore, the number of delay slots and the latencies of all C64x+/C674x instructions are unchanged on the C66x.

All new floating point instructions have a functional unit latency of one cycle (can be fully pipelined). Improvements of the existing floating have also been realized, but in order to maintain a fully backward binary compatibility, new instructions opcode have been created.

- **FMPYDP** is the optimized version of **MPYDP**
- **FADDSP/FSUBSP** are the optimized version of **ADDSP/SUBSP**
- **FADDDP/FADDDP** are the optimized version of **ADDDP/SUBDP**

Table 3-8 shows the number of delay slots associated with each type of instruction.

**Table 3-8      Delay Slot and Functional Unit Latency**

Instruction type	Delay Slots	Functional Unit Latency	Read Cycles <sup>1</sup>	Write Cycles	Branch Taken
NOP	0	1			
Store	0	1	i	i	
Load	4	1	i	i, i+4 <sup>2</sup>	
Branch	5	1	i <sup>3</sup>		i+5
Single cycle	0	1	i	i	
2-cycle	1	1	i	i+1	
3-cycle	2	1	i	i+2	
4-cycle	3	1	i	i+3	
DP compare	1	2	i, i+1	i+1	
2-cycle DP	1	1	i	i+1	
INTDP	4	1	i	i+3, i+4	
MPYSP2DP	4	2	i	i+3, i+4	
ADDDP/SUBDP	6	2	i, i+1	i+5, i+6	
MPYSPDP	6	3	i, i+1	i+5, i+6	
MPYI	8	4	i, i+1, i+2, i+3	i+8	
MPYID	9	4	i, i+1, i+2, i+3	i+8, i+9	
MPYD	9	4	i, i+1, i+2, i+3	i+8, i+9	

1. Cycle i is in the E1 pipeline phase.

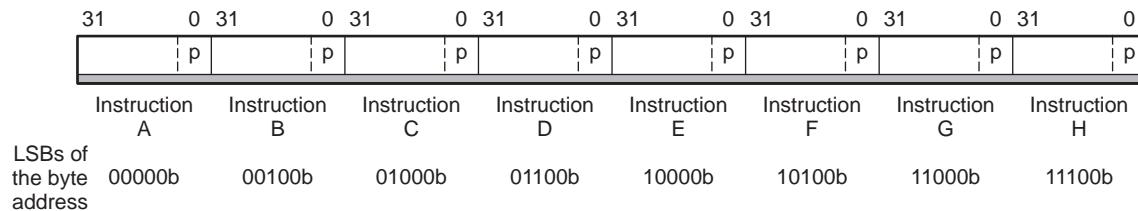
2. For loads, any address modification happens in cycle i. The loaded data is written into the register file in cycle (i+4)

3. The branch to label, branch to IRP, and branch to NRP instructions do not read any general purpose registers

## 3.5 Parallel Operations

Instructions are always fetched eight words at a time. This constitutes a *fetch packet*. CPU, this may be as many as 14 instructions due to the existence of compact instructions in a header based fetch packet. The basic format of a fetch packet is shown in [Figure 3-3](#). Fetch packets are aligned on 256-bit (8-word) boundaries.

**Figure 3-3 Basic Format of a Fetch Packet**



The CPU supports compact 16-bit instructions. Unlike the normal 32-bit instructions, the *p*-bit information for compact instructions is not contained within the instruction opcode. Instead, the *p*-bit is contained within the *p*-bits field within the fetch packet header. See [Section 3.10](#) on page 3-29 for more information.

The execution of the individual noncompact instructions is partially controlled by a bit in each instruction, the *p*-bit. The *p*-bit (bit 0) determines whether the instruction executes in parallel with another instruction. The *p*-bits are scanned from left to right (lower to higher address). If the *p*-bit of instruction *I* is 1, then instruction *I* + 1 is to be executed in parallel with (in the same cycle as) instruction *I*. If the *p*-bit of instruction *I* is 0, then instruction *I* + 1 is executed in the cycle after instruction *I*. All instructions executing in parallel constitute an *execute packet*. An execute packet can contain up to eight instructions. Each instruction in an execute packet must use a different functional unit.

On the CPU, the execute packet can cross fetch packet boundaries, but will be limited to no more than eight instructions in a fetch packet. The last instruction in an execute packet will be marked with its *p*-bit cleared to zero. There are three types of *p*-bit patterns for fetch packets. These three *p*-bit patterns result in the following execution sequences for the eight instructions:

- Fully serial
- Fully parallel
- Partially serial

**Example 3-1** through **Example 3-3** show the conversion of a  $p$ -bit sequence into a cycle-by-cycle execution stream of instructions.

**Example 3-1 Fully Serial p-Bit Pattern in a Fetch Packet**

The eight instructions are executed sequentially.

This  $p$ -bit pattern:

31	0 31	0 31	0 31	0 31	0 31	0 31	0 31	0 31	0
	0	0	0	0	0	0	0	0	0

Instruction A      Instruction B      Instruction C      Instruction D      Instruction E      Instruction F      Instruction G      Instruction H

results in this execution sequence:

Cycle/Execute Packet	Instructions
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H

**End of Example 3-1**

**Example 3-2 Fully Parallel p-Bit Pattern in a Fetch Packet**

All eight instructions are executed in parallel.

This  $p$ -bit pattern:

31	0 31	0 31	0 31	0 31	0 31	0 31	0 31	0
	1	1	1	1	1	1	1	1

Instruction A      Instruction B      Instruction C      Instruction D      Instruction E      Instruction F      Instruction G      Instruction H

results in this execution sequence:

Cycle/Execute Packet	Instructions							
1	A	B	C	D	E	F	G	H

**End of Example 3-2**

**Example 3-3 Partially Serial p-Bit Pattern in a Fetch Packet**

This  $p$ -bit pattern:

31	0 31	0 31	0 31	0 31	0 31	0 31	0 31	0 31	0
	0	0	1	1	0	1	1	0	

Instruction A      Instruction B      Instruction C      Instruction D      Instruction E      Instruction F      Instruction G      Instruction H

results in this execution sequence:

Cycle/Execute Packet	Instructions			
1	A			
2	B			
3	C			D      E
4	F		G	H

**End of Example 3-3**

### 3.5.1 Example Parallel Code

The vertical bars || signify that an instruction is to execute in parallel with the previous instruction. The code for the fetch packet in [Example 3-3](#) on page 3-13 would be represented as this:

```

instruction A
instruction B
|| instruction C
           instruction D
           instruction E
|| instruction F
           instruction G
           instruction H

```

### 3.5.2 Branching Into the Middle of an Execute Packet

If a branch into the middle of an execute packet occurs, all instructions at lower addresses are ignored. In [Example 3-3](#), if a branch to the address containing instruction D occurs, then only D and E execute. Even though instruction C is in the same execute packet, it is ignored. Instructions A and B are also ignored because they are in earlier execute packets. If your result depends on executing A, B, or C, the branch to the middle of the execute packet will produce an erroneous result.

## 3.6 Conditional Operations

Most instructions can be conditional. The condition is controlled by a 3-bit opcode field (*creg*) that specifies the condition register tested, and a 1-bit field (*z*) that specifies a test for zero or nonzero. The four MSBs of every opcode are *creg* and *z*. The specified condition register is tested at the beginning of the E1 pipeline stage for all instructions. For more information on the pipeline, see Chapter 5 “[Pipeline](#)” on page 5-1. If *z* = 1, the test is for equality with zero; if *z* = 0, the test is for nonzero. The case of *creg* = 0 and *z* = 0 is treated as always true to allow instructions to be executed unconditionally. The *creg* field is encoded in the instruction opcode as shown in [Table 3-9](#).

Compact (16-bit) instructions on the DSP do not contain a *creg* field and always execute unconditionally. See “[Compact Instructions on the CPU](#)” on page 3-29 for more information.

**Table 3-9 Registers That Can Be Tested by Conditional Operations**

Specified Conditional Register	Bit:	31	30	29	28	<i>z</i>
Unconditional		0	0	0		0
Reserved		0	0	0		1
B0		0	0	1		<i>z</i>
B1		0	1	0		<i>z</i>
B2		0	1	1		<i>z</i>
A1		1	0	0		<i>z</i>
A2		1	0	1		<i>z</i>
A0		1	1	0		<i>z</i>
Reserved		1	1	x <sup>1</sup>		x <sup>1</sup>

1. x can be any value.

Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. The following execute packet contains two **ADD** instructions in parallel. The first **ADD** is conditional on B0 being nonzero. The second **ADD** is conditional on B0 being zero. The character ! indicates the inverse of the condition.

```
[B0] ADD    .L1      A1,A2,A3
||      [!B0] ADD   .L2      B1,B2,B3
```

The above instructions are mutually exclusive, only one will execute. If they are scheduled in parallel, mutually exclusive instructions are constrained as described in [Section 3.8](#). If mutually exclusive instructions share any resources as described in [Section 3.8](#), they cannot be scheduled in parallel (put in the same execute packet), even though only one will execute.

The act of making an instruction conditional is often called predication and the conditional register is often called the predication register.

## 3.7 SPMASKed Operations

On the CPU, the **SPMASK** and **SPMASKR** instructions can be used to inhibit the execution of instructions from the SPLOOP buffer. The selection of which instruction to inhibit can be specified by the **SPMASK** or **SPMASKR** instruction argument or can be marked by the addition of a caret (^) next to the parallel code marker as shown below:

<code>SPMASK</code> <code>   ^</code>	<code>LDW .D1 *A0,A1 ;This instruction is SPMASKED</code> <code>LDW .D2 *B0,B1 ;This instruction is SPMASKED</code> <code>MPY .M1 A3,A4,A5 ;This instruction is NOT SPMASKED</code>
--	---

See Chapter 8 “Software Pipelined Loop (SPLOOP) Buffer” on page 8-1 for more information.

## 3.8 Resource Constraints

No two instructions within the same execute packet can use the same resources. Also, no two instructions can write to the same register during the same cycle. The following sections describe how an instruction can use each of the resources.

### 3.8.1 Constraints on Instructions Using the Same Functional Unit

Two instructions using the same functional unit cannot be issued in the same execute packet.

The following execute packet is invalid:

```
|| ADD .S1A0, A1, A2 ;S1 is used for
|| SHR .S1A3, 15, A4 ;...both instructions
```

The following execute packet is valid:

```
|| ADD .L1A0, A1, A2 ;Two different functional
|| SHR .S1A3, 15, A4 ;...units are used
```

### 3.8.2 Constraints on the Same Functional Unit Writing in the Same Instruction Cycle

On the C64x+/C674x, the .M unit has two 32-bit write ports; so the results of a 4-cycle 32-bit instruction and a 2-cycle 32-bit instruction operating on the same .M unit can write their results on the same instruction cycle. Any other combination of parallel writes (a 2-cycle instruction writing a 32-bit result and a 4-cycle instruction writing a 64-bit results) on the .M unit will result in a conflict. On the C674x DSP this will result in an exception.

On the C66x, the .M unit has two 64-bit write ports to the register file and the results of a 4-cycle instruction and a 2-cycle instruction operating on the same .M unit can write their results on the same instruction cycle even when the 4-cycle instruction writes a 64-bit results (like a CMPY for example).

However, a 4-cycle instruction (that writes a 128-bit result (like a CMATMPY for example)) can not write its results in the same instruction cycle as a 2-cycle instruction operating on the same .M unit. On the C66x DSP, this will result in an exception and erroneous values being written to the destination registers.

For example, the following sequence is valid and results in A3:A2 and A5 being written by the .M1 unit on the same cycle.

```
CMPY.M1 A0,A1,A3:A2 ; This instruction has 3 delay slots
; and generates a 64 bit result
NOP
AVG2 .M1 A4,A5 ; This instruction has 1 delay slot
NOP ; A3:A2 and A5 get written on this cycle
```

The following sequence is invalid. The attempt to write 160 bits of output through 128-bits of write port will fail. An exception is generated.

```
CMATMPY .M1 A3:A2, A7:A6:A5:A4, A11:A10:A9:A8; This instruction has
; 3 delay slots but
; writes a 128-bit
; result
NOP
MPY .M1 A1,A2,A3 ;This instruction has 1 delay slot
NOP
```

Even though the .L/.S units can also execute 4-cycle and 2-cycle instructions, two independent writes from the same .L or .S unit to the register file onto the same instruction cycle is not supported and will result in an exception and erroneous values being written to the destination registers.

Therefore, the following sequence is invalid since A4 and A5 are written by the .L1 unit on the same cycle.

```
INTSPU .L1A1, A5 ; this instruction has a three delay
; slot (4-cycle instruction)
NOP
DSPINTH.L1A3:A2, A4 ; this instruction has a one delay slot
; (2-cycle instruction)
NOP
```

### 3.8.3 Constraints on Cross Paths (1X and 2X)

#### 3.8.3.1 Maximum Number of Accesses to a Register on the Opposite Register File

Unlike the C64x+/C674x (which allows up to two unit per data path per execute packet to read an operand from its opposite register file), the C66x allows multiple reads through one cross path to the same register.

For example the following sequence is invalid on C64x+/C674x and valid on the C66x:

```
ADD .L1 A0, B0, A1
|| ADD .S1 A2, B0, A2
|| ADD .D1 A3, B0, A3
|| MPY .M1 A4, B0, A4
```

### 3.8.4 Cross Path Sharing

On the C66x, the same register or can be read multiple times through the cross path on the same cycle. The cross paths have been extended to 64-bit. The 64-bit cross path can be used to transport either a 64-bit register value or one 32-bit register value. Even though two 32-bit registers could be read on the opposite register file using the 64-bit cross path, this is not supported by the C66x architecture.

The following is illegal:

```
DADD .L1 A1:A0, B1:B0, A7:A6
|| ADD .S1 A1,B1, A3
|| SUB .D1 A1,B0, A5
```

The following is legal:

```
DADD .L2 B1:B0, A1:A0, B3:B2
|| DADD .S2 B5:B4, A1:A0, B7:B6
|| SUB .D2 B9,B8,B10
```

### 3.8.5 Cross Path Stalls

The DSP introduces a delay clock cycle whenever an instruction attempts to read a register via a cross path that was updated in the previous cycle. This is known as a cross path stall. This stall is inserted automatically by the hardware, no **NOP** instruction is needed. It should be noted that no stall is introduced if the register being read has data placed by a load instruction, or if an instruction reads a result one cycle after the result is generated.

Here are some examples:

```

ADD .S1 A0, A0, A1 ; Stall is introduced; A1 is updated
                     ; 1 cycle before it is used as a
ADD .S2X A1, B0, B1 ; \ cross path source

ADD .S1 A0, A0, A1 ; No stall is introduced; A0 not updated
                     ; 1 cycle before it is used as a cross
ADD .S2X A0, B0, B1 ; \ path source

LDW .D1 *++A0[1], A1 ; No stall is introduced; A1 is the load
                     ; destination
NOP 4 ; NOP 4 represents 4 instructions to
ADD .S2X A1, B0, B1 ; \ be executed between the load and add.
LDW .D1 *++A0[1], A1 ; Stall is introduced; A0 is updated
ADD .S2X A0, B0, B1 ; 1 cycle before it is used as a
                     ; \ cross path source
    
```

It is possible to avoid the cross path stall by scheduling an instruction that reads an operand via the cross path at least one cycle after the operand is updated. With appropriate scheduling, the DSP can provide one cross path operand per data path per cycle with no stalls. In many cases, the TMS320C6000 Optimizing Compiler and Assembly Optimizer automatically perform this scheduling.

### 3.8.6 Constraints on Loads and Stores

The data address paths named DA1 and DA2 are each connected to the .D units in both data paths. Load and store instructions can use an address pointer from one register file while loading to or storing from the other register file. Two load and store instructions using a destination/source from the same register file cannot be issued in the same execute packet. The address register must be on the same side as the .D unit used.

The DA1 and DA2 resources and their associated data paths are specified as T1 and T2, respectively. T1 consists of the DA1 address path and the LD1 and ST1 data paths. LD1 is comprised of LD1a and LD1b to support 64-bit loads; ST1 is comprised of ST1a and ST1b to support 64-bit stores. Similarly, T2 consists of the DA2 address path and the LD2 and ST2 data paths. LD2 is comprised of LD2a and LD2b to support 64-bit loads; ST2 is comprised of ST2a and ST2b to support 64-bit stores. The T1 and T2 designations appear in the functional unit fields for load and store instructions.

The DSP can access words and doublewords at any byte boundary using nonaligned loads and stores. As a result, word and doubleword data does not need alignment to 32-bit or 64-bit boundaries. No other memory access may be used in parallel with a nonaligned memory access. The other .D unit can be used in parallel, as long as it is not performing a memory access.

The following execute packet is invalid:

```

LDNW .D2T2 *B2[B12],B13 ; \ Two memory operations,
|| LDB .D1T1 *A2,A14 ; / one non-aligned
    
```

The following execute packet is valid:

```

LDNW .D2T2 *B2[B12], A13 ; \ One non-aligned memory
                           ; operation,
|| ADD .D1x A12, B13, A14 ; one non-memory .D unit
                           ; / operation
  
```

### 3.8.7 Constraints on Long (40-Bit) Data

Both the C62x and C67x device families had constraints on the number of simultaneous reads and writes of 40-bit data due to shared data paths.

The C674x CPU maintains separate data paths to each functional unit, so these constraints are removed.

The following, for example, is valid:

DDOTPL2 DDOTPL2 STDW STDW SUB SUB SHL SHL	.M1 .M2 .D1 .D2 .L1 .L2 .S1 .S2	A1:A0,A2,A5:A4 B1:B0,B2,B5:B4 A9:A8,*A6 B9:B8,*B6 A25:A24,A20,A31:A30 B25:B24,B20,B31:B30 A11:A10,5,A13:A12 B11:B10,8,B13:B12
--	--	--

### 3.8.8 Constraints on Register Reads

More than four reads of the same register can not occur on the same cycle. On the C66x, there are no restrictions of the number of reads perform from the same register on the same clock cycle.

The following execute packets example are invalid on C64x+/C674x and valid on the C66x:

```

MPY .M1 A1, A1, A4 ; five reads of register A1
|| ADD .L1 A1, A1, A5
|| SUB .D1 A1, A2, A3
  
```

```

MPY .M1 A1, A1, A4 ; five reads of register A1
|| ADD .L1 A1, A1, A5
|| SUB .D2x A1, B2, B3
  
```

### 3.8.9 Constraints on Register Writes

Two instructions cannot write to the same register on the same cycle. Two instructions with the same destination can be scheduled in parallel as long as they do not write to the destination register on the same cycle. For example, an **MPY** issued on cycle *I* followed by an **ADD** on cycle *I* + 1 cannot write to the same register because both instructions write a result on cycle *I* + 1. Therefore, the following code sequence is invalid unless a branch occurs after the **MPY**, causing the **ADD** not to be issued.

```

MPY .M1 A0, A1, A2
ADD .L1 A4, A5, A2
  
```

However, this code sequence is valid:

```

MPY .M1 A0, A1, A2
|| ADD .L1 A4, A5, A2
  
```

[Figure 3-4](#) shows different multiple-write conflicts. For example, **ADD** and **SUB** in execute packet L1 write to the same register. This conflict is easily detectable.

**MPY** in packet L2 and **ADD** in packet L3 might both write to B2 simultaneously; however, if a branch instruction causes the execute packet after L2 to be something other than L3, a conflict would not occur. Thus, the potential conflict in L2 and L3 might not be detected by the assembler. The instructions in L4 do not constitute a write

conflict because they are mutually exclusive. In contrast, because the instructions in L5 may or may not be mutually exclusive, the assembler cannot determine a conflict. If the pipeline does receive commands to perform multiple writes to the same register, the result is undefined.

**Figure 3-4 Examples of the Detectability of Write Conflicts by the Assembler**

```

L1:      ADD .L2      B5, B6, B7 ; \ detectable, conflict
||       SUB .S2      B8, B9, B7 ; /
L2:      MPY .M2      B0, B1, B2 ; \ not detectable
L3:      ADD .L2      B3, B4, B2 ; /
L4:      [!B0]ADD.L2   B5, B6, B7 ; \ detectable, no conflict
||      [B0] SUB.S2    B8, B9, B7 ; /
L5:      [!B1]ADD.L2   B5, B6, B7 ; \ not detectable
||      [B0] SUB.S2    B8, B9, B7 ; /

```

### 3.8.10 Constraints on AMR Writes

A write to the addressing mode register (AMR) using the **MVC** instruction that is immediately followed by a **LD**, **ST**, **ADDA**, or **SUBA** instruction causes a 1 cycle stall, if the **LD**, **ST**, **ADDA**, or **SUBA** instruction uses the A4-A7 or B4-B7 registers for addressing.

### 3.8.11 Constraints on Multicycle NOPs

Two instructions that generate multicycle NOPs cannot share the same execute packet. Instructions that generate a multicycle **NOP** are:

- **NOP** *n* (where *n* > 1)
- **IDLE**
- **BNOP** target, *n* (for all values of *n*, regardless of predication)
- **ADDKPC** label, reg, *n* (for all values of *n*, regardless of predication)

### 3.8.12 Constraints on Unitless Instructions

#### 3.8.12.1 MFENCE Restrictions

The **MFENCE** instruction is a new instruction introduced onto the C66x DSP. This instruction will stall until the completion of all the CPU-triggered memory transactions, including:

- Cache line fills
- Writes from L1D to L2 or from CorePac to MSMC and/or other system endpoints
- Victim write backs
- Block or global coherence operations
- Cache mode changes
- Outstanding XMC prefetch requests

To determine if all the memory transactions are completed, the **MFENCE** instruction checks an internal busy flag. **MFENCE** always wait at least 5 clock cycles before checking the busy flag in order to account for pipeline delays.

The following code is legal:

```

STW     A0, *A1
MFENCE           ; This will wait until the STW write above
                  ; has landed in it's final destination

```

During the course of executing a **MFENCE** operation, any enabled interrupts will still be serviced.

When an interrupt occurs during the execution of a MFENCE instruction, the address of the execute packet containing the MFENCE instruction is saved in IRP or NRP. This forces returning to the MFENCE instruction after interrupt servicing.

The MFENCE operation has the following restrictions:

1. MFENCE should not be executed in parallel with any other instructions. It should be the only instruction in the execute packet. This restriction is not enforced by hardware, and correct operation is not guaranteed if this is done.

The following code is illegal:

```
ADD.L1A0, A3, A5
|| MFENCE.S1
```

2. MFENCE cannot be included in the prolog or body of an SPLOOP. This is not enforced by hardware.
3. MFENCE cannot be included in the epilog of a reloadable SPLOOP.
4. MFENCE should not be included in the epilog of an SPLOOP earlier than 5 cycles before the SPLOOP goes completely IDLE (this is because MFENCE only guarantees 5 cycles of "NOP").

### 3.8.12.2 SPLOOP Restrictions

The NOP, NOP*n*, and BNOP instructions are the only unitless instructions allowed to be used in an SPLOOP(D/W) body. The assembler disallows the use of any other unitless instruction in the loop body.

See Chapter 8 “Software Pipelined Loop (SPLOOP) Buffer” on page 8-1 for more information.

### 3.8.12.3 BNOP <disp>,n

A BNOP instruction cannot be placed in parallel with the following instructions if the BNOP has a non-zero NOP count:

- ADDKPC
- CALLP
- NOP *n*

### 3.8.12.4 DINT

A DINT instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- MVC reg, CSR
- B IRP
- B NRP
- IDLE
- NOP *n* (if *n* > 1)
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

A DINT instruction can be placed in parallel with the NOP instruction.

### 3.8.12.5 IDLE

An IDLE instruction cannot be placed in parallel with the following instructions:

- DINT
- NOP  $n$  (if  $n > 1$ )
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

An IDLE instruction can be placed in parallel with the NOP instruction.

### 3.8.12.6 NOP $n$

A NOP $n$  (with  $n > 1$ ) instruction cannot be placed in parallel with other multicycle NOP counts (ADDKPC, BNOP, CALLP) with the exception of another NOP $n$  where the NOP count is the same. A NOP $n$  (with  $n > 1$ ) instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- RINT
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

### 3.8.12.7 RINT

A RINT instruction cannot be placed in parallel with the following instructions:

- MVC reg, TSR
- MVC reg, CSR
- B IRP
- B NRP
- DINT
- IDLE
- NOP  $n$  (if  $n > 1$ )
- SPKERNEL(R)
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

A RINT instruction can be placed in parallel with the NOP instruction.

### 3.8.12.8 SPKERNEL(R)

An **SPKERNEL(R)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP  $n$  (if  $n > 1$ )
- RINT
- SPLOOP(D/W)
- SPMASK(R)
- SWE
- SWENR

An **SPKERNEL(R)** instruction can be placed in parallel with the **NOP** instruction.

### 3.8.12.9 SPLOOP(D/W)

An **SPLOOP(D/W)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP  $n$  (if  $n > 1$ )
- RINT
- SPKERNEL(R)
- SPMASK(R)
- SWE
- SWENR

An **SPLOOP(D/W)** instruction can be placed in parallel with the **NOP** instruction.

### 3.8.12.10 SPMASK(R)

An **SPMASK(R)** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP  $n$  (if  $n > 1$ )
- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWE
- SWENR

An **SPMASK(R)** instruction can be placed in parallel with the **NOP** instruction.

### 3.8.12.11 SWE

An **SWE** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP  $n$  (if  $n > 1$ )

- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWENR

An **SWE** instruction can be placed in parallel with the **NOP** instruction.

### 3.8.12.12 SWENR

An **SWENR** instruction cannot be placed in parallel with the following instructions:

- DINT
- IDLE
- NOP  $n$  (if  $n > 1$ )
- RINT
- SPLOOP(D/W)
- SPKERNEL(R)
- SWE

An **SWENR** instruction can be placed in parallel with the **NOP** instruction.

### 3.8.13 Constraints on Floating-Point Instructions

If an instruction has a multicycle functional unit latency, it locks the functional unit for the necessary number of cycles. Any new instruction dispatched to that functional unit during this locking period causes undefined results and a resource conflict exception is generated.

An instruction of the following types scheduled on cycle I has the following constraints:

DP compare	No other instruction can use the functional unit on cycles I and I + 1.
ADDDP/SUBDP	No other instruction can use the functional unit on cycles I and I + 1.
MPYI	No other instruction can use the functional unit on cycles I, I + 1, I + 2, and I + 3.
MPYID	No other instruction can use the functional unit on cycles I, I + 1, I + 2, and I + 3.
MPYDP	No other instruction can use the functional unit on cycles I, I + 1, I + 2, and I + 3.

If a cross path is used to read a source in an instruction with a multicycle functional unit latency, you must ensure that no other instructions executing on the same side uses the cross path.

An instruction of the following types scheduled on cycle I using a cross path to read a source, has the following constraints:

DP compare	No other instruction on the same side can use the cross path on cycles I and I + 1.
ADDDP/SUBDP	No other instruction on the same side can use the cross path on cycles I and I + 1.
MPYI	No other instruction on the same side can use the cross path on cycles I, I + 1, I + 2, and I + 3.
MPYID	No other instruction on the same side can use the cross path on cycles I, I + 1, I + 2, and I + 3.
MPYDP	No other instruction on the same side can use the cross path on cycles I, I + 1, I + 2, and I + 3.

Other hazards exist because instructions have varying numbers of delay slots, and need the functional unit read and write ports of varying numbers of cycles. A read or write hazard exists when two instructions on the same functional unit attempt to read or write, respectively, to the register file on the same cycle.

An instruction of the following types scheduled on cycle I has the following constraints:

2-cycle DP	A single-cycle instruction cannot be scheduled on that functional unit on cycle I + 1 due to a write hazard on cycle I + 1.  Another 2-cycle DP instruction cannot be scheduled on that functional unit on cycle I + 1 due to a write hazard on cycle I + 1.
4-cycle	A single-cycle instruction cannot be scheduled on that functional unit on cycle I + 3 due to a write hazard on cycle I + 3.  A multiply (16 16-bit) instruction cannot be scheduled on that functional unit on cycle I + 2 due to a write hazard on cycle I + 3.
INTDP	A single-cycle instruction cannot be scheduled on that functional unit on cycle I + 3 or I + 4 due to a write hazard on cycle I + 3 or I + 4, respectively.  An INTDP instruction cannot be scheduled on that functional unit on cycle I + 1 due to a write hazard on cycle I + 1.
MPYI	A 4-cycle instruction cannot be scheduled on that functional unit on cycle I + 1 due to a write hazard on cycle I + 1.
MPYID	4-cycle instruction cannot be scheduled on that functional unit on cycle I + 4, I + 5, or I + 6.  A MPYDP instruction cannot be scheduled on that functional unit on cycle I + 4, I + 5, or I + 6.  A multiply (16 16-bit) instruction cannot be scheduled on that functional unit on cycle I + 6 due to a write hazard on cycle I + 7.
MPYDP	A 4-cycle instruction cannot be scheduled on that functional unit on cycle I + 4, I + 5, or I + 6.  A MPYI instruction cannot be scheduled on that functional unit on cycle I + 4, I + 5, or I + 6.  A MPYID instruction cannot be scheduled on that functional unit on cycle I + 4, I + 5, or I + 6.  A multiply (16 × 16-bit) instruction cannot be scheduled on that functional unit on cycle I + 7 or I + 8 due to a write hazard on cycle I + 8 or I + 9, respectively.
ADDDP/SUB DP	A single-cycle instruction cannot be scheduled on that functional unit on cycle I + 5 or I + 6 due to a write hazard on cycle I + 5 or I + 6, respectively.  A 4-cycle instruction cannot be scheduled on that functional unit on cycle I + 2 or I + 3 due to a write hazard on cycle I + 5 or I + 6, respectively.  An INTDP instruction cannot be scheduled on that functional unit on cycle I + 2 or I + 3 due to a write hazard on cycle I + 5 or I + 6, respectively.

All of the previous cases deal with double-precision floating-point instructions or the **MPYI** or **MPYID** instructions except for the 4-cycle case. A 4-cycle instruction consists of both single- and double-precision floating-point instructions. Therefore, the 4-cycle case is important for the following single-precision floating-point instructions:

- ADDSP
- SUBSP
- SPINT
- SPTRUNC
- INTSP
- MPYSP

## 3.9 Addressing Modes

The addressing modes on the DSP are linear, circular using BK0, and circular using BK1. The addressing mode is specified by the addressing mode register (AMR), described in 2.8.3 “[Addressing Mode Register \(AMR\)](#)” on page 2-12.

All registers can perform linear addressing. Only eight registers can perform circular addressing: A4-A7 are used by the .D1 unit, and B4-B7 are used by the .D2 unit. No other units can perform circular addressing. **LDB(U)/LDH(U)/LDW**, **STB/STH/STW**, **LDNDW**, **LDNW**, **STNDW**, **STNW**, **LDNW**, **STDW**, **ADDAB/ADDAH/ADDAW/ADDAD**, and **SUBAB/SUBAH/SUBAW** instructions all use AMR to determine what type of address calculations are performed for these registers. There is no **SUBAD** instruction.

### 3.9.1 Linear Addressing Mode

#### 3.9.1.1 LD and ST Instructions

For load and store instructions, linear mode simply shifts the *offsetR/cst* operand to the left by 3, 2, 1, or 0 for doubleword, word, halfword, or byte access, respectively; and then performs an add or a subtract to *baseR* (depending on the operation specified). The **LDNDW** and **STNDW** instructions also support nonscaled offsets. In nonscaled mode, the *offsetR/cst* is not shifted before adding or subtracting from the *baseR*.

For the preincrement, predecrement, positive offset, and negative offset address generation options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

#### 3.9.1.2 ADDA and SUBA Instructions

For integer addition and subtraction instructions, linear mode simply shifts the *src1/cst* operand to the left by 3, 2, 1, or 0 for doubleword, word, halfword, or byte data sizes, respectively, and then performs the add or subtract specified.

### 3.9.2 Circular Addressing Mode

The BK0 and BK1 fields in AMR specify the block sizes for circular addressing, see “[Addressing Mode Register \(AMR\)](#)” on page 2-12.

#### 3.9.2.1 LD and ST Instructions

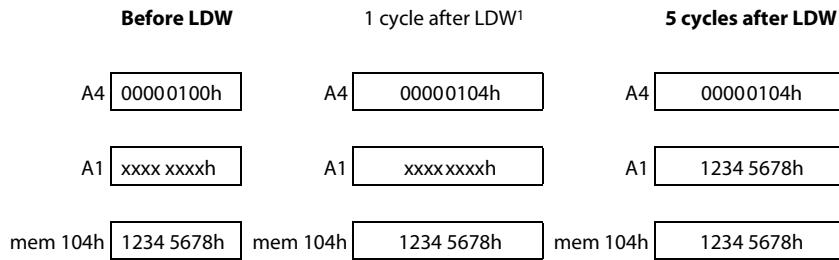
As with linear address arithmetic, *offsetR/cst* is shifted left by 3, 2, 1, or 0 according to the data size, and is then added to or subtracted from *baseR* to produce the final address. Circular addressing modifies this slightly by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N + 1 unchanged after address arithmetic. The resulting address is bounded to  $2^{(N+1)}$  range, regardless of the size of the *offsetR/cst*.

The circular buffer size in AMR is not scaled; for example, a block-size of 8 is 8 bytes, not 8 times the data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. [Example 3-4](#) shows an **LDW** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value in AMR for this example is 00040001h.

#### **Example 3-4 LDW Instruction in Circular Mode**

---

```
LDW      .D1      *++A4 [9] ,A1
```



1. 9h words is 24h bytes. 24h bytes is 4 bytes beyond the 32-byte (20h) boundary 100h-11Fh; thus, it is wrapped around to (124h - 20h = 104h).

**End of Example 3-4**

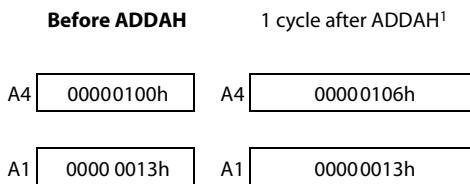
### 3.9.2.2 ADDA and SUBA Instructions

As with linear address arithmetic, *offsetR/cst* is shifted left by 3, 2, 1, or 0 according to the data size, and is then added to or subtracted from *baseR* to produce the final address. Circular addressing modifies this slightly by only allowing bits N through 0 of the result to be updated, leaving bits 31 through N + 1 unchanged after address arithmetic. The resulting address is bounded to  $2^{(N+1)}$  range, regardless of the size of the *offsetR/cst*.

The circular buffer size in AMR is not scaled; for example, a block size of 8 is 8 bytes, not 8 times the data size (byte, halfword, word). So, to perform circular addressing on an array of 8 words, a size of 32 should be specified, or N = 4. [Example 3-5](#) shows an **ADDAH** performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value in AMR for this example is 00040001h.

#### Example 3-5 ADDAH Instruction in Circular Mode

ADDAH .D1 A4,A1,A4



1. 13h halfwords is 26h bytes. 26h bytes is 6 bytes beyond the 32-byte (20h) boundary 100h-11Fh; thus, it is wrapped around to (126h - 20h = 106h).

**End of Example 3-5**

### 3.9.2.3 Circular Addressing Considerations with Nonaligned Memory

Circular addressing may be used with nonaligned accesses. When circular addressing is enabled, address updates and memory accesses occur in the same manner as for the equivalent sequence of byte accesses.

On the CPU, the circular buffer size must be at least 32 bytes. Nonaligned access to circular buffers that are smaller than 32 bytes will cause undefined results.

Nonaligned accesses to a circular buffer apply the circular addressing calculation to *logically adjacent* memory addresses. The result is that nonaligned accesses near the boundary of a circular buffer will correctly read data from both ends of the circular buffer, thus seamlessly causing the circular buffer to “wrap around” at the edges.

Consider, for example, a circular buffer size of 16 bytes. A circular buffer of this size at location 20h, would look like this in physical memory:

1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3
7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
x	x	x	x	x	x	x	x	x	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	

The effect of circular buffering is to make it so that memory accesses and address updates in the 20h-2Fh range stay completely inside this range. Effectively, the memory map behaves in this manner:

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
h	i	j	k	l	m	n	o	p	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	

[Example 3-6](#) shows an LDNW performed with register A4 in circular mode and BK0 = 4, so the buffer size is 32 bytes, 16 halfwords, or 8 words. The value in AMR for this example is 0004 0001h. The buffer starts at address 0020h and ends at 0040h. The register A4 is initialized to the address 003Ah.

#### Example 3-6 LDNW in Circular Mode

LDNW .D1      \*++A4 [2] ,A1

Before LDNW	1 cycle after LDNW <sup>1</sup>	5 cycles after LDNW
A4    0000003Ah	A4    00000022h	A4    00000022h
A1    xxxx xxxxh	A1    xxxxxxxxh	A1    5678 9ABCh
mem 0022h    5678 9ABCh	mem 0022h    5678 9ABCh	mem 0022h    5678 9ABCh

1. 2h words is 8h bytes. 8h bytes is 2 bytes beyond the 32-byte (20h) boundary starting at address 003Ah; thus, it is wrapped around to 0022h (003Ah + 8h = 0022h).

#### 3.9.3 Syntax for Load/Store Address Generation

The DSP has a load/store architecture, which means that the only way to access data in memory is with a load or store instruction. [Table 3-10](#) shows the syntax of an indirect address to a memory location. Sometimes a large offset is required for a load/store. In this case, you can use the B14 or B15 register as the base register, and use a 15-bit constant (*ucst15*) as the offset.

**Table 3-11** describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

**Table 3-10** Indirect Address Generation for Load/Store

Addressing Type	No Modification of Address Register	Preincrement or Predecrement of Address Register	Postincrement or Postdecrement of Address Register
Register indirect	*R	*++R *- -R	*R++ *R- -
Register relative	*+R[ <i>ucst5</i> ]	*++R[ <i>ucst5</i> ]	*R++[ <i>ucst5</i> ]
	*-R[ <i>ucst5</i> ]	*- -R[ <i>ucst5</i> ]	*R- -[ <i>ucst5</i> ]
Register relative with 15-bit constant offset	*+B14/B15[ <i>ucst15</i> ]	not supported	not supported
Base + index	*+R[offsetR] *- R[offsetR]	*++R[offsetR] *- -R[offsetR]	*R++[offsetR] *R- -[offsetR]

**Table 3-11** Address Generator Options for Load/Store

Mode Field				Syntax	Modification Performed
0	0	0	0	*-R[ <i>ucst5</i> ]	Negative offset
0	0	0	1	*+R[ <i>ucst5</i> ]	Positive offset
0	1	0	0	*-R[offsetR]	Negative offset
0	1	0	1	*+R[offsetR]	Positive offset
1	0	0	0	*- -R[ <i>ucst5</i> ]	Predecrement
1	0	0	1	*++R[ <i>ucst5</i> ]	Preincrement
1	0	1	0	*R- -[ <i>ucst5</i> ]	Postdecrement
1	0	1	1	*R++[ <i>ucst5</i> ]	Postincrement
1	1	0	0	*- -R[offsetR]	Predecrement
1	1	0	1	*+R[offsetR]	Preincrement
1	1	1	0	*R- -[offsetR]	Postdecrement
1	1	1	1	*R++[offsetR]	Postincrement

## 3.10 Compact Instructions on the CPU

The CPU supports a header based set of 16-bit-wide compact instructions in addition to the normal 32-bit wide instructions.

### 3.10.1 Compact Instruction Overview

The availability of compact instructions is enabled by the replacement of the eighth word of a fetch packet with a 32-bit header word. The header word describes which of the other seven words of the fetch packet contain compact instructions, which of the compact instructions in the fetch packet operate in parallel, and also contains some decoding information which supplements the information contained in the 16-bit compact opcode. [Figure 3-5](#) compares the standard fetch packet with a header-based fetch packet containing compact instructions.

**Figure 3-5 CPU Fetch Packet Types**

Word	Standard C6000 Fetch Packet	Header-Based Fetch Packet
0	32-bit opcode	16-bit opcode   16-bit opcode
1	32-bit opcode	32-bit opcode
2	32-bit opcode	16-bit opcode   16-bit opcode
3	32-bit opcode	32-bit opcode
4	32-bit opcode	16-bit opcode   16-bit opcode
5	32-bit opcode	32-bit opcode
6	32-bit opcode	16-bit opcode   16-bit opcode
7	32-bit opcode	Header

Within the other seven words of the fetch packet, each word may be composed of a single 32-bit opcode or two 16-bit opcodes. The header word specifies which words contain compact opcodes and which contain 32-bit opcodes.

The compiler will automatically code instructions as 16-bit compact instructions when possible.

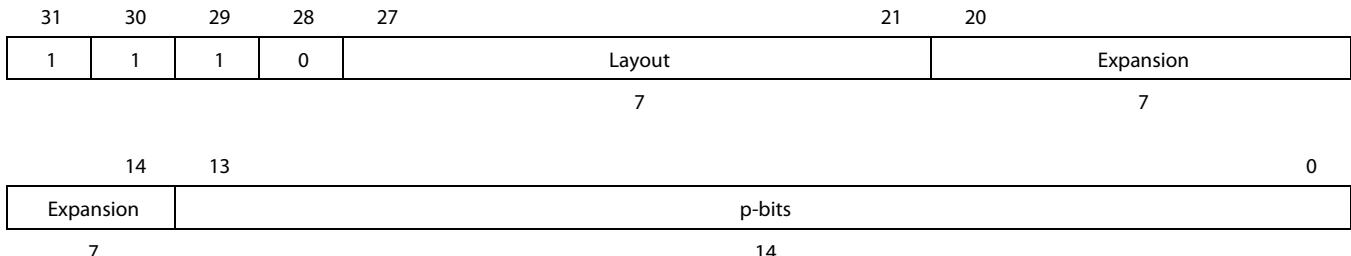
There are a number of restrictions to the use of compact instructions:

- No dedicated predication field
- 3-bit register address field
- Very limited 3 operand instructions
- Subset of 32-bit instructions

### 3.10.2 Header Word Format

[Figure 3-6](#) describes the format of the compact instruction header word.

**Figure 3-6 Compact Instruction Header Format**



**Bits 27-21** (Layout field) indicate which words in the fetch packet contain 32-bit opcodes and which words contain two 16-bit opcodes.

**Bits 20-14** (Expansion field) contain information that contributes to the decoding of all compact instructions in the fetch packet.

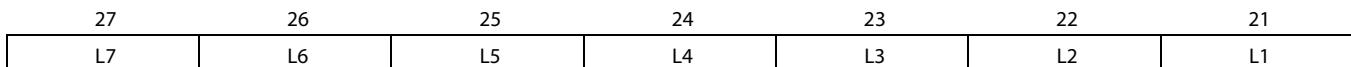
**Bits 13-0** (p-bits field) specify which compact instructions are run in parallel.

#### 3.10.2.1 Layout Field in Compact Header Word

Bits 27-21 of the compact instruction header contains the layout field. This field specifies which of the other seven words in the current fetch packet contain 32-bit full-sized instructions and which words contain two 16-bit compact instructions.

[Figure 3-7](#) shows the layout field in the compact header word and [Table 3-12](#) describes the bits.

**Figure 3-7 Layout Field in Compact Header Word**



**Table 3-12 Layout Field Description in Compact Instruction Packet Header**

Bit	Field	Value	Description
27	L7	0	Seventh word of fetch packet contains a single 32-bit opcode.
		1	Seventh word of fetch packet contains two 16-bit compact instructions.
26	L6	0	Sixth word of fetch packet contains a single 32-bit opcode.
		1	Sixth word of fetch packet contains two 16-bit compact instructions.
25	L5	0	Fifth word of fetch packet contains a single 32-bit opcode.
		1	Fifth word of fetch packet contains two 16-bit compact instructions.
24	L4	0	Fourth word of fetch packet contains a single 32-bit opcode.
		1	Fourth word of fetch packet contains two 16-bit compact instructions.
23	L3	0	Third word of fetch packet contains a single 32-bit opcode.
		1	Third word of fetch packet contains two 16-bit compact instructions.
22	L2	0	Second word of fetch packet contains a single 32-bit opcode.
		1	Second word of fetch packet contains two 16-bit compact instructions.
21	L1	0	First word of fetch packet contains a single 32-bit opcode.
		1	First word of fetch packet contains two 16-bit compact instructions.

### 3.10.2.2 Expansion Field in Compact Header Word

Bits 20-14 of the compact instruction header contains the opcode expansion field. This field specifies properties that apply to all compact instructions contained in the current fetch packet.

[Figure 3-8](#) shows the expansion field in the compact header word and [Table 3-13](#) describes the bits.

**Figure 3-8 Expansion Field in Compact Header Word**

20	19	18	16	15	14
PROT	RS		DSZ	BR	SAT

**Table 3-13 Expansion Field Description in Compact Instruction Packet Header**

Bit	Field	Value	Description
20	PROT	0	Loads are nonprotected (NOPs must be explicit).
		1	Loads are protected (4 NOP cycles added after every LD instruction).
19	RS	0	Instructions use low register set for data source and destination.
		1	Instructions use high register set for data source and destination.
18-16	DSZ	0-7h	Defines primary and secondary data size (see <a href="#">Table 3-14</a> on page 3-32)
15	BR	0	Compact instructions in the S unit are not decoded as branches
		1	Compact Instructions in the S unit are decoded as branches.
14	SAT	0	Compact instructions do not saturate.
		1	Compact instructions saturate.

**Bit 20 (PROT)** selects between protected and nonprotected mode for all **LD** instructions within the fetch packet. When PROT is 1, four cycles of NOP are added after each **LD** instruction within the fetch packet whether the **LD** is in 16-bit compact format or 32-bit format.

**Bit 19 (RS)** specifies which register set is used by compact instructions within the fetch packet. The register set defines which subset of 8 registers on each side are data registers. The 3-bit register field in the compact opcode indicates which one of eight registers is used. When RS is 1, the high register set (A16-A23 and B16-B23) is used; when RS is 0, the low register set (A0-A7 and B0-B7) is used.

**Bits 18-16 (DSZ)** determine the two data sizes available to the compact versions of the **LD** and **ST** instructions in a fetch packet. Bit 18 determines the primary data size that is either word (W) or doubleword (DW). In the case of DW, an opcode bit selects between aligned (DW) and nonaligned (NDW) accesses. Bits 17 and 16 determine the secondary data size: byte unsigned (BU), byte (B), halfword unsigned (HU), halfword (H), word (W), or nonaligned word (NW). [Table 3-14](#) describes how the bits map to data size.

**Bit 15 (BR).** When BR is 1, instructions in the S unit are decoded as branches.

**Bit 14 (SAT).** When SAT is 1, the ADD, SUB, SHL, MPY, MPYH, MPYLH, and MPYHL instructions are decoded as SADD, SUBS, SSHL, SMPY, SMPYH, SMPYLH, and SMPYHL, respectively.

**Table 3-14 LD/ST Data Size Selection**

DSZ Bits			Primary Data Size <sup>1</sup>	Secondary Data Size <sup>2</sup>
18	17	16		
0	0	0	W	BU
0	0	1	W	B
0	1	0	W	HU
0	1	1	W	H
1	0	0	DW/NDW	W
1	0	1	DW/NDW	B
1	1	0	DW/NDW	NW
1	1	1	DW/NDW	H

1. Primary data size is word (W) or doubleword (DW). In the case of DW, aligned (DW) or nonaligned (NDW).

2. Secondary data size is byte unsigned (BU), byte (B), halfword unsigned (HU), halfword (H), word (W), or nonaligned word (NW).

### 3.10.2.3 P-bit Field in Compact Header Word

Unlike normal 32-bit instructions in which the *p*-bit field in each opcode determines whether the instruction executes in parallel with other instructions; the parallel/nonparallel execution information for compact instructions is contained in the compact instruction header word.

Bits 13-0 of the compact instruction header contain the *p*-bit field. This field specifies which of the compact instructions within the current fetch packet are executed in parallel. If the corresponding bit in the layout field is 0 (indicating that the word is a noncompact instruction), then the bit in the *p*-bit field must be zero; that is, 32-bit instructions within compact fetch packets use their own *p*-bit field internal to the 32-bit opcode; therefore, the associated *p*-bit field in the header should always be zero.

Figure 3-9 shows the *p*-bits field in the compact header word and Table 3-15 describes the bits.

**Figure 3-9 P-bits Field in Compact Header Word**

13	12	11	10	9	8	7	6	5	4	3	2	1	0
P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

**Table 3-15 P-bits Field Description in Compact Instruction Packet Header (Part 1 of 2)**

Bit	Field	Value	Description
13	P13	0	Word 6 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 6 (16 most-significant bits) of fetch packet has parallel bit set.
12	P12	0	Word 6 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 6 (16 least-significant bits) of fetch packet has parallel bit set.
11	P11	0	Word 5 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 5 (16 most-significant bits) of fetch packet has parallel bit set.
10	P10	0	Word 5 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 5 (16 least-significant bits) of fetch packet has parallel bit set.

**Table 3-15 P-bits Field Description in Compact Instruction Packet Header (Part 2 of 2)**

<b>Bit</b>	<b>Field</b>	<b>Value</b>	<b>Description</b>
9	P9	0	Word 4 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 4 (16 most-significant bits) of fetch packet has parallel bit set.
8	P8	0	Word 4 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 4 (16 least-significant bits) of fetch packet has parallel bit set.
7	P7	0	Word 3 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 3 (16 most-significant bits) of fetch packet has parallel bit set.
6	P6	0	Word 3 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 3 (16 least-significant bits) of fetch packet has parallel bit set.
5	P5	0	Word 2 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 2 (16 most-significant bits) of fetch packet has parallel bit set.
4	P4	0	Word 2 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 2 (16 least-significant bits) of fetch packet has parallel bit set.
3	P3	0	Word 1 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 1 (16 most-significant bits) of fetch packet has parallel bit set.
2	P2	0	Word 1 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 1 (16 least-significant bits) of fetch packet has parallel bit set.
1	P1	0	Word 0 (16 most-significant bits) of fetch packet has parallel bit cleared.
		1	Word 0 (16 most-significant bits) of fetch packet has parallel bit set.
0	P0	0	Word 0 (16 least-significant bits) of fetch packet has parallel bit cleared.
		1	Word 0 (16 least-significant bits) of fetch packet has parallel bit set.

### 3.10.3 Processing of Fetch Packets

The header information is used to fully define the 32-bit version of the 16-bit instructions. In the case where an execute packet crosses fetch packet boundaries, there are two headers in use simultaneously. Each instruction uses the header information from its fetch packet header.

### 3.10.4 Execute Packet Restrictions

Execute packets that span fetch packet boundaries may not be the target of branches in the case where one of the two fetch packets involved are header-based. The only exception to this is where an interrupt is taken in the cycle before a spanning execute packet reaches E1. The target of the return may be a normally disallowed target.

If the execute packet contains eight instructions, then neither of the two fetch packets may be header-based.

### 3.10.5 Available Compact Instructions

Table 3-16 lists the available compact instructions and their functional unit.

**Table 3-16 Available Compact Instructions (Part 1 of 2)**

Instruction	L Unit	M Unit	S Unit	D Unit
ADD	3		3	3
ADDAW				3
ADDK			3	
AND	3			
BNOP			3	
CALLP			3	
CLR			3	
CMPEQ	3			
CMPGT	3			
CMPGTU	3			
CMPLT	3			
CMPLTU	3			
EXT			3	
EXTU			3	
LDB(U)				3
LDB(U)				3
LDDW				3
LDH(U)				3
LDH(U)				3
LDNDW				3
LDNW				3
LDW				3
LDW				3
MPY		3		
MPYH		3		
MPYHL		3		
MPYLH		3		
MV	3		3	3
MVC			3	
MVK	3		3	3
NEG	3			
NOP			No unit	
OR	3			
SADD	3		3	
SET			3	
SHL			3	
SHR			3	
SHRU			3	
SMPY		3		
SMPYH		3		
SMPYHL		3		
SMPYLH		3		

**Table 3-16 Available Compact Instructions (Part 2 of 2)**

<b>Instruction</b>	<b>L Unit</b>	<b>M Unit</b>	<b>S Unit</b>	<b>D Unit</b>
<b>SPKERNEL</b>	No unit			
<b>SPLOOP</b>	No unit			
<b>SPLOOPD</b>	No unit			
<b>SPMASK</b>	No unit			
<b>SPMASKR</b>	No unit			
<b>SSHLD</b>			3	
<b>SSUB</b>	3			
<b>STB</b>				3
<b>STDW</b>				3
<b>STH</b>				3
<b>STNDW</b>				3
<b>STNW</b>				3
<b>STW</b>				3
<b>STW</b>				3
<b>SUB</b>	3		3	3
<b>SUBAW</b>				3
<b>XOR</b>	3			

### 3.11 Instruction Compatibility

See [Appendix A](#) on page A-1 for a list of the instructions that are common to the C62x, C64x, C64x+, C67x, C67x+, C674x, and C66x DSPs.



# Instruction Descriptions

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

- Assembler syntax
- Functional units
- Operands
- Opcode
- Description
- Execution
- Pipeline
- Instruction type
- Delay slots
- Functional Unit Latency
- Examples

The **ADD** instruction is used as an example to familiarize you with the way each instruction is described. The example describes the kind of information you will find in each part of the individual instruction description and where to obtain more information.

## 4.1 Example

The way each instruction is described.

### Syntax

**EXAMPLE** (.unit) *src*, *dst*

.unit = .L1, .L2, .S1, .S2, .D1, .D2

*src* and *dst* indicate source and destination, respectively. The (.unit) dictates which functional unit the instruction is mapped to (.L1, .L2, .S1, .S2, .M1, .M2, .D1, or .D2).

A table is provided for each instruction that gives the opcode map fields, units the instruction is mapped to, types of operands, and the opcode.

The opcode shows the various fields that make up each instruction. These fields are described in [Table 4-2](#).

There are instructions that can be executed on more than one functional unit. [Table 4-1](#) on page 4-3 shows how this is documented for the **ADD** instruction. This instruction has three opcode map fields: *src1*, *src2*, and *dst*. In the fifth group, the operands have the types *cst5*, *long*, and *long* for *src1*, *src2*, and *dst*, respectively. The ordering of these fields implies *cst5 + long → long*, where + represents the operation being performed by the **ADD**. This operation can be done on .L1 or .L2 (both are specified in the unit column). The *s* in front of each operand signifies that *src1* (*scst5*), *src2* (*slong*), and *dst* (*slong*) are all signed values.

In the ninth group, *src1*, *src2*, and *dst* are *int*, *cst5*, and *int*, respectively. The *u* in front of the *cst5* operand signifies that *src1* (*ucst5*) is an unsigned value. Any operand that begins with *x* can be read from a register file that is different from the destination register file. The operand comes from the register file opposite the destination, if the *x* bit in the instruction is set (shown in the opcode map).

### Description

Instruction execution and its effect on the rest of the processor or memory contents are described. Any constraints on the operands imposed by the processor or the assembler are discussed. The description parallels and supplements the information given by the execution block.

### Execution for .L1, .L2 and .S1, .S2 OpCodes

```
if (cond) src1 + src2 → dst else nop
```

### Execution for .D1, .D2 OpCodes

```
if (cond) src2 + src1 → dst else nop
```

The execution describes the processing that takes place when the instruction is executed. The symbols are defined in [Table 4-1](#).

**Pipeline**

This section contains a table that shows the sources read from, the destinations written to, and the functional unit used during each execution cycle of the instruction.

**Instruction Type**

This section gives the type of instruction. See Section [5.2](#) for information about the pipeline execution of this type of instruction.

**Delay Slots**

This section gives the number of delay slots the instruction takes to execute. See Section [3.4](#) on page 3-9 for an explanation of delay slots.

**Functional Unit Latency**

This section gives the number of cycles that the functional unit is in use during the execution of the instruction.

**Example**

Examples of instruction execution. If applicable, register and memory values are given before and after instruction execution.

**Table 4-1 Relationships Between Operands, Operand Size, Functional Units, and Opfields for Example Instruction (ADD)**

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.L1, .L2	0000011
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	sint	.L1, .L2	0100011
<i>src2</i>	xsint		
<i>dst</i>	slong		
<i>src1</i>	xsint	.L1, .L2	0100001
<i>src2</i>	slong		
<i>dst</i>	slong		
<i>src1</i>	scst5	.L1, .L2	0000010
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.L1, .L2	0100000
<i>src2</i>	slong		
<i>dst</i>	slong		
<i>src1</i>	sint	.S1, .S2	000111
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.S1, .S2	000110
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	010000
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	010010
<i>src1</i>	ucst5		
<i>dst</i>	sint		

## 4.2 ABS

Absolute Value With Saturation

**Syntax**    **ABS** (.unit) *src2*, *dst*

or

**ABS** (.unit) *src2\_h:src2\_l, dst\_h:dst\_l*

unit = .L1 or .L2

**Opcode**

31	29	28	27		23	22		18	17	16
<i>creg</i>	<i>z</i>			<i>dst</i>			<i>src2</i>	0	0	
3	1			5			5			
15	14	13	12	11			5	4	3	2
0	0	0	x		<i>op</i>		1	1	0	s
			1		7				1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	xsint	.L1, .L2	0011010
<i>dst</i>	sint		
<i>src2</i>	slong	.L1, L2	0111000
<i>dst</i>	slong		

**Description**

The absolute value of *src2* is placed in *dst*.

The absolute value of *src2* when *src2* is an sint is determined as follows:

1. If *src2* > 0, then *src2* → *dst*
2. If *src2* < 0 and *src2* ≠ -2<sup>31</sup>, then -*src2* → *dst*
3. If *src2* = -2<sup>31</sup>, then 2<sup>31</sup> - 1 → *dst*

The absolute value of *src2* when *src2* is an slong is determined as follows:

1. If *src2* > 0, then *src2* → *dst\_h:dst\_l*
2. If *src2* < 0 and *src2* ≠ -2<sup>39</sup>, then -*src2* → *dst\_h:dst\_l*
3. If *src2* = -2<sup>39</sup>, then 2<sup>39</sup> - 1 → *dst\_h:dst\_l*

<b>Execution</b>	if (cond) abs(src2) → dst else nop
<b>Pipeline</b>	
	<b>Pipeline Stage</b>
	<b>E1</b>
Read	src2
Written	dst
Unit in use	.L
<b>Instruction Type</b>	Single-cycle
<b>Delay Slots</b>	0
<b>See Also</b>	<a href="#">ABS2</a>
<b>Examples</b>	<b>Example 1</b>

ABS .L1 A1,A5

<b>Before instruction</b>		<b>1 cycle after instruction</b>			
A1	80004E3Dh	-2,147,463,619	A1	80004E3Dh	
A5	xxxxxxxxh		A5	7FFF81C3h	2,147,463,619

### Example 2

ABS .L1 A1,A5

<b>Before instruction</b>		<b>1 cycle after instruction</b>			
A1	3FF60010h	1,073,086,480	A1	3FF60010h	
A5	xxxxxxxxh		A5	3FF60010h	1,073,086,480

### Example 3

ABS .L1 A1:A0,A5:A4

<b>Before instruction</b>		<b>1 cycle after instruction</b>			
A0	FFFF FFFFh	1,073,086,480	A0	FFFF FFFFh	1,073,086,480
A1	0000 00FFh		A1	0000 00FFh	
A4	xxxxxxxxh		A4	0000 0001h	

A5

xxxxxxxh

A5

0000 0000h

## 4.3 ABS2

Absolute Value With Saturation, Signed, Packed 16-Bit

**Syntax**   **ABS2 (.unit) src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22						18	17	16			
				dst						src2		0	0		
3				5						5					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	x	0	0	1	1	0	1	0	1	1	0	s	p
				1									1	1	

Opcode map field used...	For operand type...	Unit
src2	xs2	.L1, .L2
dst	s2	

**Description**

The absolute values of the upper and lower halves of the *src2* operand are placed in the upper and lower halves of the *dst*.

31	16 15	0
	a_hi	a_lo

**ABS2**

↓

↓

31	16 15	0
	abs(a_hi)	abs(a_lo)

Specifically, this instruction performs the following steps for each halfword of *src2*, then writes its result to the appropriate halfword of *dst*:

1. If the value is between 0 and  $2^{15}$ , then value  $\rightarrow dst$
2. If the value is less than 0 and not equal to  $-2^{15}$ , then -value  $\rightarrow dst$
3. If the value is equal to  $-2^{15}$ , then  $2^{15} - 1 \rightarrow dst$



**Note**—This operation is performed on each 16-bit value separately. This instruction does not affect the SAT bit in the CSR.

**Execution**

```

if (cond) {
    abs (lsb16 (src2)) → lsb16 (dst)
    abs (msb16 (src2)) → msb16 (dst)
}
else nop
  
```

**Pipeline**

Pipeline Stage	E1
Read	src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ABS](#)

**Examples** [Example 1](#)

ABS2 .L1 A0,A2

**Before instruction**

A0	FF68 4E3Dh	-152 20029
A2	xxxx xxxxh	

**1 cycle after instruction**

A0	FF68 4E3Dh	
A2	0098 4E3Dh	152 20029

**Example 2**

ABS2 .L1 A0,A2

**Before instruction**

A0	3FF6 F105h	16374 -3835
A2	xxxx xxxxh	

**1 cycle after instruction**

A0	3FF6 F105h	
A2	3FF6 0EFBh	16374 3835

## 4.4 ABSDP

Absolute Value, Double-Precision Floating-Point

**Syntax**   **ABSDP (.unit) src2, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	23 22						18	17		
<i>creg</i>	<i>z</i>	<i>dst</i>						<i>src2</i>					
3	1	5						5					
Reserved	x	1	0	1	1	0	0	1	0	0	0	s	p
	1											1	1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	dp	.S1, .S2
<i>dst</i>	dp	

**Description**

The absolute value of *src2* is placed in *dst*. The 64-bit double-precision operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.

The absolute value of *src2* is determined as follows:

1. If *src2*  $\geq 0$ , then *src2*  $\rightarrow$  *dst*
2. If *src2*  $< 0$ , then  $-\text{src2}$   $\rightarrow$  *dst*

 **Note—**

- 1) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 2) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 3) If *src2* is denormalized, +0 is placed in *dst* and the INEX and DEN2 bits are set.
- 4) If *src2* is +infinity or -infinity, +infinity is placed in *dst* and the INFO bit is set.

**Execution**

```
if (cond)    abs(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2_l, src2_h</i>	
Written	<i>dst_l</i>	<i>dst_h</i>
Unit in use	.S	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type** 2-cycle DP

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [ABS](#), [ABSSP](#)

**Example** ABSDP .S1 A1:A0,A3:A2

<b>Before instruction</b>				<b>2 cycles after instruction</b>			
A1:A0	C004 0000h	0000 0000h	-2.5	A1:A0	C004 0000h	0000 0000h	
A3:A2	xxxx xxxxh	xxxx xxxxh		A3:A2	4004 0000h	0000 0000h	2.5

## 4.5 ABSSP

## Absolute Value, Single-Precision Floating-Point

**Syntax**      **ABSSP** (.unit) *src2, dst*

unit = .S1 or .S2

## *Opcode*

31	29	28	27					23	22					18	17	16									
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				0	0														
3	1	5				5				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	x	1	1	1	1	0	0	1	0	0	0	0	s	p									
															1			1		1					

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xsp	.S1, .S2
<i>dst</i>	sp	

### *Description*

The absolute value of *src2* is placed in *dst*.

The absolute value of  $src2$  is determined as follows:

1. If  $src2 \geq 0$ , then  $src2 \rightarrow dst$
  2. If  $src2 < 0$ , then  $-src2 \rightarrow dst$



**Note—**

- 1) If  $src2$  is SNaN,  $\text{NaN\_out}$  is placed in  $dst$  and the INVAL and NAN2 bits are set.
  - 2) If  $src2$  is QNaN,  $\text{NaN\_out}$  is placed in  $dst$  and the NAN2 bit is set.
  - 3) If  $src2$  is denormalized,  $+0$  is placed in  $dst$  and the INEX and DEN2 bits are set.
  - 4) If  $src2$  is  $+\infty$  or  $-\infty$ ,  $+\infty$  is placed in  $dst$  and the INFO bit is set.

## *Execution*

```
if (cond)    abs(src2) → dst  
else nop
```

## *Pipeline*

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	<i>.S</i>

**Instruction Type** Single-cycle

## ***Delay Slots***

### **Functional Unit Latency**

*See Also* ABS, ABSDP

**Example** ABSSP - S1X B1 A5

**Before instruction****1 cycle after instruction**

B1	<table border="1"><tr><td>C020 0000h</td></tr></table>	C020 0000h	-2.5	B1	<table border="1"><tr><td>C020 0000h</td></tr></table>	C020 0000h	
C020 0000h							
C020 0000h							
A5	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A5	<table border="1"><tr><td>4020 0000h</td></tr></table>	4020 0000h	2.5
xxxxxxxxh							
4020 0000h							

## 4.6 ADD

Add Two Signed Integers Without Saturation

**Syntax**    **ADD (.unit) src1, src2, dst**

or

**ADD (.L1 or .L2) src1, src2\_h:src2\_l, dst\_h:dst\_l**

or

**ADD (.S1 or .S2) src1\_h:src1\_l, src2, dst\_h:dst\_l**

or

**ADD (.D1 or .D2) src2, src1, dst** (if the cross path form is not used)

or

**ADD (.D1 or .D2) src1, src2, dst** (if the cross path form is used)

or

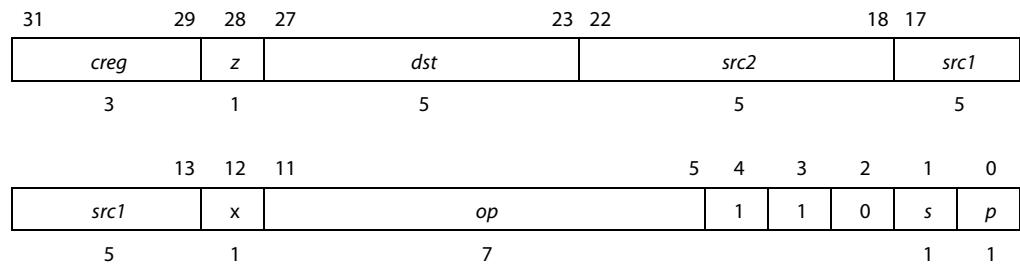
**ADD (.D1 or .D2) src2, src1, dst** (if the cross path form is used with a constant)

unit = .D1, .D2, .L1, .L2, .S1, .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L3	<a href="#">Figure D-4</a>
	L3i	<a href="#">Figure D-5</a>
	Lx1	<a href="#">Figure D-11</a>
.S	S3	<a href="#">Figure F-17</a>
	Sx2op	<a href="#">Figure F-24</a>
	Sx1	<a href="#">Figure F-26</a>
.D	Dx2op	<a href="#">Figure C-17</a>
.L, .S, .D	LSDx1	<a href="#">Figure G-4</a>

**Opcode**    .L unit



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.L1, .L2	0000011
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	sint	.L1, .L2	0100011
<i>src2</i>	xsint		
<i>dst</i>	slong		
<i>src1</i>	xsint	.L1, .L2	0100001
<i>src2</i>	slong		
<i>dst</i>	slong		
<i>src1</i>	scst5	.L1, .L2	0000010
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.L1, .L2	0100000
<i>src2</i>	slong		
<i>dst</i>	slong		

**Opcode** .S unit

31	29	28	27	2 2	1 17
				3 2	8
<i>creg</i>	<i>z</i>	<i>dst</i>	<i>src2</i>	<i>src1</i>	
3		5	5	5	4
14	13	12	11	2	1 0
<i>src1</i>	<i>op</i>	<i>x</i>	1011011000	<i>s</i>	<i>p</i>
			10	1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	slong	.S1, .S2	0000001
<i>src2</i>	xsint		
<i>dst</i>	slong		

**Description for .L1, .L2 and .S1, .S2 OpCodes**

*src2* is added to *src1*. The result is placed in *dst*.

**Execution for .L1, .L2 and .S1, .S2 OpCodes**

```
if (cond) src1 + src2 → dst
else nop
```

**Opcode** .D unit (if the cross path form is not used)

31	29	28	27	23 22	18 17
<i>creg</i>	<i>z</i>	<i>dst</i>		<i>src2</i>	<i>src1</i>
3	1	5		5	5
13 12		7 6 5 4 3 2 1 0			
<i>src1</i>	<i>op</i>	1 0 0 0 0 0 1 0			
5		6			1 1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	010000
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	010010
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Opcode** .D unit (if the cross path form is used)

31	29	28	27			23	22			18	17
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>	
3	1			5				5		5	
		13	12	11	10	9	8	7	6	5	4
		<i>src1</i>	<i>x</i>	1	0	1	0	1	0	1	1
		5	1							0	0
										<i>s</i>	<i>p</i>
										1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sint	.D1, .D2
<i>src2</i>	xsint	
<i>dst</i>	sint	

**Opcode** .D unit (if the cross path form is used with a constant)

31	29	28	27			23	22			18	17
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>	
3	1			5				5		5	
		13	12	11	10	9	8	7	6	5	4
		<i>src1</i>	<i>x</i>	1	0	1	0	1	1	1	1
		5	1							0	0
										<i>s</i>	<i>p</i>
										1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>scst5</i>	.D1, .D2
<i>src2</i>	<i>xsint</i>	
<i>dst</i>	<i>sint</i>	

**Description for .D1, .D2 Opcodes**

*src1* is added to *src2*. The result is placed in *dst*.

**Execution for .D1, .D2 Opcodes**

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S, or .D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADDU](#), [ADD2](#), [SADD](#)

**Examples**

**Example 1**

ADD .L2X A1,B1,B2

Before instruction			1 cycle after instruction		
A1	0000325Ah	12,890	A1	0000325Ah	
B1	FFFFFFFFFF12h	-238	B1	FFFFFFFFFF12h	
B2	xxxxxxxxh		B2	0000316Ch	12,652

**Example 2**

ADD .L1 A1,A3:A2,A5:A4

Before instruction			1 cycle after instruction			
A1	0000325Ah	12,890	A1	0000325Ah		
A3:A2	0000 00FFh	FFFFFFFFFF12h	-228 <sup>1</sup>	A3:A2	000000FFh	FFFFFFFFFF12h
A5:A4	0000 0000h	0000 0000h		A5:A4	00000000h	0000316Ch
					12,652 <sup>1</sup>	

1. Signed 40-bit (long) integer

### Example 3

ADD .L1 -13,A1,A6

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A1	0000325Ah 12,890	A1	0000325Ah
A6	xxxxxxxxh	A6	0000324Dh 12,877

### Example 4

ADD .D1 A1,26,A6

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A1	0000325Ah 12,890	A1	0000325Ah
A6	xxxxxxxxh	A6	00003274h 12,916

### Example 5

ADD .D1 B0,5,A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B0	00000007h	B0	00000007h
A2	xxxxxxxxh	A2	0000 000Ch 12

## 4.7 ADDAB

Add Using Byte Addressing Mode

**Syntax**    **ADDAB** (.unit) *src2, src1, dst*

or

**ADDAB** (.unit) B14/B15, ucst15, *dst*

unit = .D1 or .D2

**Opcode**

31	29	28	27	23	22	18	17		
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>			
3	1		5		5		5		
13	12	7	6	5	4	3	2	1	0
<i>src1</i>		<i>op</i>		1	0	0	0	<i>s</i>	<i>p</i>
5		6						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	110000
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	110010
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description**

*src1* is added to *src2* using the byte addressing mode specified for *src2*. The addition defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “Addressing Mode Register (AMR)” on page 2-12). The result is placed in *dst*.

**Execution**

```
if (cond) src2 +a src1 → dst else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.D

**Opcode**

31	30	29	28	27		23	22						
0	0	0	1		<i>dst</i>		<i>ucst15</i>						
					5		15						
					8      7      6      5      4      3      2      1      0								
					<i>ucst15</i>	<i>y</i>	0	1	1	1	1	<i>s</i>	<i>p</i>
					15		1					1	1

**Description**

This instruction reads a register (*baseR*), B14 (*y* = 0) or B15 (*y* = 1), and adds a 15-bit unsigned constant (*ucst15*) to it, writing the result to a register (*dst*). This instruction is executed unconditionally, it cannot be predicated.

The offset, *ucst15*, is added to *baseR*. The result of the calculation is written into *dst*. The addressing arithmetic is always performed in linear mode.

## 4.7 ADDAB

## Chapter 4—Instruction Descriptions

The *s* bit determines the unit used (D1 or D2) and the file the destination is written to: *s* = 0 indicates the unit is D1 and *dst* is in the A register file; and *s* = 1 indicates the unit is D2 and *dst* is in the B register file.

**Execution**  $B14/B15 + ucst15 \rightarrow dst$

**Pipeline**

Pipeline Stage	E1
Read	<i>B14/B15</i>
Written	<i>dst</i>
Unit in use	.D

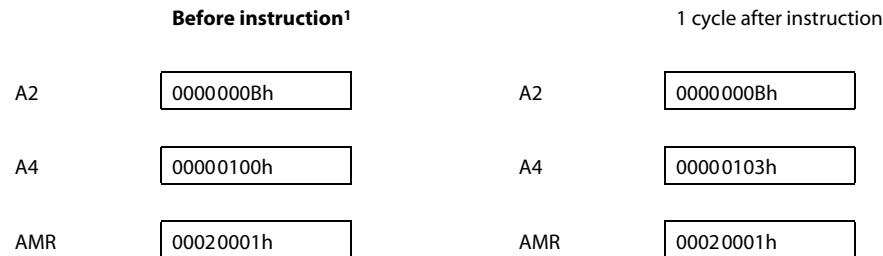
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADDAD](#), [ADDAH](#), [ADDAW](#)

**Examples** **Example 1**

ADDAB .D1 A4, A2, A4



1. BK0 = 2: block size = 8  
A4 in circular addressing mode using BK0

**Example 2**

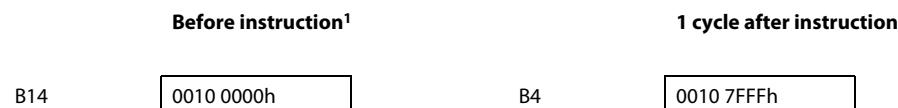
ADDAB .D1X B14, 42h, A4



1. Using linear addressing.

**Example 3**

ADDAB .D2 B14, 7FFFh, B4



1. Using linear addressing.

## 4.8 ADDAD

Add Using Doubleword Addressing Mode

**Syntax** **ADDAD (.unit) src2, src1, dst**

unit = .D1 or .D2

**Opcode**

31	29	28	27	23	22	18	17
src2	z		dst		src2	src1	
3	1		5		5	5	
src1		op		1	0	0	0
5		6		1	0	s	p
						1	1

Opcode map field used...	For operand type...	Unit	Opfield
src2	sint	.D1, .D2	111100
src1	sint		
dst	sint		
src2	sint	.D1, .D2	111101
src1	ucst5		
dst	sint		

**Description** *src1* is added to *src2* using the doubleword addressing mode specified for *src2*. The addition defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “Addressing Mode Register (AMR)” on page 2-12). *src1* is left shifted by 3 due to doubleword data sizes. The result is placed in *dst*.



**Note**—There is no SUBAD instruction.

**Execution**

```
if (cond)    src2 + (src1 << 3) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADDAB](#), [ADDAH](#), [ADDAW](#)

**Example** ADDAD .D1 A1,A2,A3

**Before instruction**

A1	0000 1234h	4660
----	------------	------

**1 cycle after instruction**

A1	0000 1234h
----	------------

## 4.8 ADDAD

## Chapter 4—Instruction Descriptions

A2	<table border="1"><tr><td>00000002h</td></tr></table>	00000002h	2	A2	<table border="1"><tr><td>0000 0002h</td></tr></table>	0000 0002h	
00000002h							
0000 0002h							
A3	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A3	<table border="1"><tr><td>0000 1244h</td></tr></table>	0000 1244h	4676
xxxxxxxxh							
0000 1244h							

## 4.9 ADDAH

Add Using Halfword Addressing Mode

**Syntax**    **ADDAH (.unit) src2, src1, dst**

or

**ADDAH (.unit) B14/B15, ucst15, dst**

unit = .D1 or .D2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3	1		5			5	5

13	12	7	6	5	4	3	2	1	0
<i>src1</i>		<i>op</i>		1	0	0	0	<i>s</i>	<i>p</i>
5		6						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	110100
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	110110
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description** *src1* is added to *src2* using the halfword addressing mode specified for *src2*. The addition defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “Addressing Mode Register (AMR)” on page 2-12). *src1* is left shifted by 1. The result is placed in *dst*.

**Execution**

```
if (cond) src2 +a src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.D

**Opcode**

31	30	29	28	27		23	22	
0	0	0	1		<i>dst</i>			<i>ucst15</i>
						5		15
						8	7	6
						<i>ucst15</i>	<i>y</i>	1
						15	1	
						5	4	3
						<i>1</i>	1	1
						1	0	
						1	1	1

**Description**

This instruction reads a register (*baseR*), B14 (*y* = 0) or B15 (*y* = 1), and adds a scaled 15-bit unsigned constant (*ucst15*) to it, writing the result to a register (*dst*). This instruction is executed unconditionally, it cannot be predicated.

The offset, *ucst15*, is scaled by a left-shift of 1 and added to *baseR*. The result of the calculation is written into *dst*. The addressing arithmetic is always performed in linear mode.

The *s* bit determines the unit used (D1 or D2) and the file the destination is written to: *s* = 0 indicates the unit is D1 and *dst* is in the A register file; and *s* = 1 indicates the unit is D2 and *dst* is in the B register file.

**Execution**  $B14/B15 + (ucst15 << 1) \rightarrow dst$

**Pipeline**

Pipeline Stage	E1
Read	<i>B14/B15</i>
Written	<i>dst</i>
Unit in use	.D

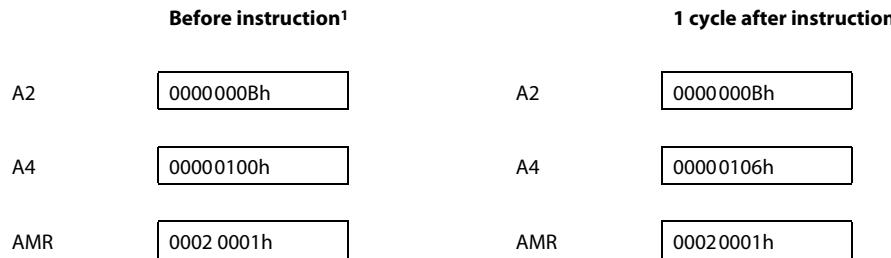
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADDAB](#), [ADDAD](#), [ADDAW](#)

**Examples** [Example 1](#)

ADDAH .D1 A4, A2, A4



1. BK0 = 2: block size = 8  
A4 in circular addressing mode using BK0

**Example 2**

ADDAH .D1X B14, 42h, A4



1. Using linear addressing.

**Example 3**

ADDAH .D2 B14, 7FFFh, B4



1. Using linear addressing.

## 4.10 ADDAW

Add Using Word Addressing Mode

**Syntax**   **ADDAW (.unit) src2, src1, dst**

or

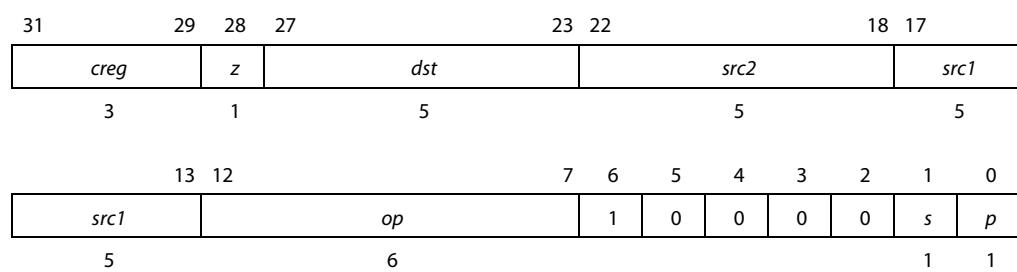
**ADDAW (.unit) B14/B15, ucst15, dst**

unit = .D1 or .D2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Dx5	<a href="#">Figure C-18</a>
	Dx5p	<a href="#">Figure C-19</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	111000
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	111010
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description** *src1* is added to *src2* using the word addressing mode specified for *src2*. The addition defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “Addressing Mode Register (AMR)” on page 2-12). *src1* is left shifted by 2. The result is placed in *dst*.

**Execution**

```
if (cond) src2 +a src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.D

**Opcode**

31	30	29	28	27			23	22										
0	0	0	1		<i>dst</i>				<i>ucst15</i>									
					5				15									
						8	7	6	5	4	3	2	1	0				
						<i>ucst15</i>		<i>y</i>	1	1	1	1	<i>s</i>	<i>p</i>			1	1
						15		1										

**Description**

This instruction reads a register (*baseR*), B14 (*y* = 0) or B15 (*y* = 1), and adds a scaled 15-bit unsigned constant (*ucst15*) to it, writing the result to a register (*dst*). This instruction is executed unconditionally, it cannot be predicated.

The offset, *ucst15*, is scaled by a left-shift of 2 and added to *baseR*. The result of the calculation is written into *dst*. The addressing arithmetic is always performed in linear mode.

The *s* bit determines the unit used (D1 or D2) and the file the destination is written to: *s* = 0 indicates the unit is D1 and *dst* is in the A register file; and *s* = 1 indicates the unit is D2 and *dst* is in the B register file.

**Execution** B14/B15 + (*ucst15* << 2) → *dst*

**Pipeline**

Pipeline Stage	E1
Read	B14/B15
Written	<i>dst</i>
Unit in use	.D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** ADDAB, ADDAD, ADDAH

**Examples** Example 1

ADDAW .D1 A4,2,A4

**Before instruction<sup>1</sup>**

A4	00020000h
----	-----------

AMR	00020001h
-----	-----------

**1 cycle after instruction**

A4	00020000h
----	-----------

AMR	00020001h
-----	-----------

1. BK0 = 2: block size = 8  
 A4 in circular addressing mode using BK0

**Example 2**

ADDAW .D1X B14,42h,A4

**Before instruction<sup>1</sup>**

B14	0020 1000h
-----	------------

1. Using linear addressing.

**1 cycle after instruction**

A4	0020 1108h
----	------------

**Example 3**

ADDAW .D2 B14,7FFFh,B4

**Before instruction<sup>1</sup>**

B14	0010 0000h
-----	------------

1. Using linear addressing.

**1 cycle after instruction**

B4	0011 FFFCh
----	------------

## 4.11 ADDDP

Add Two Double-Precision Floating-Point Values

**Syntax** **ADDDP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, .S2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	<i>src1</i>
3	1		5			5	5
		13	12	11		5	4
		<i>src1</i>	<i>x</i>	<i>op</i>		1	1
		5	1	7		0	
						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	dp	.L1, .L2	0011000
<i>src2</i>	x dp		
<i>dst</i>	dp		
<i>src1</i>	dp	.S1, .S2	1110010
<i>src2</i>	x dp		
<i>dst</i>	dp		

**Description**

*src2* is added to *src1*. The result is placed in *dst*.



**Note—**

- 1) This instruction takes the rounding mode from and sets the warning bits in the floating-point adder configuration register (FADCR), not the floating-point auxiliary configuration register (FAUCR) as for other .S unit instructions.
- 2) If rounding is performed, the INEX bit is set.
- 3) If one source is SNaN or QNaN, the result is NaN\_out. If either source is SNaN, the INVAL bit is also set.
- 4) If one source is +infinity and the other is -infinity, the result is NaN\_out and the INVAL bit is set.
- 5) If one source is signed infinity and the other source is anything except NaN or signed infinity of the opposite sign, the result is signed infinity and the INFO bit is set.
- 6) If overflow occurs, the INEX and OVER bits are set and the results are rounded as follows (LFPN is the largest floating-point number):

Overflow Output Rounding Mode				
Result Sign	Nearest Even	Zero	+Infinity	Infinity
+	+infinity	+LFPN	+infinity	+LFPN
-	-infinity	-LFPN	-LFPN	-infinity

- 7) If underflow occurs, the INEX and UNDER bits are set and the results are rounded as follows (SPFN is the smallest floating-point number):

<b>Result Sign</b>	<b>Underflow Output Rounding Mode</b>				
	<b>Nearest Even</b>	<b>Zero</b>	<b>+Infinity</b>	<b>Infinity</b>	
+	+0	+0	+SFPN	+0	
-	-0	-0	-0	-SFPN	

- 8) If the sources are equal numbers of opposite sign, the result is +0 unless the rounding mode is +infinity, in which case the result is -0.
- 9) If the sources are both 0 with the same sign or both are denormalized with the same sign, the sign of the result is negative for negative sources and positive for positive sources.
- 10) A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is set.

**Execution**

```
if (cond) src1 + src2 → dst
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>	<b>E5</b>	<b>E6</b>	<b>E7</b>
<b>Read</b>		<i>src1_l,</i> <i>src2_l</i>	<i>src1_h,</i> <i>src2_h</i>				
<b>Written</b>						<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.L or .S	.L or .S					

The low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the ADDDP, CMPEQDP, CMPLTDP, CMPGTDP, MPYDP, MPYSPDP, MPYSP2DP, or SUBDP instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type** ADDDP/SUBDP**Delay Slots** 6**Functional Unit Latency** 2**See Also** [ADD](#), [ADDSP](#), [ADDU](#), [SUBDP](#)**Example** ADDDP .L1X B1:B0,A3:A2,A5:A4**Before instruction**

B1:B0	4021 3333h	3333 3333h
-------	------------	------------

**7 cycles after instruction**

B1:B0	4021 3333h	4021 3333h	8.6
-------	------------	------------	-----

A3:A2	C004 0000h	0000 0000h
-------	------------	------------

A3:A2	C004 0000h	0000 0000h	-2.5
-------	------------	------------	------

A5:A4	xxxx xxxxh	xxxx xxxxh
-------	------------	------------

A5:A4	4018 6666h	6666 6666h	6.1
-------	------------	------------	-----

## 4.12 ADDK

Add Signed 16-Bit Constant to Register

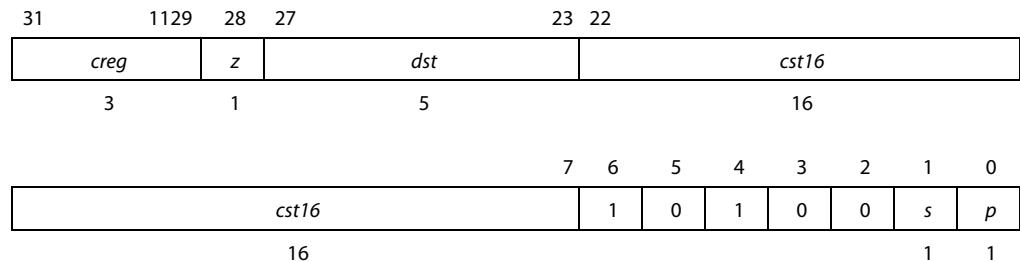
**Syntax** **ADDK (.unit) cst, dst**

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Sx5	<a href="#">Figure F-25</a>

**Opcode**



Opcode map field used...	For operand type...	Unit
cst16	scst16	.S1, .S2
dst	uint	

**Description** A 16-bit signed constant, *cst16*, is added to the *dst* register specified. The result is placed in *dst*.

**Execution**  
`if (cond) cst16 + dst → dst  
else nop`

**Pipeline**

Pipeline Stage	E1
Read	cst16
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**Example** ADDK .S1 15401,A1

**Before instruction**

A1      002137E1h      2,176,993

**1 cycle after instruction**

A1      0021740Ah      2,192,394

## 4.13 ADDKPC

Add Signed 7-Bit Constant to Program Counter

**Syntax** **ADDKPC** (.unit) *src1*, *dst*, *src2*

unit = .S2

**Opcode**

31	1229	28	27			23	22	16									
				<i>creg</i>	<i>z</i>		<i>dst</i>	<i>src1</i>									
3		1				5										7	
15	1313	12	11	10	9	8	7	6	5	4	3	2	1	0			
		<i>src2</i>	0	0	0	0	1	0	1	1	0	0	0	<i>s</i>	<i>p</i>	1	1
			3														

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>scst7</i>	.S2
<i>src2</i>	<i>ucst3</i>	
<i>dst</i>	<i>uint</i>	

**Description**

A 7-bit signed constant, *src1*, is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **ADDKPC** instruction (PCE1). The result is placed in *dst*. The 3-bit unsigned constant, *src2*, specifies the number of NOP cycles to insert after the current instruction. This instruction helps reduce the number of instructions needed to set up the return address for a function call.

The following code:

```
B      .S2    func
MVKL   .S2  LABEL, B3
MVKH   .S2  LABEL, B3
NOP    3
LABEL
```

could be replaced by:

```
B      .S2    func
ADDKPC .S2  LABEL, B3, 4
LABEL
```

The 7-bit value coded as *src1* is the difference between LABEL and PCE1 shifted right by 2 bits. The address of LABEL must be within 9 bits of PCE1.

Only one **ADDKPC** instruction can be executed per cycle. An **ADDKPC** instruction cannot be paired with any relative branch instruction in the same execute packet. If an **ADDKPC** and a relative branch are in the same execute packet, and if the **ADDKPC** instruction is executed when the branch is taken, behavior is undefined.

The **ADDKPC** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **IDLE**, **BNOP**, and the multicycle **NOP**.

**Execution**

```
if (cond) (scst7 << 2) + PCE1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [B](#), [BNOP](#)

**Example**

```
ADDKPC .S2    LABEL,B3,4
LABEL:
```

**Before instruction<sup>1</sup>**

PCE1	0040 13DCh
------	------------

B3	xxxx xxxxh
----	------------

**1 cycle after instruction**

B3	0040 13E0h
----	------------

- LABEL is equal to 0040 13DCh.

## 4.14 ADDSP

Add Two Single-Precision Floating-Point Values

**Syntax**   **ADDSP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, .S2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
		13	12	11		5	4
		<i>src1</i>	<i>x</i>	<i>op</i>		1	1
		5	1	7		0	
						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sp	.L1, .L2	0010000
<i>src2</i>	xsp		
<i>dst</i>	sp		
<i>src1</i>	sp	.S1, .S2	1110000
<i>src2</i>	xsp		
<i>dst</i>	sp		

**Description**

*src2* is added to *src1*. The result is placed in *dst*.



**Note—**

- 1) This instruction takes the rounding mode from and sets the warning bits in the floating-point adder configuration register (FADCR), not in the floating-point auxiliary configuration register (FAUCR) as for other .S unit instructions.
- 2) If rounding is performed, the INEX bit is set.
- 3) If one source is SNaN or QNaN, the result is NaN\_out. If either source is SNaN, the INVAL bit is also set.
- 4) If one source is +infinity and the other is -infinity, the result is NaN\_out and the INVAL bit is set.
- 5) If one source is signed infinity and the other source is anything except NaN or signed infinity of the opposite sign, the result is signed infinity and the INFO bit is set.
- 6) If overflow occurs, the INEX and OVER bits are set and the results are rounded as follows (LFPN is the largest floating-point number):

Overflow Output Rounding Mode				
Result Sign	Nearest Even	Zero	+Infinity	Infinity
+	+infinity	+LFPN	+infinity	+LFPN
-	-infinity	-LFPN	-infinity	-infinity

- 7) If underflow occurs, the INEX and UNDER bits are set and the results are rounded as follows (SPFN is the smallest floating-point number):

<b>Result Sign</b>	<b>Underflow Output Rounding Mode</b>			
	<b>Nearest Even</b>	<b>Zero</b>	<b>+Infinity</b>	<b>Infinity</b>
+	+0	+0	+SFPN	+0
-	-0	-0	-0	-SFPN

- 8) If the sources are equal numbers of opposite sign, the result is +0 unless the rounding mode is +infinity, in which case the result is -0.
- 9) If the sources are both 0 with the same sign or both are denormalized with the same sign, the sign of the result is negative for negative sources and positive for positive sources.
- 10) A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is set.

**Execution**

```
if (cond) src1 + src2 → dst
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
<b>Read</b>		src1, src2		
<b>Written</b>				dst
<b>Unit in use</b>	.L or .S			

**Instruction Type** 4-cycle**Delay Slots** 3**Functional Unit Latency** 1**See Also** [ADD](#), [ADDDP](#), [ADDU](#), [SUBSP](#)**Example** ADDSP .L1 A1,A2,A3**Before instruction****4 cycles after instruction**

A1	C020 0000h	A1	C020 0000h	-2.5
A2	4109 999Ah	A2	4109 999Ah	8.6
A3	xxxxxxxxh	A3	40C3 3334h	6.1

## 4.15 ADDSUB

Parallel ADD and SUB Operations On Common Inputs

**Syntax** **ADDSUB (.unit) src1, src2, dst\_o:dst\_e**

unit = .L1 or .L2

**Opcode**

31	30	29	28	27	dst			24	23	22	src2			src1		18	17
0	0	0	1					0								5	5
					5						5						
						x	0	0	0	1	1	0	0	1	1	0	0
														s	p		
																1	1

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

**Description**

The following is performed in parallel:

1. *src2* is added to *src1*. The result is placed in *dst\_o*.
2. *src2* is subtracted from *src1*. The result is placed in *dst\_e*.

**Execution**

$$\begin{aligned} src1 + src2 &\rightarrow dst_o \\ src1 - src2 &\rightarrow dst_e \end{aligned}$$

**Instruction Type**

Single-cycle

**Delay Slots**

0

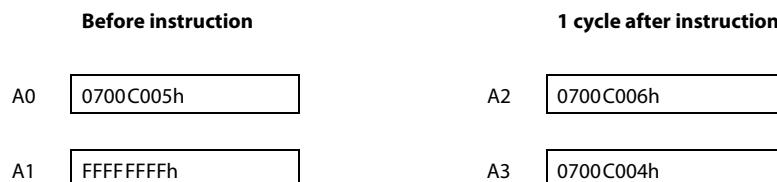
**See Also**

[ADDSUB2](#), [SADDSUB](#)

**Examples**

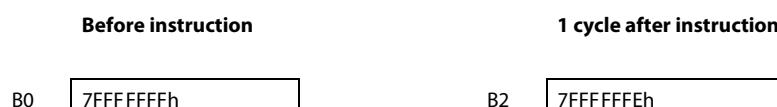
**Example 1**

ADDSUB .L1 A0,A1,A3:A2



**Example 2**

ADDSUB .L2X B0,A1,B3:B2



## 4.15 ADDSUB

## Chapter 4—Instruction Descriptions

A1      

00000001h
-----------

B3      

80000000h
-----------

## 4.16 ADDSUB2

Parallel ADD2 and SUB2 Operations On Common Inputs

**Syntax** **ADDSUB2 (.unit) src1, src2, dst\_o:dst\_e**

unit = .L1 or .L2

**Opcode**

31	30	29	28	27	dst			24	23	22	src2			src1		18	17	
0	0	0	1					0								5	5	
					5								5			5		
					13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>		x	0	0	0	1	1	0	1	1	1	0	s	p		1	1	
					5	1										1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sint	.L1, .L2
<i>src2</i>	xsint	
<i>dst</i>	dint	

**Description** For the **ADD2** operation, the upper and lower halves of the *src2* operand are added to the upper and lower halves of the *src1* operand. The values in *src1* and *src2* are treated as signed, packed 16-bit data and the results are written in signed, packed 16-bit format into *dst\_o*.

For the **SUB2** operation, the upper and lower halves of the *src2* operand are subtracted from the upper and lower halves of the *src1* operand. The values in *src1* and *src2* are treated as signed, packed 16-bit data and the results are written in signed, packed 16-bit format into *dst\_e*.

**Execution**

```

lsb16(src1) + lsb16(src2) → lsb16(dst_o)
msb16(src1) + msb16(src2) → msb16(dst_o)
lsb16(src1) - lsb16(src2) → lsb16(dst_e)
msb16(src1) - msb16(src2) → msb16(dst_e)

```

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADDSUB](#), [SADDSUB2](#)

**Examples** [Example 1](#)

ADDSUB2 .L1 A0,A1,A3:A2

Before instruction		1 cycle after instruction	
A0	0700 C005h	A2	0701 C004h
A1	FFFF 0001h	A3	06FF C006h

### Example 2

ADDSUB2 .L2X B0,A1,B3:B2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B0	7FFF 8000h	B2	8000 8001h
A1	FFFF FFFFh	B3	7FFE 7FFFh

### Example 3

ADDSUB2 .L1 A0,A1,A3:A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	9000 9000h	A2	1000 1000h
A1	8000 8000h	A3	1000 1000h

### Example 4

ADDSUB2 .L1 A0,A1,A3:A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	9000 8000h	A2	1000 F000h
A1	8000 9000h	A3	1000 1000h

## 4.17 ADDU

Add Two Unsigned Integers Without Saturation

**Syntax**    **ADDU (.L1 or .L2) src1, src2, dst\_h:dst\_l**

or

**ADDU (.L1 or .L2) src1, src2\_h:src2\_l, dst\_h:dst\_l**

or

**ADDU (.S1 or .S2) src1\_h:src1\_l, src2, dst\_h:dst\_l**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
13	12	11		5	4	3	2
<i>src1</i>	<i>x</i>		<i>op</i>		1	1	0
1			7			1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	uint	.L1, .L2	0101011
<i>src2</i>	xuint		
<i>dst</i>	ulong		
<i>src1</i>	xuint	.L1, .L2	0101001
<i>src2</i>	ulong		
<i>dst</i>	ulong		

**Opcode**    .S unit

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>
3			5		5		4
14	13	12	11		2	1	0
<i>src1</i>	<i>op</i>	<i>x</i>		1011101000		<i>s</i>	<i>p</i>
					10		

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	slong	.S1, .S2	0000001
<i>src2</i>	xsint		
<i>dst</i>	slong		

**Description** *src2* is added to *src1*. The result is placed in *dst*.

**Execution**

```
if (cond) src1 + src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD](#), [SADD](#)

**Examples** [Example 1](#)

ADDU .L1 A1,A2,A5:A4

Before instruction			1 cycle after instruction		
A1	0000325Ah	12,890 <sup>1</sup>	A1	0000325Ah	
A2	FFFFFFFFFF12h	4,294,967,058 <sup>1</sup>	A2	FFFFFFFFFF12h	
A5:A4	xxxx xxxxh		A5:A4	00000001h	0000316Ch 4,294,979,948 <sup>2</sup>

1. Unsigned 32-bit integer
2. Unsigned 40-bit (long) integer

### Example 2

ADDU .L1 A1,A3:A2,A5:A4

Before instruction			1 cycle after instruction		
A1	0000325Ah	12,890 <sup>1</sup>	A1	0000325Ah	
A3:A2	0000 00FFh	FFFFFFFFFF12h 1,099,511,627,538 <sup>2</sup>	A3:A2	0000 00FFh	FFFFFFFFFF12h
A5:A4	0000 0000h	0000 0000h 0	A5:A4	0000 0000h	0000316Ch 12,652 <sup>2</sup>

1. Unsigned 32-bit integer
2. Unsigned 40-bit (long) integer

4.18 ADD2

### Add Two 16-Bit Integers on Upper and Lower Register Halves

**Syntax**      ADD2 (.unit) *src1*, *src2*, *dst*

unit = .S1, .S2, .L1, .L2, .D1, .D2

*Opcode* .S unit

31	29	28	27	23 22				18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>		
3	1	5				5				5		
13	12	11	10	9	8	7	6	5	4	3	2	1    0
<i>src1</i>	x	0	0	0	0	0	1	1	0	0	0	s    p
5	1									1	1	

### ***Opcode*** .L Unit

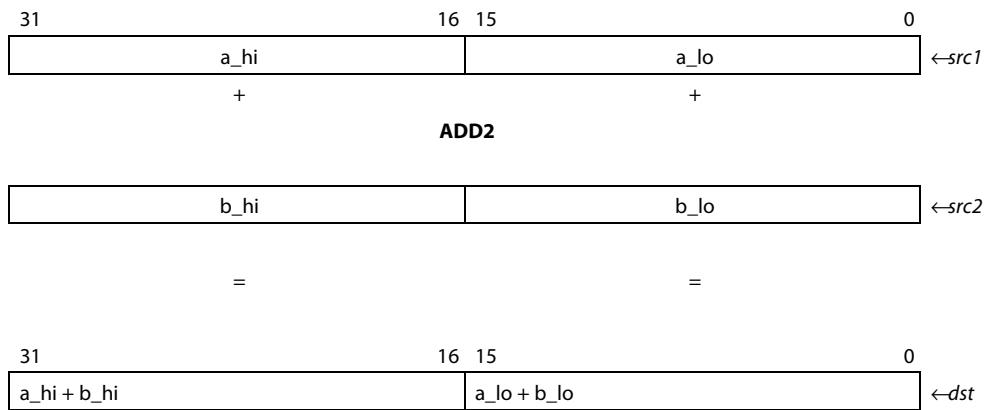
*Opcode* .D unit

31	29	28	27	23 22				18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>		
3	1	5				5				5		
13	12	11	10	9	8	7	6	5	4	3	2	1 0
<i>src1</i>	x	1	0	0	1	0	0	1	1	0	0	s p
5	1									1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	i2	.D1, .D2
<i>src2</i>	xi2	
<i>dst</i>	i2	

**Description** The upper and lower halves of the *src1* operand are added to the upper and lower halves of the *src2* operand. The values in *src1* and *src2* are treated as signed, packed 16-bit data and the results are written in signed, packed 16-bit format into *dst*.

For each pair of signed packed 16-bit values found in the *src1* and *src2*, the sum between the 16-bit value from *src1* and the 16-bit value from *src2* is calculated to produce a 16-bit result. The result is placed in the corresponding positions in the *dst*. The carry from the lower half add does not affect the upper half add.



**Execution**

```
if (cond) {
    msb16(src1) + msb16(src2) → msb16(dst);
    lsb16(src1) + lsb16(src2) → lsb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S, .L, .D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD](#), [ADD4](#), [SADD2](#), [SUB2](#)

**Examples** **Example 1**

ADD2 .S1X A1,B1,A2

**Before instruction**
**1 cycle after instruction**

A1	<input type="text" value="002137E1h"/>	33 14305	A1	<input type="text" value="002137E1h"/>
A2	<input type="text" value="xxxx xxxxh"/>		A2	<input type="text" value="03BB1C99h"/> 955 7321
B1	<input type="text" value="039A E4B8h"/>	922 58552	B1	<input type="text" value="039A E4B8h"/>

**Example 2**

ADD2 .L1 A0,A1,A2

**Before instruction**
**1 cycle after instruction**

A0	<input type="text" value="0021 37E1h"/>	33 14305 signed	A0	<input type="text" value="0021 37E1h"/>
A1	<input type="text" value="039A E4B8h"/>	922 -6984 signed	A1	<input type="text" value="039A E4B8h"/>
A2	<input type="text" value="xxxx xxxxh"/>		A2	<input type="text" value="03BB 1C99h"/> 955 7321 signed

## 4.19 ADD4

Add Without Saturation, Four 8-Bit Pairs for Four 8-Bit Results

**Syntax** **ADD4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	dst					src2					src1	
3	1			5				5				5			
	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	src1	x	1	1	0	0	1	0	1	1	1	0	s	p	

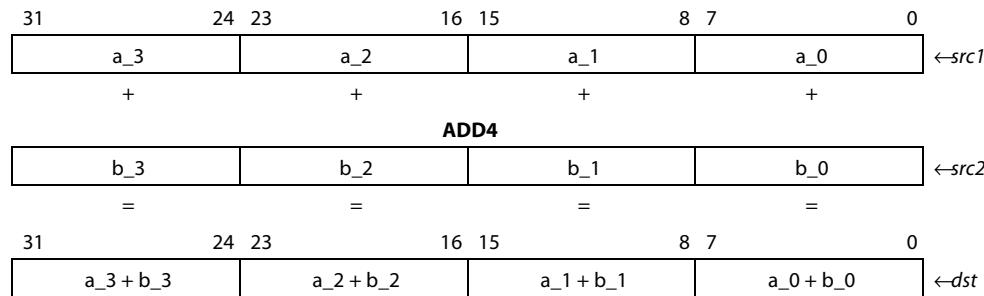
Opcode map field used...	For operand type...	Unit
src1	i4	.L1, .L2
src2	xi4	
dst	i4	

**Description**

Performs 2s-complement addition between packed 8-bit quantities. The values in *src1* and *src2* are treated as packed 8-bit data and the results are written into *dst* in a packed 8-bit format.

For each pair of packed 8-bit values in *src1* and *src2*, the sum between the 8-bit value from *src1* and the 8-bit value from *src2* is calculated to produce an 8-bit result. No saturation is performed. The carry from one 8-bit add does not affect the add of any other 8-bit add. The result is placed in the corresponding positions in *dst*:

- The sum of *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*.
- The sum of *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*.
- The sum of *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*.
- The sum of *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.



**Execution**

```
if (cond) {
    byte0(src1) + byte0(src2) → byte0(dst);
    byte1(src1) + byte1(src2) → byte1(dst);
    byte2(src1) + byte2(src2) → byte2(dst);
    byte3(src1) + byte3(src2) → byte3(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD](#), [ADD2](#), [SADDU4](#), [SUB4](#)

**Examples**

**Example 1**

ADD4 .L1 A0,A1,A2

Before instruction		1 cycle after instruction	
A0	FF 68 4E 3Dh	-1 104 78 61	A0 FF 68 4E 3Dh
A1	3F F6 F1 05h	63 -10 -15 5	A1 3F F6 F1 05h
A2	xxxx xxxxh		A2 3E 5E 3F 42h 62 94 63 66

**Example 2**

ADD4 .L1 A0,A1,A2

Before instruction		1 cycle after instruction	
A0	4A E2 D3 1Fh	74 226 211 31	A0 4A E2 D3 1Fh

A1	<table border="1"><tr><td>32 1A C1 28h</td></tr></table>	32 1A C1 28h	50 26 -63 40	A1	<table border="1"><tr><td>32 1A C1 28h</td></tr></table>	32 1A C1 28h	
32 1A C1 28h							
32 1A C1 28h							
A2	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh		A2	<table border="1"><tr><td>7C FC 94 47h</td></tr></table>	7C FC 94 47h	124 252 148 71
xxxx xxxxh							
7C FC 94 47h							

4.20 AND

## Bitwise AND

**Syntax**      **AND** (.unit) *src1, src2, dst*

unit= .L1, .L2, .S1, .S2, .D1, or .D2

**Opcodes**      Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z	dst		src2		src1		x		opfield		1	1	0	s	p
3			5		5					7						

Opcode fields:

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1111011
src1,src2,dst	scst5,xop2,dst	.L1 or .L2	1111010

**Opcode**      Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1111011

**Opcodes**      Opcode for .S Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	opfield	1	0	0	0	s	p		
3			5		5		5		6								

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	op1,xop2,dst	.S1 or .S2	011111
src1,src2,dst	scst5,xop2,dst	.S1 or .S2	011110

**Opcode**      Opcode for .S Unit, 1/2 src, unconditional

Opcode fields:

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	011111

**Opcode**    Opcode for .D Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
creg		z		dst		src2		src1	x	1	0	opfield	1	1	0	0	s	p	
3				5		5		5				4							

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	op1,xop2,dst	.D1 or .D2	0110
src1,src2,dst	scst5,xop2,dst	.D1 or .D2	0111

**Description**    The AND instruction performs the bit-wise AND between two source registers and stores the result in a third register.

**Execution**    if(cond) src1 AND src2 -> dst  
else nop

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [OR](#), [XOR](#)

**Example**    A0 == 0xffffffff  
AND .L 15,A0,A15  
A15 == 0x0000000f

```
A0 == 0xdeadbeef
A1 == 0xbeefbabe
AND .L A0,A1,A2
A2 == 0x9eadbaae
```

```
A1 == 0xdeadbeef
A0 == 0x12340000
A3 == 0xbeefbabe
A2 == 0x00006789
AND .L A1:A0,A3:A2,A9:A8
A9 == 0x9eadbaae
A8 == 0x00000000
```

## 4.21 ANDN

Bitwise AND Invert

**Syntax**    **ANDN (.unit) src1, src2, dst**

unit= .L1, .L2, .D1, .D2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x	opfield	1	1	0	s	p		
3			5		5		5		7							

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1111100

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	1	0	s	p	
3			5		5		5		5		7						

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1111100

**Opcode**    Opcode for .D Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	1	0	opfield	1	1	0	0	s	p		
3			5		5		5				4								

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.D1 or .D2	0000

**Opcode**    Opcode for .S Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	1	1	opfield	1	1	0	0	s	p		
3			5		5		5				4								

## 4.21 ANDN

## Chapter 4—Instruction Descriptions

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.S1 or .S2	0110

**Opcode**      Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	1	1	opfield	1	1	0	0	s	p	

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	0110

**Description**      The ANDN instruction performs the bitwise AND between *op1* and the bitwise inverse of *xop2*, storing the result in *dst*.

This instruction can be used in the following way, to perform a multiplex function:

```
;A10 and A11 contain 2 different quantities
;A12 is used to select between A10 and A11 on a bit by bit basis
    AND.D1A10, A12, A13
||    ANDN.S1 A11, A12, A14
;
;A13 contains the final multiplexed output of A10 and A11 using A12
OR.L1    A13, A14, A13
```

**Execution**      if(cond) src1 AND ~src2 -> dst  
else nop

**Instruction Type**      Single-cycle

**Delay Slots**      0

**Functional Unit Latency**      1

**See Also**      [OR](#), [XOR](#), [AND](#)

**Example**

```
A0 == 0xdeadbeef
A1 == 0xbeefbabe
ANDN .L A0,A1,A2
A2 == 0x40000441

A0 == 0x00000000 ; all 0's
A1 == 0x00000000 ; all 0's
ANDN .L A0,A1,A2
A2 == 0x00000000

A0 == 0x00000000 ; all 0's
A1 == 0xffffffff ; all 1's
ANDN .L A0,A1,A2
A2 == 0x00000000

A0 == 0xffffffff ; all 1's
A1 == 0x00000000 ; all 0's
ANDN .L A0,A1,A2
A2 == 0xffffffff

A0 == 0xffffffff ; all 1's
A1 == 0xffffffff ; all 1's
ANDN .L A0,A1,A2
A2 == 0x00000000

A0 == 0xaaaaaaaa
A1 == 0xcccccccc
```

```

ANDN .L A0,A1,A2 ; all bit combinations
A2 == 0x22222222

A3 == 0xdeadbeef
A2 == 0x00000000
A13 == 0xbeefbabe
A12 == 0x00000000
ANDN . A3:A2,A13:A12,A11:A10
A11 == 0x40000441
A10 == 0x00000000

A3 == 0x00000000
A2 == 0xfffffffff
A13 == 0xfffffffff
A12 == 0x00000000
ANDN . A3:A2,A13:A12,A11:A10
A11 == 0x00000000
A10 == 0xfffffffff

A3 == 0xfffffffff
A2 == 0xaaaaaaaa
A13 == 0xfffffffff
A12 == 0xcccccccc
ANDN . A3:A2,A13:A12,A11:A10
A11 == 0x00000000
A10 == 0x22222222

```

## 4.22 AVG2

Average, Signed, Packed 16-Bit

**Syntax**   **AVG2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst					src2					src1		18	17
3																	5
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
	src1	x	0	1	0	0	1	1	1	1	0	0	s	p			
	5		1												1	1	

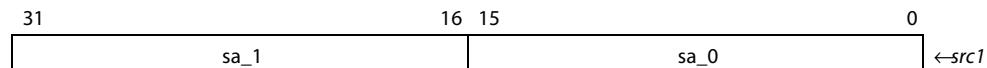
Opcode map field used...	For operand type...	Unit
src1	s2	.M1, .M2
src2	xs2	
dst	s2	

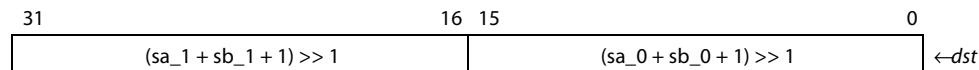
**Description**

Performs an averaging operation on packed 16-bit data. For each pair of signed 16-bit values found in *src1* and *src2*, **AVG2** calculates the average of the two values and returns a signed 16-bit quantity in the corresponding position in the *dst*.

The averaging operation is performed by adding 1 to the sum of the two 16-bit numbers being averaged. The result is then right-shifted by 1 to produce a 16-bit result.

No overflow conditions exist.


**AVG2**

 $\downarrow$ 
 $\downarrow$ 


**Execution**

```

if (cond) {
    ((lsb16(src1) + lsb16(src2) + 1) >> 1) → lsb16(dst);
    ((msb16(src1) + msb16(src2) + 1) >> 1) → msb16(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1	E2
Read	$src1, src2$	
Written		$dst$
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [AVGU4](#)

**Example** AVG2 .M1 A0,A1,A2

**Before instruction**

A0	6198 4357h	24984 17239
A1	7582 AE15	30082 -20971
A2	xxxx xxxxh	

**2 cycles after instruction**

A0	6198 4357h
A1	7582 AE15h
A2	6B8D F8B6h

27533 -1866

## 4.23 AVGU4

Average, Unsigned, Packed 8-Bit

**Syntax** **AVGU4 (.unit) *src1*, *src2*, *dst***

*unit* = .M1 or .M2

**Opcode**

31	29	28	27	dst					src2					src1		18	17
				3	1			5						5			5
						13	12	11	10	9	8	7	6	5	4	3	2
						<i>src1</i>	x	0	1	0	0	1	0	1	1	0	0
						5	1									1	1

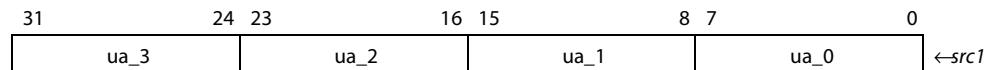
Opcode map field used...	For operand type...	Unit
<i>src1</i>	u4	.M1, .M2
<i>src2</i>	xu4	
<i>dst</i>	u4	

**Description**

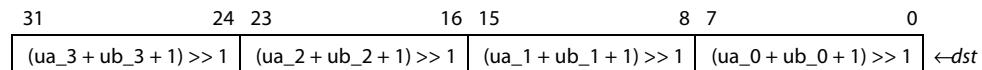
Performs an averaging operation on packed 8-bit data. The values in *src1* and *src2* are treated as unsigned, packed 8-bit data and the results are written in unsigned, packed 8-bit format. For each unsigned, packed 8-bit value found in *src1* and *src2*, **AVGU4** calculates the average of the two values and returns an unsigned, 8-bit quantity in the corresponding positions in the *dst*.

The averaging operation is performed by adding 1 to the sum of the two 8-bit numbers being averaged. The result is then right-shifted by 1 to produce an 8-bit result.

No overflow conditions exist.

**AVGU4**

↓                    ↓                    ↓                    ↓



**Execution**

```

if (cond) {
    ((ubyte0(src1) + ubyte0(src2) + 1) >> 1) → ubyte0(dst);
    ((ubyte1(src1) + ubyte1(src2) + 1) >> 1) → ubyte1(dst);
    ((ubyte2(src1) + ubyte2(src2) + 1) >> 1) → ubyte2(dst);
    ((ubyte3(src1) + ubyte3(src2) + 1) >> 1) → ubyte3(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [AVG2](#)

**Example** AVGU4 .M1 A0,A1,A2

**Before instruction**

A0 

1A 2E 5F 4Eh
--------------

 26 46 95 78  
unsigned

**2 cycles after instruction**

A0 

1A 2E 5F 4Eh
--------------

A1 

9E F2 6E 3Fh
--------------

 158 242 110 63  
unsigned

A1 

9E F2 6E 3Fh
--------------

A2 

xxxx xxxxh
------------

A2 

5C 90 67 47h
--------------

 92 144 103 71  
unsigned

## 4.24 B

### Branch Using a Displacement

**Syntax**    **B** (.unit) label

unit = .S1 or .S2

#### Opcode

31	29	28	27								
<i>creg</i>	<i>z</i>	<i>cst21</i>									
3	1									21	
								7	6	5	4
								3	2	1	0
								<i>cst21</i>	0	0	1
								21	0	0	<i>s</i>
								1	1		

Opcode map field used...	For operand type...	Unit
<i>cst21</i>	<i>scst21</i>	.S1, .S2

**Description**    A 21-bit signed constant, *cst21*, is shifted left by 2 bits and is added to the address of the first instruction of the fetch packet that contains the branch instruction. The result is placed in the program fetch counter (PFC). The assembler/linker automatically computes the correct value for *cst21* by the following formula:

$$cst21 = (\text{label} - \text{PCE1}) \gg 2$$

If two branches are in the same execute packet and both are taken, behavior is undefined.

Two conditional branches can be in the same execute packet if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a true condition, the code executes in a well-defined way.



#### Note—

- 1) PCE1 (program counter) represents the address of the first instruction in the fetch packet in the E1 stage of the pipeline.  
PFC is the program fetch counter.
- 2) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
- 3) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.
- 4) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

- 5) A relative branch instruction cannot be in the same execute packet as an ADDKPC instruction.

**Execution**

```
if (cond) (cst21 << 2) + PCE1 → PFC
else nop
```

**Pipeline**

Pipeline Stage	Target Instruction						
	E1	PS	PW	PR	DP	DC	E1
Read							
Written							
Branch taken							✓
Unit in use	.S						

**Instruction Type** Branch

**Delay Slots** 5

**Example** Table 4-2 gives the program counter values and actions for the following code example.

```
0000 0000      B .S1 LOOP
0000 0004      ADD .L1 A1, A2, A3
0000 0008      ||ADD .L2 B1, B2, B3
0000 000CLOOP: MPY.M1X A3, B3, A4
0000 0010      ||SUB .D1 A5, A6, A6
0000 0014      MPY .M1 A3, A6, A5
0000 0018      MPY .M1 A6, A7, A8
0000 001C      SHR .S1 A4, 15, A4
0000 0020      ADD .D1 A4, A6, A4
```

**Table 4-2 Program Counter Values for Branch Using a Displacement Example**

Cycle	Program Counter Value	Action
Cycle 0	0000 0000h	Branch command executes (target code fetched)
Cycle 1	0000 0004h	
Cycle 2	0000 000Ch	
Cycle 3	0000 0014h	
Cycle 4	0000 0018h	
Cycle 5	0000 001Ch	
Cycle 6	0000 000Ch	Branch target code executes
Cycle 7	0000 0014h	

## 4.25 B

Branch Using a Register

**Syntax**    **B (.unit) src2**

unit = .S2

**Opcode**

31	29	28	27	26	25	24	23	22	18	17	16
<i>creg</i>		<i>z</i>	0	0	0	0	0		<i>src2</i>	0	0
3		1						5			
15	14	13	12	11	10	9	8	7	6	5	4
0	0	0	x	0	0	1	1	0	1	1	0
				0	1			1	0	0	s
					1				1	1	p

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.S2

**Description**    *src2* is placed in the program fetch counter (PFC).

If two branches are in the same execute packet and are both taken, behavior is undefined.

Two conditional branches can be in the same execute packet if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a true condition, the code executes in a well-defined way.



**Note—**

- 1) This instruction executes on .S2 only. PFC is program fetch counter.
- 2) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
- 3) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.

- 4) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

**Execution**

```
if (cond) src2 → PFC
else nop
```

**Pipeline**

Pipeline Stage	Target Instruction						
	E1	PS	PW	PR	DP	DC	E1
Read	src2						
Written							
Branch taken							✓
Unit in use	.S2						

**Instruction Type** Branch

**Delay Slots**

5

**Example**

Table 4-3 on page 4-61 gives the program counter values and actions for the following code example. In this example, the B10 register holds the value 1000 000Ch.

1000 0000	B	.S2 B10
1000 0004	ADD	.L1 A1, A2, A3
1000 0008	ADD	.L2 B1, B2, B3
1000 000C	MPY	.M1X A3, B3, A4
1000 0010	SUB	.D1 A5, A6, A6
1000 0014	MPY	.M1 A3, A6, A5
1000 0018	MPY	.M1 A6, A7, A8
1000 001C	SHR	.S1 A4, 15, A4
1000 0020	ADD	.D1 A4, A6, A4

**Table 4-3 Program Counter Values for Branch Using a Register Example**

Cycle	Program Counter Value	Action
Cycle 0	1000 0000h	Branch command executes (target code fetched)
Cycle 1	1000 0004h	
Cycle 2	1000 000Ch	
Cycle 3	1000 0014h	
Cycle 4	1000 0018h	
Cycle 5	1000 001Ch	
Cycle 6	1000 000Ch	Branch target code executes
Cycle 7	1000 0014h	

## 4.26 B IRP

Branch Using an Interrupt Return Pointer

**Syntax**    **B (.unit) IRP**

unit = .S2

**Opcode**

31	29	28	27						23	22	21	20	19	18	17	16
				<i>creg</i>					<i>dst</i>	0	0	1	1	0	0	0
				3	1						5					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	x	0	0	0	0	1	1	1	0	0	0	s	p	
								1						1	1	

Opcodemap field used...	For operand type...	Unit
<i>src2</i>	xsint	.S2

**Description**    IRP is placed in the program fetch counter (PFC). This instruction also moves the PGIE bit value to the GIE bit. The PGIE bit is unchanged.

If two branches are in the same execute packet and are both taken, behavior is undefined.

Two conditional branches can be in the same execute packet if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a true condition, the code executes in a well-defined way.



**Note—**

- 1) This instruction executes on .S2 only. PFC is the program fetch counter.
- 2) Refer to Chapter 6 “[Interrupts](#)” on page 6-1 for more information on IRP, PGIE, and GIE.
- 3) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
- 4) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.

- 5) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

**Execution**

```
if (cond) IRP → PFC
else nop
```

**Pipeline**

Pipeline Stage	Target Instruction						
	E1	PS	PW	PR	DP	DC	E1
Read	IRP						
Written							
Branch taken							✓
Unit in use	.S2						

**Instruction Type** Branch

**Delay Slots** 5

**Example** Table 4-4 gives the program counter values and actions for the following code example. Given that an interrupt occurred at

```
PC = 0000 1000 IRP = 0000 1000
0000 0020B      .S2 IRP
0000 0024ADD    .S1 A0, A2, A1
0000 0028MPY    .M1 A1, A0, A1
0000 002CNOP
0000 0030SHR    .S1 A1, 15, A1
0000 0034ADD    .L1 A1, A2, A1
0000 0038ADD    .L2 B1, B2, B3
```

**Table 4-4 Program Counter Values for B IRP Instruction Example**

Cycle	Program Counter Value	Action
Cycle 0	0000 0020	Branch command executes (target code fetched)
Cycle 1	0000 0024	
Cycle 2	0000 0028	
Cycle 3	0000 002C	
Cycle 4	0000 0030	
Cycle 5	0000 0034	
Cycle 6	0000 1000	Branch target code executes

## 4.27 B NRP

Branch Using NMI Return Pointer

**Syntax**    **B (.unit) NRP**

unit = .S2

**Opcode**

31	29	28	27					23	22	21	20	19	18	17	16
				<i>creg</i>	<i>z</i>		<i>dst</i>		0	0	1	1	1	0	0
				3	1		5								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	x	0	0	0	0	1	1	1	0	0	0	s	p
								1						1	1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xsint	.S2

**Description**    NRP is placed in the program fetch counter (PFC). This instruction also sets the NMIE bit. The PGIE bit is unchanged.

If two branches are in the same execute packet and are both taken, behavior is undefined.

Two conditional branches can be in the same execute packet if one branch uses a displacement and the other uses a register, IRP, or NRP. As long as only one branch has a true condition, the code executes in a well-defined way.



**Note—**

- 1) This instruction executes on .S2 only. PFC is program fetch counter.
- 2) Refer to Chapter 6 “[Interrupts](#)” on page 6-1 for more information on NRP and NMIE.
- 3) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
- 4) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.

- 5) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

**Execution**

```
if (cond) NRP → PFC
else nop
```

**Pipeline**

Pipeline Stage	Target Instruction						
	E1	PS	PW	PR	DP	DC	E1
Read	NRP						
Written							
Branch taken							✓
Unit in use	.S2						

**Instruction Type** Branch

**Delay Slots** 5

**Example** Table 4-5 on page 4-65 gives the program counter values and actions for the following code example. Given that an interrupt occurred at

```
PC = 0000 1000 IRP = 0000 1000
0000 0020B      .S2 NRP
0000 0024ADD    .S1 A0, A2, A1
0000 0028MPY    .M1 A1, A0, A1
0000 002CNOP
0000 0030SHR    .S1 A1, 15, A1
0000 0034ADD    .L1 A1, A2, A1
0000 0038ADD    .L2 B1, B2, B3
```

**Table 4-5 Program Counter Values for B NRP Instruction Example**

Cycle	Program Counter Value	Action
Cycle 0	0000 0020	Branch command executes (target code fetched)
Cycle 1	0000 0024	
Cycle 2	0000 0028	
Cycle 3	0000 002C	
Cycle 4	0000 0030	
Cycle 5	0000 0034	
Cycle 6	0000 1000	Branch target code executes

## 4.28 BDEC

Branch and Decrement

**Syntax**    **BDEC (.unit) src, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27											23	22
<i>creg</i>	z			<i>dst</i>											
3	1			5										10	
<i>src</i>	1	0	0	0	0	0	0	0	1	0	0	0	s	p	

Opcode map field used...	For operand type...	Unit
<i>src</i>	scst10	.S1, .S2
<i>dst</i>	int	

**Description**

If the predication and decrement register (*dst*) is positive (greater than or equal to 0), the **BDEC** instruction performs a relative branch and decrements *dst* by 1. The instruction performs the relative branch using a 10-bit signed constant, *scst10*, in *src*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **BDEC** instruction (PCE1). The result is placed in the program fetch counter (PFC).

This instruction helps reduce the number of instructions needed to decrement a register and conditionally branch based upon the value of the register. Note also that any register can be used that can free the predicate registers (A0-A2 and B0-B2) for other uses.

The following code:

```

[!A1] SUB      CMPLT.L1    A10,0,A1
      .L1      A10,1,A10
|| [!A1] B       .S1      func
      NOP5

```

could be replaced by:

```

BDEC.S1 func, A10
NOP      5

```

 **Note—**

- 1) Only one **BDEC** instruction can be executed per cycle. The **BDEC** instruction can be predicated by using any conventional condition register. The conditions are effectively ANDed together. If two branches are in the same execute packet, and if both are taken, behavior is undefined.
- 2) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.

- 3) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.
- 4) The **BDEC** instruction cannot be in the same execute packet as an **ADDKPC** instruction.

**Execution**

```
if (cond) {
    if (dst >= 0), PFC = ((PCE1 + se(scst10)) << 2);
    if (dst >= 0), dst = dst - 1;
    else nop
}
else nop
```

**Pipeline**

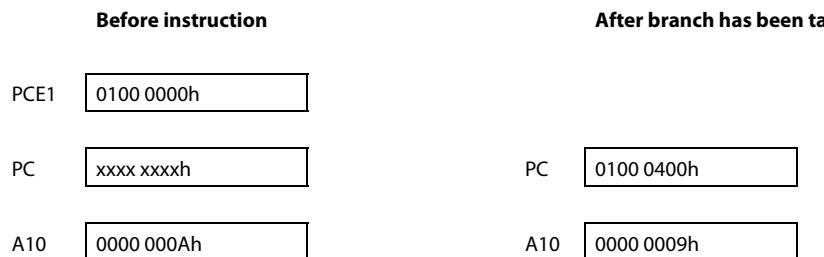
Pipeline Stage	Target Instruction					
	E1	PS	PW	PR	DP	DC
Read	<i>dst</i>					
Written		<i>dst, PC</i>				
Branch taken						✓
Unit in use	.S					

**Instruction Type** Branch

**Delay Slots** 5

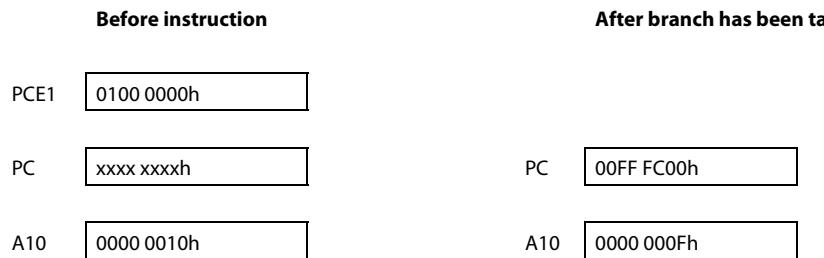
**Examples** **Example 1**

BDEC .S1 100h,A10



**Example 2**

BDEC .S1 300h,A10 ; 300h is sign extended



## 4.29 BITC4

Bit Count, Packed 8-Bit

**Syntax**    **BITC4 (.unit) src2, dst**

unit = .M1 or .M2

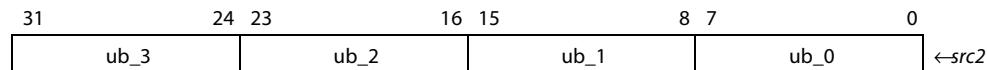
**Opcode**

31	29	28	27						23	22						18	17	16
				<i>dst</i>							<i>src2</i>					1	1	
3			1					5							5			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

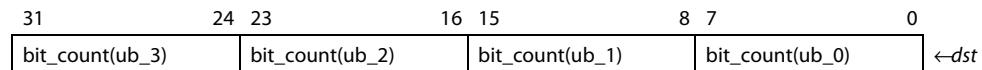
Opcode map field used...	For operand type...	Unit
<i>src2</i> <i>dst</i>	xu4 u4	.M1, .M2

**Description**

Performs a bit-count operation on 8-bit quantities. The value in *src2* is treated as packed 8-bit data, and the result is written in packed 8-bit format. For each of the 8-bit quantities in *src2*, the count of the number of 1 bits in that value is written to the corresponding position in *dst*.


**BITC4**

↓                    ↓                    ↓                    ↓



**Execution**

```

if (cond) {
    bit_count(src2(ubyte0)) → ubyte0(dst);
    bit_count(src2(ubyte1)) → ubyte1(dst);
    bit_count(src2(ubyte2)) → ubyte2(dst);
    bit_count(src2(ubyte3)) → ubyte3(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**Example** BITC4 .M1 A1,A2

**Before instruction**

A1 

9E 52 6E 30h
--------------

A2 

xxxx xxxxh
------------

**2 cycles after instruction**

A1 

9E 52 6E 30h
--------------

A2 

05 03 05 02h
--------------

## 4.30 BITR

Bit Reverse

**Syntax**    **BITR** (.unit) *src2*, *dst*

unit = .M1 or .M2

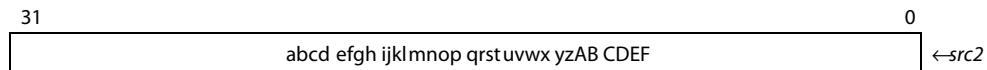
**Opcode**

31	29	28	27	dst						src2						18	17	16	
				z													1	1	
				creg	3	1	5						5						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	x	0	0	0	0	1	1	1	1	0	0	s	p	1	1		
					1														

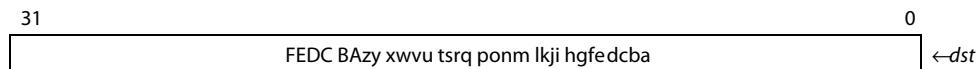
Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.M1, .M2
<i>dst</i>	uint	

**Description**

Implements a bit-reversal function that reverses the order of bits in a 32-bit word. This means that bit 0 of the source becomes bit 31 of the result, bit 1 of the source becomes bit 30 of the result, bit 2 becomes bit 29, and so on.



**BITR**



**Execution**

```
if (cond)bit_reverse(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type**   Two-cycle

**Delay Slots**   1

**Example**   BITR .M2 B4,B5

**Before instruction**B4 

A6E2 C179h
------------

B5 

xxxx xxxxh
------------

**2 cycles after instruction**B4 

A6E2 C179h
------------

B5 

9E83 4765h
------------

## 4.31 BNOP

Branch Using a Displacement With NOP

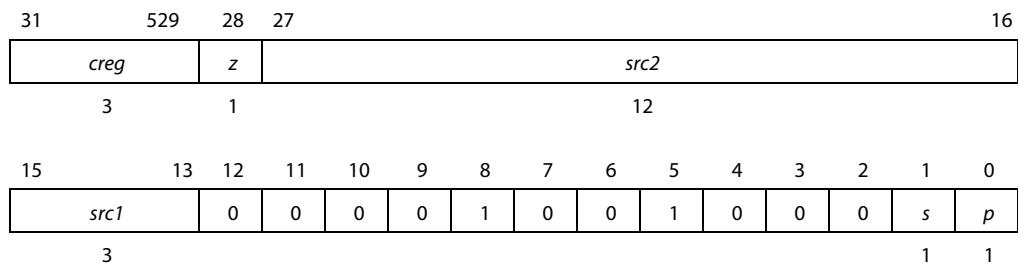
**Syntax** **BNOP (.unit) src2, src1**

unit = .S1, .S2, or none

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Sbs7	<a href="#">Figure F-12</a>
	Sbu8	<a href="#">Figure F-13</a>
	Sbs7c	<a href="#">Figure F-15</a>
	Sbu8c	<a href="#">Figure F-16</a>
	Sx1b	<a href="#">Figure F-27</a>

**Opcode**



Opcode map field used...	For operand type...	Unit
<b>src2</b>	<b>scst12</b>	<b>.S1, .S2</b>
<b>src1</b>	<b>ucst3</b>	

**Description**

The constant displacement form of the **BNOP** instruction performs a relative branch with **NOP** instructions. The instruction performs the relative branch using the 12-bit signed constant specified by *src2*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **BNOP** instruction (PCE1). The result is placed in the program fetch counter (PFC).

The 3-bit unsigned constant specified in *src1* gives the number of delay slot **NOP** instructions to be inserted, from 0 to 7. With *src1* = 0, no **NOP** cycles are inserted.

This instruction helps reduce the number of instructions to perform a branch when **NOP** instructions are required to fill the delay slots of a branch.

The following code:

```
B          .S1      LABEL
NOP          N
LABEL: ADD
```

could be replaced by:

```
BNOP  .S1      LABEL, N
LABEL: ADD
```

**Note—**

- 1) BNOP instructions may be predicated. The predication condition controls whether or not the branch is taken, but does not affect the insertion of NOPs. BNOP always inserts the number of NOPs specified by N, regardless of the predication condition.
- 2) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
- 3) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.
- 4) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

Only one branch instruction can be executed per cycle. If two branches are in the same execute packet, and if both are taken, the behavior is undefined. It should also be noted that when a predicated BNOP instruction is used with a NOP count greater than 5, the CPU inserts the full delay slots requested when the predicated condition is false.

For example, the following set of instructions will insert 7 cycles of NOPs:

```
ZERO    .L1   A0
[A0]BNOP  .S1 LABEL,7 ; branch is not taken and
; 7 cycles of NOPs are inserted
```

Conversely, when a predicated BNOP instruction is used with a NOP count greater than 5 and the predication condition is true, the branch will be taken and the multi-cycle NOP is terminated when the branch is taken.

For example in the following set of instructions, only 5 cycles of NOP are inserted:

```
MVK      .D1 1,A0
[A0]BNOP  .S1 LABEL,7 ; branch is taken and
; 5 cycles of NOPs are inserted
```

The BNOP instruction cannot be paired with any other multicycle NOP instruction in the same execute packet. Instructions that generate a multicycle NOP are: IDLE, ADDKPC, CALLP, and the multicycle NOP.

The BNOP instruction does not require the use of the .S unit. If no unit is specified, then it may be scheduled in parallel with instructions executing on both the .S1 and .S2 units. If either the .S1 or .S2 unit is specified for BNOP, then the .S unit specified is not available for another instruction in the same execute packet. This is enforced by the assembler.

It is possible to branch into the middle of a 32-bit instruction. The only case that will be detected and result in an exception is when the 32-bit instruction is contained in a compact header-based fetch packet. The header cannot be the target of a branch instruction. In the event that the header is the target of a branch, an exception will be raised.

**Execution (if instruction is within compact instruction fetch packet)**

```
if (cond) {
    PFC = (PCE1 + (se(scst12) << 1));
    nop (src1)
}
else nop (src1 + 1)

if (cond) {
    PFC = (PCE1 + (se(scst12) << 2));
    nop (src1)
}
else nop (src1 + 1)
```

**Execution (if instruction is not within compact instruction fetch packet)**

**Pipeline**

Pipeline Stage	Target Instruction						E1
	E1	PS	PW	PR	DP	DC	
Read	<i>src2</i>						
Written	PC						
Branch taken							✓
Unit in use	.S						

**Instruction Type** Branch

**Delay Slots** 5

**See Also** ADDKPC, B, NOP

**Example** BNOP .S1 30h,2

**Before instruction**

PCE1 0100 0500h

**After branch has been taken**

PC xxxx xxxxh PC 0100 1100h

## 4.32 BNOP

Branch Using a Register With NOP

**Syntax**    **BNOP (.unit) src2, src1**

unit = .S2

**Opcode**

31	29	28	27	26	25	24	23	22	18	17	16
<i>creg</i>	<i>z</i>	0	0	0	0	1		<i>src2</i>	0	0	
3	1							5			

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	0	1	1	0	1	1	0	0	0	1	<i>p</i>	
3	1												1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.S2
<i>src1</i>	ucst3	

**Description**

The register form of the **BNOP** instruction performs an absolute branch with **NOP** instructions. The register specified in *src2* is placed in the program fetch counter (PFC).

For branch targets residing in compact header-based fetch packets, the 31 most-significant bits of the register are used to determine the branch target. For branch targets not residing in compact header-based fetch packets, the 30 most-significant bits of the register are used to determine the branch target.

The 3-bit unsigned constant specified in *src1* gives the number of delay slots **NOP** instructions to be inserted, from 0 to 7. With *src1* = 0, no NOP cycles are inserted.

This instruction helps reduce the number of instructions to perform a branch when **NOP** instructions are required to fill the delay slots of a branch.

The following code:

```
B      .S2  B3
NOP      N
```

could be replaced by:

```
BNOP.S2 B3,N
```



**Note—**

- 1) BNOP instructions may be predicated. The predication condition controls whether or not the branch is taken, but does not affect the insertion of NOPs. BNOP always inserts the number of NOPs specified by N, regardless of the predication condition.
- 2) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.

- 3) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.
- 4) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

Only one branch instruction can be executed per cycle. If two branches are in the same execute packet, and if both are taken, the behavior is undefined. It should also be noted that when a predicated **BNOP** instruction is used with a **NOP** count greater than 5, the CPU inserts the full delay slots requested when the predicated condition is false.

For example, the following set of instructions will insert 7 cycles of **NOPs**:

```
ZERO .L1 A0
[A0] BNOP .S2 B3,7; branch is not taken and 7 cycles of NOPs are inserted
```

Conversely, when a predicated **BNOP** instruction is used with a **NOP** count greater than 5 and the predication condition is true, the branch will be taken and multi-cycle **NOP** is terminated when the branch is taken.

For example, in the following set of instructions only 5 cycles of **NOP** are inserted:

```
MVK .D1 1,A0
[A0] BNOP .S2 B3,7; branch is taken and 5 cycles of NOPs are inserted
```

The **BNOP** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **IDLE**, **ADDKPC**, **CALLP**, and the multicycle **NOP**.

**Execution**

```
if (cond) {
    src2 → PFC;
    nop (src1)
}
else nop (src1 + 1)
```

**Pipeline**

Pipeline Stage	Target Instruction					
	E1	PS	PW	PR	DP	DC
Read	src2					
Written	PC					
Branch taken						✓
Unit in use	.S2					

**Instruction Type**

Branch

**Delay Slots**

5

**See Also**

[ADDKPC](#), [B](#), [NOP](#)

**Example**

BNOP .S2 A5,2

**Before instruction**

PCE1 0010 0000h

**After branch has been taken**

PC xxxx xxxxh

PC 0100 F000h

A5

0100 F000h

A5

0100 F000h

## 4.33 BPOS

Branch Positive

**Syntax**    **BPOS (.unit) src, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27						23	22		
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src</i>	
3	1			5							10	
<i>src</i>	0	0	0	0	0	0	0	1	0	0	0	<i>s</i> <i>p</i>

Opcode map field used...	For operand type...	Unit
<i>src</i> <i>dst</i>	scst10 int	.S1, .S2

**Description**

If the predication register (*dst*) is positive (greater than or equal to 0), the **BPOS** instruction performs a relative branch. If *dst* is negative, the **BPOS** instruction takes no other action.

The instruction performs the relative branch using a 10-bit signed constant, *scst10*, in *src*. The constant is shifted 2 bits to the left, then added to the address of the first instruction of the fetch packet that contains the **BDEC** instruction (PCE1). The result is placed in the program fetch counter (PFC).

Any register can be used that can free the predicate registers (A0-A2 and B0-B2) for other uses.



**Note—**

- 1) Only one **BPOS** instruction can be executed per cycle. The **BPOS** instruction can be predicated by using any conventional condition register. The conditions are effectively ANDed together. If two branches are in the same execute packet, and if both are taken, behavior is undefined.
- 2) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.
- 3) See “[Branching Into the Middle of an Execute Packet](#)” on page 3-13 for information on branching into the middle of an execute packet.
- 4) A branch to an execute packet that spans two fetch packets will cause a stall while the second fetch packet is fetched.

- 5) The **BPOS** instruction cannot be in the same execute packet as an **ADDKPC** instruction.

**Execution**

```
if (cond) {
    if (dst >= 0), PFC = (PCE1 + (se(scst10) << 2));
    else nop
}
else nop
```

**Pipeline**

Pipeline Stage	Target Instruction						E1
	E1	PS	PW	PR	DP	DC	
Read	<i>dst</i>						
Written	PC						
Branch taken							✓
Unit in use	.S						

**Instruction Type** Branch

**Delay Slots** 5

**Example** BPOS .S1 200h,A10

**Before instruction**

PCE1	0010 0000h
------	------------

PC	xxxx xxxxh
----	------------

A10	0000 000Ah
-----	------------

**After branch has been taken**

PC	0100 0800h
----	------------

A10	0000 000Ah
-----	------------

## 4.34 CALLP

### Call Using a Displacement

**Syntax**    **CALLP** (.unit) label, A3/B3

unit = .S1 or .S2

## ***Compact Instruction Format***

<b>Unit</b>	<b>Opcode Format</b>	<b>Figure</b>
.S	Scs10	<a href="#">Figure F-14</a>

## *Opcode*

31	30	29	28	27	
0	0	0	1		cst21
21					

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
cst21	scst21	.S1,.S2

### **Description**

A 21-bit signed constant, *cst21*, is shifted left by 2 bits and is added to the address of the first instruction of the fetch packet that contains the branch instruction. The result is placed in the program fetch counter (PFC). The assembler/linker automatically computes the correct value for *cst21* by the following formula:

*cst21* = (label - PCE1) >> 2

The address of the execute packet immediately following the execute packet containing the **CALLP** instruction is placed in A3, if the S1 unit is used; or in B3, if the S2 unit is used. This write occurs in E1. An implied **NOP 5** is inserted into the instruction pipeline occupying E2-E6.

Since this branch is taken unconditionally, it cannot be placed in the same execute packet as another branch. Additionally, no other branches should be pending when the **CALLP** instruction is executed.

**CALLP**, like other relative branch instructions, cannot have an **ADDKPC** instruction in the same execute packet with it.



**Note—**

- 1) PCE1 (program counter) represents the address of the first instruction in the fetch packet in the E1 stage of the pipeline. PFC is the program fetch counter. retPC represents the address of the first instruction of the execute packet in the DC stage of the pipeline.

- 2) The execute packets in the delay slots of a branch cannot be interrupted. This is true regardless of whether the branch is taken.

**Execution**

```
(cst21 << 2) + PCE1 → PFC
if (unit = S2), retPC → B3
else if (unit = S1), retPC → A3
nop 5
```

**Pipeline**

Pipeline Stage	Target Instruction					
	E1	PS	PW	PR	DP	DC
Read						
Written	A3/B3					
Branch taken						✓
Unit in use	.S					

**Instruction Type** Branch

**Delay Slots** 5

## 4.35 CCMATMPY

Complex Conjugate Matrix Multiply, Signed Complex 16-bit (16-bit real/16-bit Imaginary)

**Syntax**    CCMATMPY (.unit) .L1 or .L2 *src1, src2, dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	0	0	0	0	0	s	p	

5                        5                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,qwop2,qwdst	.M1 or .M2	00101

**Description**    This instruction performs multiply of conjugate of 1x2 complex vector by a 2x2 complex matrix giving two 64-bit complex results. The input format is a 32-bit complex number i.e. two 16-bit numbers packed together. The high 16-bits are the real part, and the low 16-bits are the imaginary part.

The output of the instruction is a 64-bit complex result having 32-bits for the real and 32-bits for the imaginary part. The real part goes in the upper 32-bits of the register pair, and the imaginary part is in the low 32-bits.

For example, CCMATMPY A7:A6,A3:A2:A1:A0,A11:A10:A9:A8 will perform as following:

$$[A_{11}:A_{10} \ A_{9}:A_8] = [\overline{A_7} \ \overline{A_6}] * \begin{bmatrix} A_3 & A_2 \\ A_1 & A_0 \end{bmatrix}$$

-or-

$$\begin{aligned} A_{11}:A_{10} &= \text{conj}(\overline{A_7}) * A_3 + \text{conj}(\overline{A_6}) * A_1 \\ A_{9}:A_8 &= \text{conj}(\overline{A_7}) * A_2 + \text{conj}(\overline{A_6}) * A_0 \end{aligned}$$

The CCMATMPY instruction is roughly equivalent to following instruction sequence:

```
DAPYS2 A7:A6, A19:A18,A5:A4
Where A19:A18 contains "0x0000FFFF:0000FFFF"
CMPY  A3,A5,A31:A30
CMPY  A1,A4,A29:A28
CMPY  A2,A5,A27:A26
CMPY  A0,A4,A25:A24
NOP
DSADD A31:A30,A29:A28,A11:A10
DSADD A27:A26,A25:A24,A9:A10
```

The main difference between executing CCMATMPY and the above sequence is that saturation is only performed once at the end and intermediate precision is kept at 34 bits

**Execution**

```
((msb16(src1_e) x lsb16(src2_0))-(lsb16(src1_e) x msb16(src2_0)))-> tmp0_e
((msb16(src1_e) x msb16(src2_0))+ (lsb16(src1_e) x lsb16(src2_0)))-> tmp0_o
((msb16(src1_o) x lsb16(src2_2))-(lsb16(src1_o) x msb16(src2_2)))-> tmp1_e
((msb16(src1_o) x msb16(src2_2))+ (lsb16(src1_o) x lsb16(src2_2)))-> tmp1_o
((msb16(src1_e) x lsb16(src2_1))-(lsb16(src1_e) x msb16(src2_1)))-> tmp2_e
```

```
((msb16(src1_e) x msb16(src2_1))+(lsb16(src1_e) x lsb16(src2_1)))-> tmp2_o
((msb16(src1_o) x lsb16(src2_3))-(lsb16(src1_o) x msb16(src2_3)))-> tmp3_e
((msb16(src1_o) x msb16(src2_3))+(lsb16(src1_o) x lsb16(src2_3)))-> tmp3_o
sat(tmp0_e + tmp1_e)->dst_0
sat(tmp0_o + tmp1_o)->dst_1
sat(tmp2_e + tmp3_e)->dst_2
sat(tmp2_o + tmp3_o)->dst_3
```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [CMATMPYR1](#), [CCMATMPYR1](#), [CMATMPY](#)

**Example**

```
CSR == 0x10010000 ; 4
A7 == 0x12345678
A6 == 0x12345678
A3 == 0x12345678
A2 == 0x12345678
A1 == 0x12345678
A0 == 0x12345678
CCMATMPY .M A7:A6,A3:A2:A1:A0,A11:A10:A9:A8
A11 == 0x3d0065a0
A10 == 0x00000000
A9 == 0x3d0065a0
A8 == 0x00000000

CSR= 0x10010000

CSR == 0x10010000 ; 4
A7 == 0x80008000
A6 == 0x80008000
A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80008000
A0 == 0x80008000
CCMATMPY .M A7:A6,A3:A2:A1:A0,A11:A10:A9:A8
A11 == 0x7fffffff
A10 == 0x00000000
A9 == 0x7fffffff
A8 == 0x00000000

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0x7FFF7FFF
B6 == 0x7FFF8000
B3 == 0x7FFF7FFF
B2 == 0x7FFF8000
B1 == 0x7FFF8000
B0 == 0x7FFF7FFF
CCMATMPY .M B7:B6,B3:B2:B1:B0,B11:B10:B9:B8
B11 == 0x7fffffff
B10 == 0x00000000
B9 == 0xfffff002
B8 == 0x00000000

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0xFFFFFFFF
B6 == 0xFFFFFFFF
B3 == 0xFFFFFFFF
B2 == 0xFFFFFFFF
B1 == 0xFFFFFFFF
B0 == 0xFFFFFFFF
CCMATMPY .M B7:B6,B3:B2:B1:B0,B11:B10:B9:B8
B11 == 0x00000004
B10 == 0x00000000
B9 == 0x00000004
B8 == 0x00000000
```

## 4.35 CCMATMPY

## Chapter 4—Instruction Descriptions

```
CSR == 0x10010000i
CSR == 0x10010000 ; 4
```

## 4.36 CCMATMPYR1

Complex Conjugate Matrix Multiply With Rounding, Signed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    CCMATMPYR1 (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	0	0	0	0	0	s	p	

5                        5                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,qwop2,dwdst	.M1 or .M2	00111

**Description**    This instruction performs multiply of conjugate of a 1x2 complex vector by a 2x2 complex matrix with rounding giving two 32-bit complex results. For example, CCMATMPYR1 A7:A6,A3:A2:A1:A0,A9:A8 will perform as following:

$$[A_9 \ A_8] = [\overline{A_7} \ \overline{A_6}] * \begin{bmatrix} A_3 & A_2 \\ A_1 & A_0 \end{bmatrix}$$

-or-

$$\begin{aligned} A_9 &= \text{conj}(\overline{A_7}) * A_3 + \text{conj}(\overline{A_6}) * A_1 \\ A_8 &= \text{conj}(\overline{A_7}) * A_2 + \text{conj}(\overline{A_6}) * A_0 \end{aligned}$$

The CCMATMPYR1 instruction is roughly equivalent to following instruction sequence:

```

DCONJ A7,A6,A5:A4
CMPYR1 A3,A5,A31
CMPYR1 A1,A4,A29
CMPYR1 A2,A5,A30
CMPYR1 A0,A4,A28
NOP
DSADD A31:A30,A29:A28,A9:A8

```

The difference between executing CCMATMPYR1 and the above sequence is that saturation is only performed once at the end and intermediate precision is kept at 34 bits

**Execution**

```

((msb16(src1_e) x lsb16(src2_0)) - (lsb16(src1_e) x msb16(src2_0))) -> tmp0_e
((msb16(src1_e) x msb16(src2_0)) + (lsb16(src1_e) x lsb16(src2_0))) -> tmp0_o
((msb16(src1_o) x lsb16(src2_2)) - (lsb16(src1_o) x msb16(src2_2))) -> tmp1_e
((msb16(src1_o) x msb16(src2_2)) + (lsb16(src1_o) x lsb16(src2_2))) -> tmp1_o
((msb16(src1_e) x lsb16(src2_1)) - (lsb16(src1_e) x msb16(src2_1))) -> tmp2_e
((msb16(src1_e) x msb16(src2_1)) + (lsb16(src1_e) x lsb16(src2_1))) -> tmp2_o
((msb16(src1_o) x lsb16(src2_3)) - (lsb16(src1_o) x msb16(src2_3))) -> tmp3_e
((msb16(src1_o) x msb16(src2_3)) + (lsb16(src1_o) x lsb16(src2_3))) -> tmp3_o
msb16(sat((tmp0_e + tmp1_e) + 00004000h) << 1)) -> lsb16(dst_e)

```

```
msbl16(sat(((tmp0_o + tmp1_o) + 00004000h) << 1)) -> msbl16(dst_e)
msbl16(sat(((tmp2_e + tmp3_e) + 00004000h) << 1)) -> lsbl16(dst_o)
msbl16(sat(((tmp2_o + tmp3_o) + 00004000h) << 1)) -> msbl16(dst_o)
```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [CCMATMPY](#), [CMATMPYR1](#), [CMATMPY](#)

**Example**

```
CSR == 0x10010000 ; 4
A7 == 0x12345678
A6 == 0x12345678
A3 == 0x12345678
A2 == 0x12345678
A1 == 0x12345678
A0 == 0x12345678
CCMATMPYR1 .M A7:A6,A3:A2:A1:A0,A11:A10
A11 == 0x7a010000
A10 == 0x7a010000

CSR= 0x10010000

CSR == 0x10010000 ; 4
A7 == 0x80008000
A6 == 0x80008000
A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80008000
A0 == 0x80008000
CCMATMPYR1 .M A7:A6,A3:A2:A1:A0,A11:A10
A11 == 0x7fff0000
A10 == 0x7fff0000

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0x7FFF7FFF
B6 == 0x7FFF8000
B3 == 0x7FFF7FFF
B2 == 0x7FFF8000
B1 == 0x7FFF8000
B0 == 0x7FFF7FFF
CCMATMPYR1 .M B7:B6,B3:B2:B1:B0,B11:B10:
B11 == 0x7fff0000
B10: == 0xffffe0000

CSR == 0x10010200

CSR == 0x10010000 ; 4
B7 == 0xFFFFFFF
B6 == 0xFFFFFFF
B3 == 0xFFFFFFF
B2 == 0xFFFFFFF
B1 == 0xFFFFFFF
B0 == 0xFFFFFFF
CCMATMPYR1 .M B7:B6,B3:B2:B1:B0,B11:B10:
B11 == 0x00000000
B10: == 0x00000000

CSR == 0x10010000

CSR == 0x10010000 ; 4
```

4.37 CCMPY32R1

## Complex Multiply With Rounding and Conjugate, Signed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    **CCMPY32R1** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcodes**      Opcode for .M Unit, Compound Results, new opcode space

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	01001

**Description** The CCMPY32R1 instruction performs one complex multiply between the 64-bit complex number in src1 and the complex conjugate of the complex 64-bit number in src2. Each 64-bit complex number is contained in a register pair. The Odd register in the pair (the most significant word) represents the real component of the complex number as a 32-bit signed quantity. The Even register in the pair represents the imaginary component of the complex number. The saturation condition of  $0x80000000 * 0x80000000 + 0x80000000 * 0x80000000$  is taken into account, yielding a result of "7FFFFFFF:00000000.

After multiplying and adding the 32-bit numbers together, they are shifted right by 31 and rounded. Intermediate results are calculated at 64-bits.

The CCMPY32R1 instruction is functionally equivalent to following C sequence:

```

long long      src1_real = dwop1.high ; // Cast 32-bit signed integer to 64-bits
long long      src1_im   = dwop1.low ;
long long      src2_real = xdwp2.high ; // Cast 32-bit signed integer to 64-bits
long long      src2_im   = xdwp2.low ;
long long      tmp_real, tmp_im ;

if ( src1_real == 0x80000000 &&
    src1_im   == 0x80000000 &&
    src2_real == 0x80000000 &&
    src2_im   == 0x80000000 )
    tmp_real = 0x7fffffff ;
else
    tmp_real = (src1_real * src2_real + src1_im * src2_im + (1<<30) ) >> 31 ;

tmp_im = (src1_real * -src2_im + src1_im * src2_real + (1<<30) ) >> 31 ;

```

```
    dwdst.high = tmp_real ;
    dwdst.low = tmp_im ;
```

**Execution**       $\text{sat}(\text{src1\_o} \times \text{src2\_e} - \text{src1\_e} \times \text{src2\_o} + (1 << 30)) >> 31 \rightarrow \text{dst\_e}$   
 $\text{src1\_o} \times \text{src2\_o} + \text{src1\_e} \times \text{src2\_e} + (1 << 30)) >> 31 \rightarrow \text{dst\_o}$

**Instruction Type**    4-cycle

**Delay Slots**       3

**Functional Unit Latency**    1

**See Also**            [CMPY](#), [SMPY32](#), [CCMPY32R1](#)

**Example**

```
CSR == 0x00000000 ; 4
A3 == 0x00000000
A2 == 0x00000000
A1 == 0x00000000
A0 == 0x00000000
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x00000000
A2 == 0x08000400
A1 == 0x00000000
A0 == 0x09000200
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00900068
A14 == 0x00000000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x80000000
A2 == 0x80000000
A1 == 0x80000000
A0 == 0x80000000
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x7FFFFFFF
A14 == 0x00000000

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0x7FFF7FFF
A2 == 0x7FFF8000
A1 == 0x7FFF8000
A0 == 0x7FFF7FFF
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x7fffffff
A14 == 0x00000002

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0xFFFFFFF
A2 == 0xFFFFFFF
A1 == 0xFFFFFFF
A0 == 0xFFFFFFF
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x55555555
A2 == 0x55555555
A1 == 0x55555555
A0 == 0x55555555
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x71c71c71
```

```

A14 == 0x00000000
CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x01234567
A2 == 0x89ABCDEF
A1 == 0x89ABCDEF
A0 == 0x01234567
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0xfde578db
A14 == 0x6D60DCE3

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x80000000
A2 == 0x7ffff7fff
A1 == 0x7ffff7fff
A0 == 0x7fffffff
CCMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0xffffffff
A14 == 0x7fffffff

CSR= 0x00000200 ; 4
CSR == 0x00000000 ; 4

```

## 4.38 CLR

Clear a Bit Field

**Syntax**    **CLR (.unit) src2, csta, cstb, dst**

or

**CLR (.unit) src2, src1, dst**

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Sc5	<a href="#">Figure F-22</a>

**Opcode** Constant form

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>csta</i>
3	1		5		5		5
13	12		8	7	6	5	4
<i>csta</i>		<i>cstb</i>	1	1	0	0	1
5		5					1
2	1	0					0
							1
							1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	uint	.S1, .S2
<i>csta</i>	ucst5	
<i>cstb</i>	ucst5	
<i>dst</i>	uint	

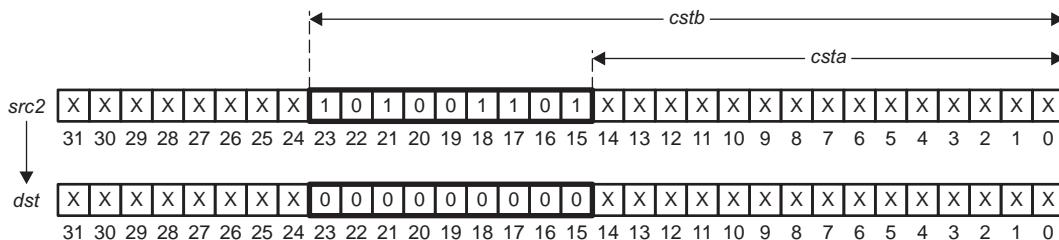
**Opcode** Register form

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
13	12	11	10	9	8	7	6
<i>src1</i>	x	1	1	1	1	1	1
5	1						
5	4	3	2	1	0	0	0
							s
							p
							1
							1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.S1, .S2
<i>src1</i>	uint	
<i>dst</i>	uint	

**Description** For  $cstb > csta$ , the field in *src2* as specified by *csta* to *cstb* is cleared to all 0s in *dst*. The *csta* and *cstb* operands may be specified as constants or in the 10 LSBs of the *src1* register, with *cstb* being bits 0–4 ( $src1_{4..0}$ ) and *csta* being bits 5–9 ( $src1_{9..5}$ ). *csta* is the LSB of the field and *cstb* is the MSB of the field. In other words, *csta* and *cstb* represent the beginning and ending bits, respectively, of the field to be cleared to all 0s in *dst*. The LSB location of *src2* is bit 0 and the MSB location of *src2* is bit 31.

In the following example, *csta* is 15 and *cstb* is 23. For the register version of the instruction, only the 10 LSBs of the *src1* register are valid. If any of the 22 MSBs are non-zero, the result is invalid.



For  $cstb < csta$ , the *src2* register is copied to *dst*. The *csta* and *cstb* operands may be specified as constants or in the 10 LSBs of the *src1* register, with *cstb* being bits 0–4 ( $src1_{4..0}$ ) and *csta* being bits 5–9 ( $src1_{9..5}$ ).

**Execution** If the constant form is used when  $cstb > csta$ :

```
if (cond) src2 clear csta, cstb → dst
else nop
```

If the register form is used when  $cstb > csta$ :

```
if (cond) src2 clear src19..5, src14..0 → dst
else nop
```

#### Pipeline

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SET](#)

**Examples** [Example 1](#)

```
CLR .S1      A1,4,19,A2
```

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A1	07A43F2Ah	A1	07A43F2Ah
A2	xxxxxxxxh	A2	07A0000Ah

### Example 2

CLR .S2      B1,B3,B2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B1	03B6E7D5h	B1	03B6E7D5h
B2	xxxxxxxxh	B2	03B0 0001h
B3	0000 0052h	B3	0000 0052h

## 4.39 CMATMPY

Complex Matrix Multiply, Signed Complex 16-bit (16-bit real/16-bit Imaginary)

**Syntax**    **CMATMPY** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	0	0	0	0	0	s	p	

5                        5                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,qwop2,qwdst	.M1 or .M2	00100

**Description**    This instruction performs multiply of 1x2 complex vector by a 2x2 complex matrix giving two 64-bit complex results. The input format is a 32-bit complex number i.e. 2 16-bit numbers packed together. The high 16-bits are the real part, and the low 16-bits are the imaginary part. The output of the instruction is 64-bit complex results having 32-bits for the real and 32-bits for the imaginary part. The real part goes in the upper 32-bits of the register pair, and the imaginary part is in the low 32-bits.

For example, CMATMPY A7:A6,A3:A2:A1:A0,A11:A10:A9:A8 will perform as following:

$$[A11:A10 \ A9:A8] = [A7 \ A6] * \begin{bmatrix} A3 & A2 \\ A1 & A0 \end{bmatrix}$$

-or-

$$\begin{aligned} A1:A10 &= A7*A3 + A6*A1 \\ A9:A8 &= A7*A2 + A6*A0 \end{aligned}$$

The CMATMPY instruction is roughly equivalent to following instruction sequence:

```
CMPY  A3,A7,A31:A30
CMPY  A1,A6,A29:A28
CMPY  A2,A7,A27:A26
CMPY  A0,A6,A25:A24
NOP
DSADD A31:A30,A29:A28,A11:A10
DSADD A27:A26,A25:A24,A9:A8
```

The difference between executing CMATMPY and the above sequence is that saturation is only performed once at the end and intermediate precision is kept at 34 bits

**Execution**

```
((msb16(src1_e) x lsb16(src2_0))+(lsb16(src1_e) x msb16(src2_0)))-> tmp0_e
((msb16(src1_e) x msb16(src2_0))-(lsb16(src1_e) x lsb16(src2_0)))-> tmp0_o
((msb16(src1_o) x lsb16(src2_2))+(lsb16(src1_o) x msb16(src2_2)))-> tmp1_e
((msb16(src1_o) x msb16(src2_2))-(lsb16(src1_o) x lsb16(src2_2)))-> tmp1_o
((msb16(src1_e) x lsb16(src2_1))+(lsb16(src1_e) x msb16(src2_1)))-> tmp2_e
((msb16(src1_e) x msb16(src2_1))-(lsb16(src1_e) x lsb16(src2_1)))-> tmp2_o
((msb16(src1_o) x lsb16(src2_3))+(lsb16(src1_o) x msb16(src2_3)))-> tmp3_e
((msb16(src1_o) x msb16(src2_3))-(lsb16(src1_o) x lsb16(src2_3)))-> tmp3_o
```

```

sat(tmp0_e + tmp1_e)->dst_0
sat(tmp0_o + tmp1_o)->dst_1
sat(tmp2_e + tmp3_e)->dst_2
sat(tmp2_o + tmp3_o)->dst_3

```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [CCMATMPYR1](#), [CMATMPYR1](#), [CMATMPY](#)

**Example**

```

CSR == 0x10010000 ; 4
A7 == 0x12345678
A6 == 0x12345678
A3 == 0x12345678
A2 == 0x12345678
A1 == 0x12345678
A0 == 0x12345678
CMATMPY .M A7:A6,A3:A2:A1:A0,A11:A10:A9:A8
A11 == 0xc82d04a0
A10 == 0x18980180
A9 == 0xc82d04a0
A8 == 0x18980180

CSR= 0x10010000

CSR == 0x10010000 ; 4
A7 == 0x80008000
A6 == 0x80008000
A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80008000
A0 == 0x80008000
CMATMPY .M A7:A6,A3:A2:A1:A0,A11:A10:A9:A8
A11 == 0x00000000
A10 == 0x7fffffff
A9 == 0x00000000
A8 == 0x7fffffff

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0x7FFF7FFF
B6 == 0x7FFF8000
B3 == 0x7FFF7FFF
B2 == 0x7FFF8000
B1 == 0x7FFF8000
B0 == 0x7FFF7FFF
CMATMPY .M B7:B6,B3:B2:B1:B0,B11:B10:B9:B8
B11 == 0xfffff0001
B10 == 0xfffff0002
B9 == 0x7fffffff
B8 == 0xfffff0002

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0x01234567
B6 == 0x89ABCDEF
B3 == 0x01234567
B2 == 0x89ABCDEF
B1 == 0x89ABCDEF
B0 == 0x01234567
CMATMPY .M B7:B6,B3:B2:B1:B0,B11:B10:B9:B8
B11 == 0x1a186e70
B10 == 0x2ee6b374
B9 == 0x1a186e70
B8 == 0xbf6522f4

CSR= 0x10010000

CSR == 0x10010000 ; 4

```

## 4.40 CMATMPYR1

Complex Matrix Multiply With Rounding, Signed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    **CMATMPYR1** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	0	0	0	0	0	s	p	

5                        5                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,qwop2,dwdst	.M1 or .M2	00110

**Description**    This instruction performs multiply of a 1x2 complex vector by a 2x2 complex matrix with rounding giving two 32-bit complex results. For example, CMATMPYR1 A7:A6,A3:A2:A1:A0,A9:A8 will perform as following:

$$[A_9 \ A_8] = [A_7 \ A_6] * \begin{bmatrix} A_3 & A_2 \\ A_1 & A_0 \end{bmatrix}$$

-or-

$$\begin{aligned} A_9 &= A_7 * A_3 + A_6 * A_1 \\ A_8 &= A_7 * A_2 + A_6 * A_0 \end{aligned}$$

The CMATMPYR1 instruction is roughly equivalent to following instruction sequence:

```
CMPYR1 A3,A7,A31
CMPYR1 A1,A6,A29
CMPYR1 A2,A7,A30
CMPYR1 A0,A6,A28
NOP
DSADD A31:A30,A29:A28,A9:A8
```

The difference between executing CMATMPYR1 and the above sequence is that saturation is only performed once at the end and intermediate precision is kept at 34 bits

**Execution**

```
((msb16(src1_e) x lsb16(src2_0))+(lsb16(src1_e) x msb16(src2_0)))-> tmp0_e
((msb16(src1_e) x msb16(src2_0))-(lsb16(src1_e) x lsb16(src2_0)))-> tmp0_o
((msb16(src1_o) x lsb16(src2_2))+(lsb16(src1_o) x msb16(src2_2)))-> tmp1_e
((msb16(src1_o) x msb16(src2_2))-(lsb16(src1_o) x lsb16(src2_2)))-> tmp1_o
((msb16(src1_e) x lsb16(src2_1))+(lsb16(src1_e) x msb16(src2_1)))-> tmp2_e
((msb16(src1_e) x msb16(src2_1))-(lsb16(src1_e) x lsb16(src2_1)))-> tmp2_o
((msb16(src1_o) x lsb16(src2_3))+(lsb16(src1_o) x msb16(src2_3)))-> tmp3_e
((msb16(src1_o) x msb16(src2_3))-(lsb16(src1_o) x lsb16(src2_3)))-> tmp3_o
msb16(sat(((tmp0_e + tmp1_e) + 00004000h) << 1)) -> lsb16(dst_e)
```

```
msbl16(sat(((tmp0_o + tmp1_o) + 00004000h) << 1)) -> msbl16(dst_e)
msbl16(sat(((tmp2_e + tmp3_e) + 00004000h) << 1)) -> lsbl16(dst_o)
msbl16(sat(((tmp2_o + tmp3_o) + 00004000h) << 1)) -> msbl16(dst_o)
```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [CCMATMPY](#), [CCMATMPYR1](#), [CMATMPY](#)

**Example**

```
CSR == 0x10010000 ; 4
A7 == 0x12345678
A6 == 0x12345678
A3 == 0x12345678
A2 == 0x12345678
A1 == 0x12345678
A0 == 0x12345678
CMATMPYR1 .M A7:A6,A3:A2:A1:A0,A11:A10:
A11 == 0x905a3130
A10: == 0x905a3130

CSR= 0x10010000

CSR == 0x10010000 ; 4
A7 == 0x80008000
A6 == 0x80008000
A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80008000
A0 == 0x80008000
CMATMPYR1 .M A7:A6,A3:A2:A1:A0,A11:A10:
A11 == 0x00007fff
A10: == 0x00007fff

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0x7FFF7FFF
B6 == 0x7FFF8000
B3 == 0x7FFF7FFF
B2 == 0x7FFF8000
B1 == 0x7FFF8000
B0 == 0x7FFF7FFF
CMATMPYR1 .M B7:B6,B3:B2:B1:B0,B11:B10:
B11 == 0xfffffeeffe
B10: == 0x7ffffffe

CSR= 0x10010200

CSR == 0x10010000 ; 4
B7 == 0xFFFFFFFF
B6 == 0xFFFFFFFF
B3 == 0xFFFFFFFF
B2 == 0xFFFFFFFF
B1 == 0xFFFFFFFF
B0 == 0xFFFFFFFF
CMATMPYR1 .M B7:B6,B3:B2:B1:B0,B11:B10:
B11 == 0x00000000
B10: == 0x00000000

CSR= 0x10010000

CSR == 0x10010000 ; 4
```

## 4.41 CMPEQ

Compare for Equality, Signed Integer

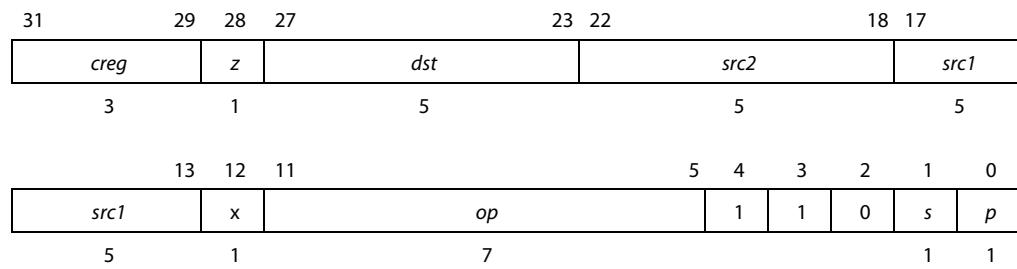
**Syntax** **CMPEQ (.unit) src1, src2, dst**

unit = .L1 or .L2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L2c Lx3c	<a href="#">Figure D-7</a> <a href="#">Figure D-9</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i> <i>src2</i> <i>dst</i>	sint xsint uint	.L1, .L2	1010011
<i>src1</i> <i>src2</i> <i>dst</i>	scst5 xsint uint	.L1, .L2	1010010
<i>src1</i> <i>src2</i> <i>dst</i>	xsint slong uint	.L1, .L2	1010001
<i>src1</i> <i>src2</i> <i>dst</i>	scst5 slong uint	.L1, .L2	1010000

**Description** Compares *src1* to *src2*. If *src1* equals *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

**Execution**

```
if (cond) {
    if (src1 == src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

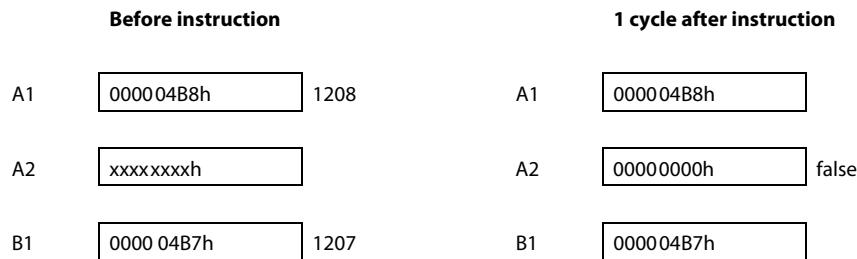
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ2](#), [CMPEQ4](#)

**Examples** [Example 1](#)

CMPEQ .L1X A1,B1,A2



**Example 2**

CMPEQ .L1 Ch,A1,A2

**Before instruction**
**1 cycle after instruction**

A1	<table border="1"><tr><td>0000000Ch</td></tr></table>	0000000Ch	12	A1	<table border="1"><tr><td>0000000Ch</td></tr></table>	0000000Ch
0000000Ch						
0000000Ch						
A2	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A2	<table border="1"><tr><td>00000001h</td></tr></table>	00000001h
xxxxxxxxh						
00000001h						

**Example 3**

CMPEQ .L2X A1,B3:B2,B1

**Before instruction**
**1 cycle after instruction**

A1	<table border="1"><tr><td>F23A3789h</td></tr></table>	F23A3789h	A1	<table border="1"><tr><td>F23A3789h</td></tr></table>	F23A3789h		
F23A3789h							
F23A3789h							
B1	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh	B1	<table border="1"><tr><td>00000001h</td></tr></table>	00000001h		
xxxxxxxxh							
00000001h							
B3:B2	<table border="1"><tr><td>0000 00FFh</td><td>F23A3789h</td></tr></table>	0000 00FFh	F23A3789h	B3:B2	<table border="1"><tr><td>00000FFh</td><td>F23A3789h</td></tr></table>	00000FFh	F23A3789h
0000 00FFh	F23A3789h						
00000FFh	F23A3789h						

## 4.42 CMPEQ2

Compare for Equality, Packed 16-Bit

**Syntax** **CMPEQ2** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

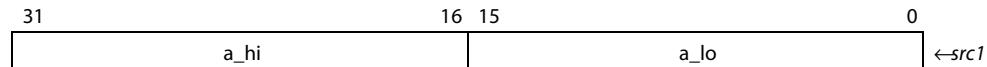
**Opcode**

31	29	28	27	23 22					18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>					<i>src1</i>	
3	1	5					5					5	
src1	x	0	1	1	1	0	1	1	0	0	0	s	p

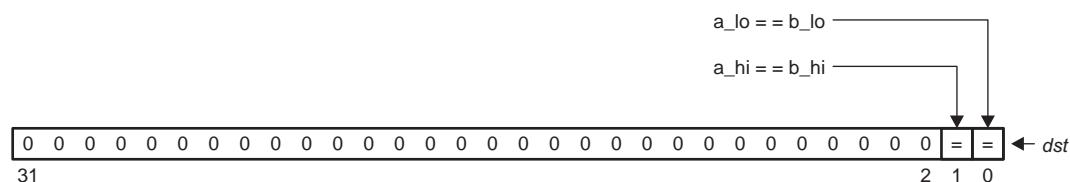
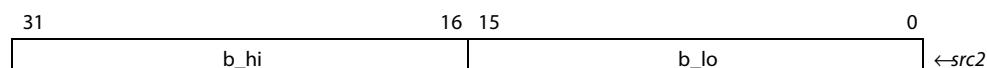
Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.S1, .S2
<i>src2</i>	<i>xs2</i>	
<i>dst</i>	<i>bv2</i>	

**Description**

Performs equality comparisons on packed 16-bit data. Each 16-bit value in *src1* is compared against the corresponding 16-bit value in *src2*, returning either a 1 if equal or a 0 if not equal. The equality results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are cleared to 0.



**CMPEQ2**



**Execution**

```

if (cond){
    if (lsb16(src1) == lsb16(src2)), 1 → dst0 else 0 → dst0;
    if (msb16(src1) == msb16(src2)), 1 → dst1 else 0 → dst1
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	src <sub>1</sub> , src <sub>2</sub>
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ](#), [CMPEQ4](#), [CMPGT2](#), [XPND2](#)

**Examples** [Example 1](#)

CMPEQ2 .S1 A3,A4,A5

Before instruction		1 cycle after instruction	
A3	1105 6E30h	A3	1105 6E30h
A4	1105 6980h	A4	1105 6980h
A5	xxxx xxxxh	A5	0000 0002h true, false

**Example 2**

CMPEQ2 .S2 B2,B8,B15

Before instruction		1 cycle after instruction	
B2	F23A 3789h	B2	F23A 3789h
B8	04B8 3789h	B8	04B8 3789h
B15	xxxx xxxxh	B15	0000 0001h false, true

**Example 3**

CMPEQ2 .S2 B2,B8,B15

Before instruction		1 cycle after instruction	
B2	01B6 2451h	B2	01B6 2451h

B8 B8 B15 B15  true, true

## 4.43 CMPEQ4

Compare for Equality, Packed 8-Bit

**Syntax** **CMPEQ4 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**

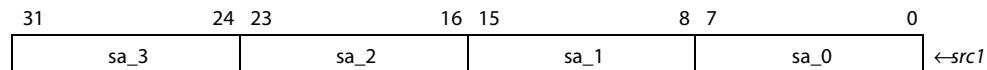
31	29	28	27	dst				src2				src1		
3													5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
5			x	0	1	1	1	0	0	1	0	0	s	p
				1									1	1

Opcode map field used...	For operand type...	Unit
src1	s4	.S1, .S2
src2	xs4	
dst	bv4	

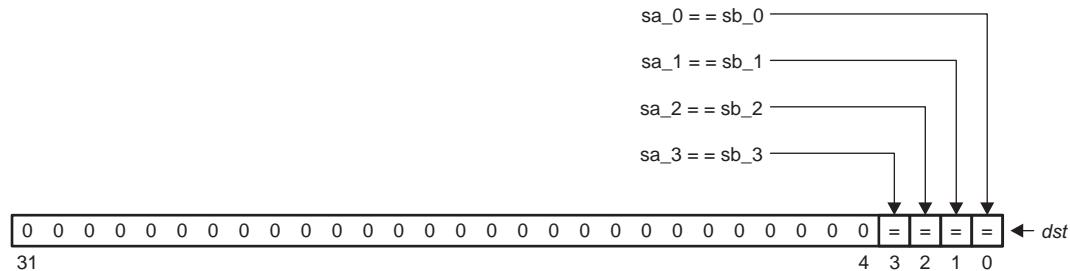
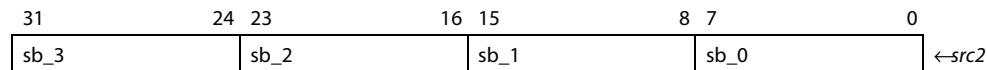
**Description**

Performs equality comparisons on packed 8-bit data. Each 8-bit value in *src1* is compared against the corresponding 8-bit value in *src2*, returning either a 1 if equal or a 0 if not equal. The equality comparison results are packed into the four least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0 to 3, starting with the least-significant byte, then working towards the most-significant byte. The comparison results for byte 0 are written to bit 0 of the result. Likewise the results for byte 1 to 3 are written to bits 1 to 3 of the result, respectively, as shown in the diagram below. The remaining bits of *dst* are cleared to 0.

**CMPEQ4**

↓↑                  ↓↑                  ↓↑                  ↓↑



**Execution**

```

if (cond) {
    if (sbyte0(src1) == sbyte0(src2)), 1 → dst0 else 0 → dst0;
    if (sbyte1(src1) == sbyte1(src2)), 1 → dst1 else 0 → dst1;
    if (sbyte2(src1) == sbyte2(src2)), 1 → dst2 else 0 → dst2;
    if (sbyte3(src1) == sbyte3(src2)), 1 → dst3 else 0 → dst3;
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	<i>.S</i>

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ](#), [CMPEQ2](#), [CMPGTU4](#), [XPND4](#)

**Examples** [Example 1](#)

CMPEQ4 .S1 A3,A4,A5

**Before instruction**

A3      

02 3A 4E 1Ch
--------------

A4      

02 B8 4E 76h
--------------

**1 cycle after instruction**

A3      

02 3A 4E 1Ch
--------------

A4      

02 B8 4E 76h
--------------

A5	xxxx xxxxh	A5	0000 000Ah	true, false, false, false
----	------------	----	------------	---------------------------

### Example 2

CMPEQ4 .S2 B2,B8,B13

	<b>Before instruction</b>	<b>1 cycle after instruction</b>	
B2	F2 3A 37 89h	B2	F2 3A 37 89h
B8	04 B8 37 89h	B8	04 B8 37 89h
B13	xxxx xxxxh	B13	0000 0003h

false, false, true, true

### Example 3

CMPEQ4 .S2 B2,B8,B13

	<b>Before instruction</b>	<b>1 cycle after instruction</b>	
B2	01 B6 24 51h	B2	01 B6 24 51h
B8	05 B6 24 51h	B8	05 B6 24 51h
B13	xxxx xxxxh	B13	0000 0007h

false, true, true, true

## 4.44 CMPEQDP

Compare for Equality, Double-Precision Floating-Point Values

**Syntax**   **CMPEQDP (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		
3				z					23	22				18	17	
	3				5						5				5	
		13	12	11	10	9	8	7	6	5	4	3	2	1	0	
		src1	x	1	0	1	0	0	0	1	0	0	0	s	p	
		5		1										1	1	

Opcode map field used...	For operand type...	Unit
src1	dp	.S1, .S2
src2	x dp	
dst	sint	

**Description**    Compares *src1* to *src2*. If *src1* equals *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Special cases of inputs:

Input		Output	FAUCR Bits	
src1	src2		UNORD	INVAL
NaN	don't care	0	1	0
don't care	NaN	0	1	0
NaN	NaN	0	1	0
+/-denormalized	+/-0	1	0	0
+/-0	+/-denormalized	1	0	0
+/-0	+/-0	1	0	0
+/-denormalized	+/-denormalized	1	0	0
+infinity	+infinity	1	0	0
+infinity	other	0	0	0
-infinity	-infinity	1	0	0
-infinity	other	0	0	0



**Note—**

- 1) In the case of NaN compared with itself, the result is false.

- 2) No configuration bits other than those in the preceding table are set, except the NaNn and DENn bits when appropriate.

**Execution**

```
if (cond) {
    if (src1 == src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1_l, src2_l</i>	<i>src1_h, src2_h</i>
Written		<i>dst</i>
Unit in use	.S	.S

**Instruction Type** DP compare

**Delay Slots** 1

**Functional Unit Latency** 2

**See Also** [CMPEQ](#), [CMPEQSP](#), [CMPGTDP](#), [CMPLTDP](#)

**Example** CMPEQDP .S1 A1:A0,A3:A2,A4

**Before instruction**

A1:A0	4021 3333h	3333 3333h
A3:A2	C004 0000h	0000 0000h
A4	xxxx xxxxh	

**7 cycles after instruction**

A1:A0	4021 3333h	3333 3333h	8.6
A3:A2	C004 0000h	0000 0000h	-2.5
A4	0000 0000h	false	

## 4.45 CMPEQSP

Compare for Equality, Single-Precision Floating-Point Values

**Syntax**   **CMPEQSP (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		
3		1		5					5					5		
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	1	1	1	0	0	0	1	0	0	0	s	p		
		5		1									1	1		

Opcode map field used...	For operand type...	Unit
src1	sp	.S1, .S2
src2	xsp	
dst	sint	

**Description**    Compares *src1* to *src2*. If *src1* equals *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Special cases of inputs:

Input		Output	FAUCR Bits	
src1	src2		UNORD	INVAL
NaN	don't care	0	1	0
don't care	NaN	0	1	0
NaN	NaN	0	1	0
+/-denormalized	+/-0	1	0	0
+/-0	+/-denormalized	1	0	0
+/-0	+/-0	1	0	0
+/-denormalized	+/-denormalized	1	0	0
+infinity	+infinity	1	0	0
+infinity	other	0	0	0
-infinity	-infinity	1	0	0
-infinity	other	0	0	0



**Note—**

- 1) In the case of NaN compared with itself, the result is false.

- 2) No configuration bits other than those in the preceding table are set, except the NaNn and DENn bits when appropriate.

**Execution**

```
if (cond) {
    if (src1 == src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [CMPEQ](#), [CMPEQDP](#), [CMPGTSP](#), [CMPLTSP](#)

**Example** CMPEQSP .S1 A1,A2,A3

**Before instruction**

**1 cycle after instruction**

A1	C020 0000h	A1	C020 0000h	-2.5
A2	4109 999Ah	A2	4109 999Ah	8.6
A3	xxxxxxxxh	A3	0000 0000h	false

## 4.46 CMPGT

Compare for Greater Than, Signed Integers

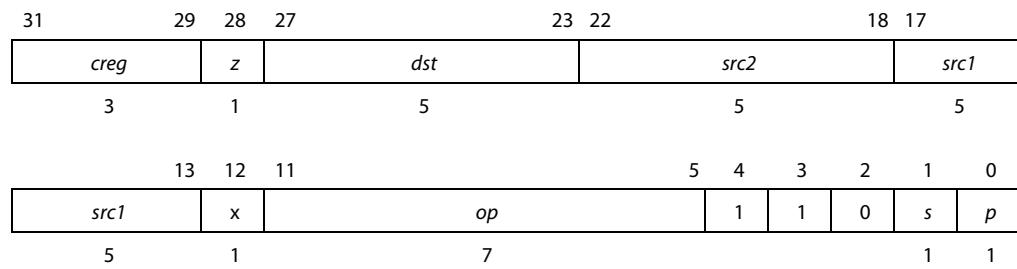
**Syntax** **CMPGT (.unit) src1, src2, dst**

unit = .L1 or .L2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L2c	<a href="#">Figure D-7</a>
	Lx1c	<a href="#">Figure D-10</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.L1,.L2	1000111
<i>src2</i>	xsint		
<i>dst</i>	uint		
<i>src1</i>	scst5	.L1,.L2	1000110
<i>src2</i>	xsint		
<i>dst</i>	uint		
<i>src1</i>	xsint	.L1,.L2	1000101
<i>src2</i>	slong		
<i>dst</i>	uint		
<i>src1</i>	scst5	.L1,.L2	1000100
<i>src2</i>	slong		
<i>dst</i>	uint		

**Description**

Performs a signed comparison of *src1* to *src2*. If *src1* is greater than *src2*, then a 1 is written to *dst*; otherwise, a 0 is written to *dst*.



**Note**—The **CMPGT** instruction allows using a 5-bit constant as *src1*. If *src2* is a 5-bit constant, as in

CMPGT .L1 A4, 5, A0

Then to implement this operation, the assembler converts this instruction to

CMPLT .L1 5, A4, A0

These two instructions are equivalent, with the second instruction using the conventional operand types for *src1* and *src2*.

Similarly, the **CMPGT** instruction allows a cross path operand to be used as *src2*. If *src1* is a cross path operand as in

CMPGT .L1x B4, A5, A0

Then to implement this operation the assembler converts this instruction to

CMPLT .L1x A5, B4, A0

In both of these operations the listing file (.lst) will have the first implementation, and the second implementation will appear in the debugger.

**Execution**

```
if (cond) {
    if (src1 > src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPGT2](#), [CMPGTU](#), [CMPGTU4](#)

**Examples** **Example 1**

CMPGT .L1X A1,B1,A2

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A1	000001B6h	438	A1	000001B6h	
A2	xxxxxxxx		A2	00000000h	false
B1	0000 08BDh	2237	B1	000008BDh	

**Example 2**

CMPGT .L1X A1,B1,A2

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A1	FFFF FE91h	-367	A1	FFFF FE91h	
A2	xxxxxxxx		A2	00000001h	true
B1	FFFF FDC4h	-572	B1	FFFF FDC4h	

### Example 3

CMPGT .L1 8,A1,A2

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A1	0000 0023h	35	A1	0000 0023h	
A2	xxxxxxxx		A2	0000000h	false

### Example 4

CMPGT .L1X A1,B1,A2

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A1	0000 00EBh	235	A1	0000 00EBh	
A2	xxxxxxxx		A2	0000000h	false
B1	0000 00EBh	235	B1	0000 00EBh	

## 4.47 CMPGT2

Compare for Greater Than, Packed 16-Bit

**Syntax** **CMPGT2 (.unit) src1, src2, dst**

unit = .S1 or .S2

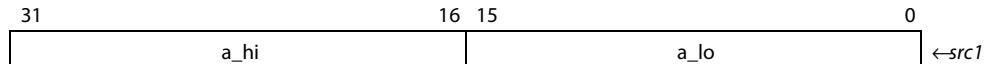
**Opcode**

31	29	28	27	dst					src2					src1		
3	1	5					5					5		5		
src1										x	0	1	0	1	0	0
5	1														1	1

Opcode map field used...	For operand type...	Unit
src1	s2	.S1, .S2
src2	xs2	
dst	bv2	

**Description**

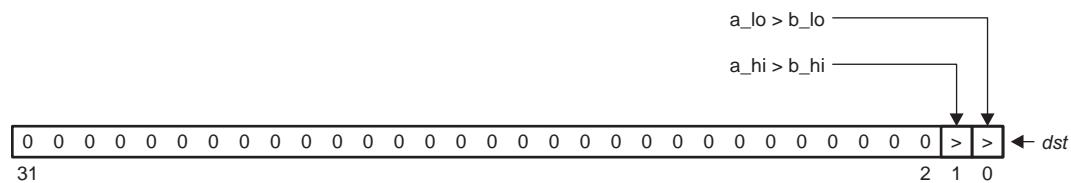
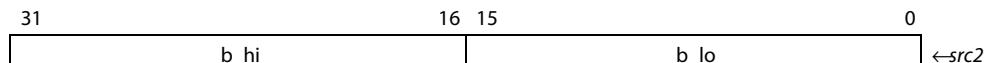
Performs comparisons for greater than values on signed, packed 16-bit data. Each signed 16-bit value in *src1* is compared against the corresponding signed 16-bit value in *src2*, returning a 1 if *src1* is greater than *src2* or returning a 0 if it is not greater. The comparison results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are cleared to 0.



**CMPGT2**

↑↓

↓↑



**Execution**

```

if (cond){
    if (lsb16(src1) > lsb16(src2)), 1 → dst0 else 0 → dst0;
    if (msb16(src1) > msb16(src2)), 1 → dst1 else 0 → dst1
}
else nop
  
```

**Pipeline**

Pipeline Stage	E1
Read	src <sub>1</sub> , src <sub>2</sub>
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ2](#), [CMPGT](#), [CMPGTU](#), [CMPGTU4](#), [XPND2](#)

**Examples** [Example 1](#)

CMPGT2 .S1 A3,A4,A5

**Before instruction**

A3	1105 6E30h	4357 28208
A4	1105 6980h	4357 27008
A5	xxxx xxxxh	

**1 cycle after instruction**

A3	1105 6E30h	
A4	1105 6980h	
A5	0000 0001h	false, true

**Example 2**

CMPGT2 .S2 B2,B8,B15

**Before instruction**

B2	F348 3789h	-3526 14217
B8	04B8 4975h	1208 18805
B15	xxxx xxxxh	

**1 cycle after instruction**

B2	F348 3789h	
B8	04B8 4975h	
B15	0000 0000h	false, false

**Example 3**

CMPGT2 .S2 B2, B8, B15

**Before instruction**

B2	01A6 2451h	422 9297
----	------------	----------

**1 cycle after instruction**

B2	01A6 2451h	
----	------------	--

B8	<input type="text" value="0124 A051h"/>	292 -24495	B8	<input type="text" value="0124 A051h"/>
B15	<input type="text" value="xxxx xxxxh"/>		B15	<input type="text" value="0000 0003h"/> true, true

## 4.48 CMPGTDP

Compare for Greater Than, Double-Precision Floating-Point Values

**Syntax** **CMPGTDP** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		
3		1			5					5					5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
5		x	1	0	1	0	0	1	1	0	0	0	s	p	1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	dp	.S1, .S2
<i>src2</i>	xdp	
<i>dst</i>	sint	

**Description** Compares *src1* to *src2*. If *src1* is greater than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Special cases of inputs:

Input		Output	FAUCR Bits	
src1	src2		UNORD	INVAL
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/-denormalized	+/-0	0	0	0
+/-0	+/-denormalized	0	0	0
+/-0	+/-0	0	0	0
+/-denormalized	+/-denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	1	0	0
-infinity	-infinity	0	0	0
-infinity	other	0	0	0



**Note**—No configuration bits other than those in the preceding table are set, except the NaNn and DENn bits when appropriate.

**Execution**

```
if (cond) {
    if (src1 > src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src1_l, src2_l	src1_h, src2_h
Written		dst
Unit in use	.S	.S

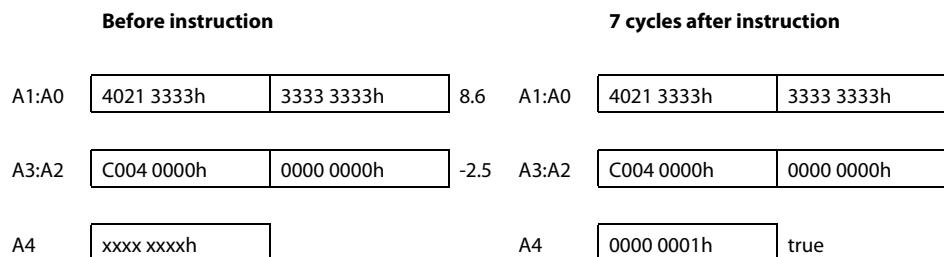
**Instruction Type** DP compare

**Delay Slots** 1

**Functional Unit Latency** 2

**See Also** [CMPEQDP](#), [CMPGT](#), [CMPGTU](#), [CMPGTSP](#), [CMPLTDP](#)

**Example** CMPGTDP .S1 A1:A0,A3:A2,A4



## 4.49 CMPGTSP

Compare for Greater Than, Single-Precision Floating-Point Values

**Syntax** **CMPGTSP** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		
3		1		5					5					5		5
src1	x	1	1	1	0	0	1	1	0	0	0	s	p	1	1	
5		1														

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sp	.S1, .S2
<i>src2</i>	xsp	
<i>dst</i>	sint	

**Description** Compares *src1* to *src2*. If *src1* is greater than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Special cases of inputs:

<b>Input</b>		<b>Output</b>	<b>FAUCR Bits</b>	
<b>src1</b>	<b>src2</b>		<b>UNORD</b>	<b>INVAL</b>
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/-denormalized	+/-0	0	0	0
+/-0	+/-denormalized	0	0	0
+/-0	+/-0	0	0	0
+/-denormalized	+/-denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	1	0	0
-infinity	-infinity	0	0	0
-infinity	other	0	0	0



**Note**—No configuration bits other than those in the preceding table are set, except the NaNn and DENn bits when appropriate.

**Execution**

```
if (cond) {
    if (src1 > src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

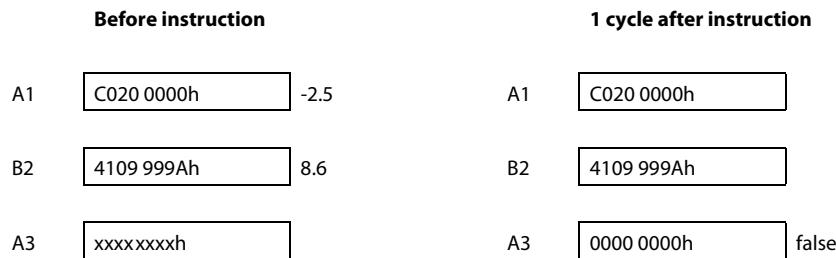
**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [CMPEQSP](#), [CMPGT](#), [CMPGTU](#), [CMPGTDP](#), [CMPLTSP](#)

**Example** CMPGTSP .S1X A1,B2,A3



## 4.50 CMPGTU

Compare for Greater Than, Unsigned Integers

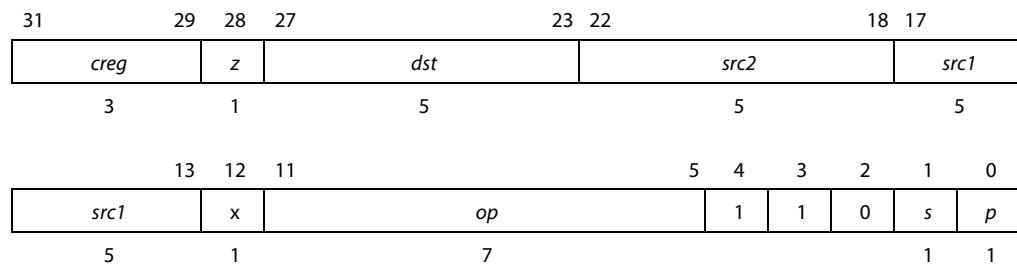
**Syntax** **CMPGTU** (.unit) *src1*, *src2*, *dst*

unit = .L1 or .L2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L2c Lx1c	<a href="#">Figure D-7</a> <a href="#">Figure D-10</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i> <i>src2</i> <i>dst</i>	uint xuint uint	.L1, .L2	1001111
<i>src1</i> <i>src2</i> <i>dst</i>	ucst4 xuint uint	.L1, .L2	1001110
<i>src1</i> <i>src2</i> <i>dst</i>	xuint ulong uint	.L1, .L2	1001101
<i>src1</i> <i>src2</i> <i>dst</i>	ucst4 ulong uint	.L1, .L2	1001100

**Description** Performs an unsigned comparison of *src1* to *src2*. If *src1* is greater than *src2*, then a 1 is written to *dst*; otherwise, a 0 is written to *dst*. Only the four LSBs are valid in the 5-bit *dst* field when the *ucst4* operand is used. If the MSB of the *dst* field is nonzero, the result is invalid.

**Execution**

```
if (cond) {
    if (src1 > src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

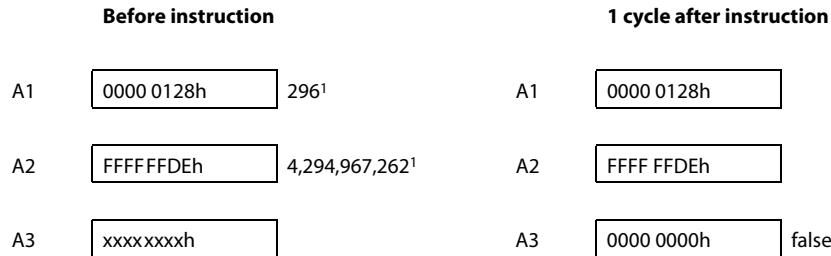
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPGT](#), [CMPGT2](#), [CMPGTU4](#)

**Examples** [Example 1](#)

CMPGTU .L1 A1,A2,A3



1. Unsigned 32-bit integer

**Example 2**

CMPGTU .L1 0Ah,A1,A2

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A1	0000 0005h	5 <sup>1</sup>	A1	0000 0005h	
A2	xxxxxxxxh		A2	0000 0001h	true

1. Unsigned 32-bit integer

### Example 3

CMPGTU .L1 0Eh, A3:A2, A4

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A3:A2	0000 0000h	0000000Ah	10 <sup>1</sup>	A3:A2	0000 0000h
A4	xxxxxxxxh		A4	00000001h	true

1. Unsigned 40-bit (long) integer

## 4.51 CMPGTU4

Compare for Greater Than, Unsigned, Packed 8-Bit

**Syntax** **CMPGTU4** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

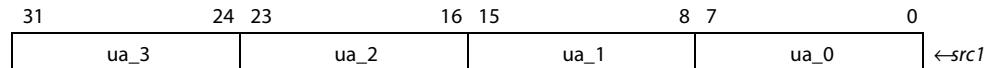
31	29	28	27	dst					src2					src1		
3															5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	<i>src1</i>	x	0	1	0	1	0	1	1	0	0	0	s	p		
	5		1										1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	u4	.S1, .S2
<i>src2</i>	xu4	
<i>dst</i>	bv4	

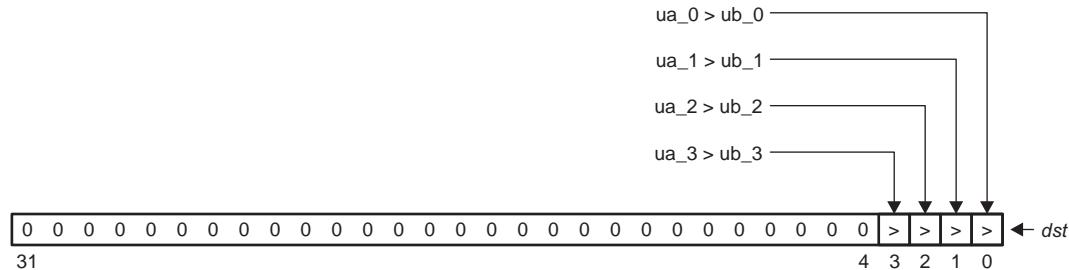
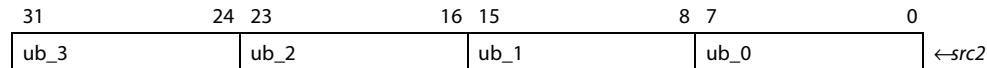
**Description**

Performs comparisons for greater than values on packed 8-bit data. Each unsigned 8-bit value in *src1* is compared against the corresponding unsigned 8-bit value in *src2*, returning a 1 if the byte in *src1* is greater than the corresponding byte in *src2* or a 0 if it is not greater. The comparison results are packed into the four least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0 to 3, starting with the least-significant byte, then working towards the most-significant byte. The comparison results for byte 0 are written to bit 0 of the result. Likewise, the results for byte 1 to 3 are written to bits 1 to 3 of the result, respectively, as shown in the diagram below. The remaining bits of *dst* are cleared to 0.


**CMPGTU4**

$\downarrow\uparrow$        $\downarrow\uparrow$        $\downarrow\uparrow$        $\downarrow\uparrow$



**Execution**

```

if (cond) {
    if (ubyte0(src1) > ubyte0(src2)), 1 → dst0 else 0 → dst0;
    if (ubyte1(src1) > ubyte1(src2)), 1 → dst1 else 0 → dst1;
    if (ubyte2(src1) > ubyte2(src2)), 1 → dst2 else 0 → dst2;
    if (ubyte3(src1) > ubyte3(src2)), 1 → dst3 else 0 → dst3
}
else nop
  
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ4](#), [CMPGT](#), [CMPGT2](#), [CMPGTU](#), [CMPLT](#), [XPND4](#)

**Examples** [Example 1](#)

CMPGTU4 .S1 A3,A4,A5

**Before instruction**

A3 

25 3A 1C E4h
--------------

 37 58 28 228

A4 

02 B8 4E 76h
--------------

 2 184 78 118

**1 cycle after instruction**

A3 

25 3A 1C E4h
--------------

A4 

02 B8 4E 76h
--------------

A5 xxxx xxxxh

 A5 0000 0009h true, false, false, true

### Example 2

CMPGTU4 .S2 B2,B8,B13

	<b>Before instruction</b>	<b>1 cycle after instruction</b>
B2	<span style="border: 1px solid black; padding: 2px;">89 F2 3A 37h</span> 137 242 58 55	<span style="border: 1px solid black; padding: 2px;">89 F2 3A 37h</span>
B8	<span style="border: 1px solid black; padding: 2px;">04 8F 17 89h</span> 4 143 23 137	<span style="border: 1px solid black; padding: 2px;">04 8F 17 89h</span>
B13	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>	<span style="border: 1px solid black; padding: 2px;">0000 000Eh</span> true, true, true, false

### Example 3

CMPGTU4 .S2 B2,B8,B13

	<b>Before instruction</b>	<b>1 cycle after instruction</b>
B2	<span style="border: 1px solid black; padding: 2px;">12 33 9D 51h</span> 18 51 157 81	<span style="border: 1px solid black; padding: 2px;">12 33 9D 51h</span>
B8	<span style="border: 1px solid black; padding: 2px;">75 67 24 C5h</span> 117 103 36 197	<span style="border: 1px solid black; padding: 2px;">75 67 24 C5h</span>
B13	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>	<span style="border: 1px solid black; padding: 2px;">0000 0002h</span> false, false, true, false

## 4.52 CMPLT

Compare for Less Than, Signed Integers

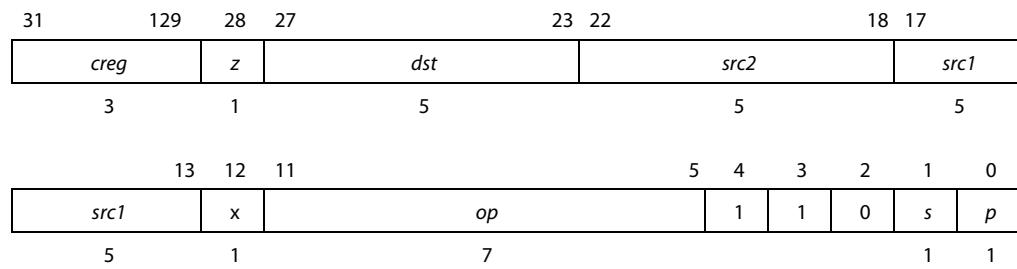
**Syntax** **CMPLT** (.unit) *src1*, *src2*, *dst*

unit = .L1 or .L2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L2c	<a href="#">Figure D-7</a>
	Lx1c	<a href="#">Figure D-10</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.L1,.L2	1010111
<i>src2</i>	xsint		
<i>dst</i>	uint		
<i>src1</i>	scst5	.L1,.L2	1010110
<i>src2</i>	xsint		
<i>dst</i>	uint		
<i>src1</i>	xsint	.L1,.L2	1010101
<i>src2</i>	slong		
<i>dst</i>	uint		
<i>src1</i>	scst5	.L1,.L2	1010100
<i>src2</i>	slong		
<i>dst</i>	uint		

**Description** Performs a signed comparison of *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.



**Note**—The **CMPLT** instruction allows using a 5-bit constant as *src1*. If *src2* is a 5-bit constant, as in

CMPLT .L1 A4, 5, A0

Then to implement this operation, the assembler converts this instruction to

CMPGT .L1 5, A4, A0

These two instructions are equivalent, with the second instruction using the conventional operand types for *src1* and *src2*.

Similarly, the **CMPLT** instruction allows a cross path operand to be used as *src2*. If *src1* is a cross path operand as in

CMPLT .L1x B4, A5, A0

Then to implement this operation, the assembler converts this instruction to

CMPGT .L1x A5, B4, A0

In both of these operations the listing file (.lst) will have the first implementation, and the second implementation will appear in the debugger.

**Execution**

```
if (cond){  
    if (src1 < src2), 1 → dst  
    else 0 → dst  
}  
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPLT2](#), [CMPLTU](#), [CMPLTU4](#)

**Examples** [Example 1](#)

CMPLT .L1 A1,A2,A3

**Before instruction**

A1	0000 07E2h	2018
A2	0000 0F6Bh	3947
A3	xxxxxxxxh	

**1 cycle after instruction**

A1	0000 07E2h
A2	0000 0F6Bh
A3	0000 0001h

true

**Example 2**

CMPLT .L1 A1,A2,A3

**Before instruction**

A1	FFFF FED6h	-298
A2	0000 000Ch	12
A3	xxxxxxxxh	

**1 cycle after instruction**

A1	FFFF FED6h
A2	0000 000Ch
A3	0000 0001h

true

### Example 3

```
CMPLT .L1    9,A1,A2
```

**Before instruction**

A1	00000005h	5
----	-----------	---

A2	xxxxxxxxh
----	-----------

**1 cycle after instruction**

A1	00000005h
----	-----------

A2	00000000h	false
----	-----------	-------

## 4.53 CMPLT2

Compare for Less Than, Packed 16-Bit

**Syntax** **CMPLT2 (.unit) src2, src1, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		18	17
creg	z																
3	1			5					5							5	
src1	x	0	1	0	1	0	0	1	0	0	0	0	s	p		1	0
5	1															1	1

Opcode map field used...	For operand type...	Unit
src1	s2	.S1, .S2
src2	xs2	
dst	bv2	

**Description**

The **CMPLT2** instruction is a pseudo-operation used to perform less-than comparisons on signed, packed 16-bit data. Each signed 16-bit value in *src2* is compared against the corresponding signed 16-bit value in *src1*, returning a 1 if *src2* is less than *src1* or returning a 0 if it is not less than. The comparison results are packed into the two least-significant bits of *dst*. The result for the lower pair of values is placed in bit 0, and the results for the upper pair of values are placed in bit 1. The remaining bits of *dst* are cleared to 0.

The assembler uses the operation **CMPGT2 (.unit) src1, src2, dst** to perform this task (see [CMPGT](#)).

**Execution**

```
if (cond){
    if (lsb16(src2) < lsb16(src1)), 1 → dst0 else 0 → dst0;
    if (msb16(src2) < msb16(src1)), 1 → dst1 else 0 → dst1
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ2](#), [CMPGT2](#), [CMPLT](#), [CMPLTU](#), [CMPLTU4](#), [XPND2](#)

**Examples**

**Example 1**

```
CMPLT2 .S1 A4,A3,A5; assembler treats as CMPGT2 A3,A4,A5
```

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
A3	1105 6E30h	4357 28208	A3    1105 6E30h
A4	1105 6980h	4357 27008	A4    1105 6980h
A5	xxxx xxxxh		A5    0000 0001h    false, true

### Example 2

CMPLT2 .S2 B8,B2,B15; assembler treats as CMPGT2 B2,B8,B15

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
B2	F23A 3789h	-3526 14217	B2    F23A 3789h
B8	04B8 4975h	1208 18805	B8    04B8 4975h
B15	xxxx xxxxh		B15    0000 0000h    false, false

### Example 3

CMPLT2 .S2 B8,B2,B12; assembler treats as CMPGT2 B2,B8,B15

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
B2	01A6 2451h	422 9297	B2    01A6 2451h
B8	0124 A051h	292 -24495	B8    0124 A051h
B12	xxxx xxxxh		B12    0000 0003h    true, true

## 4.54 CMPLTDP

Compare for Less Than, Double-Precision Floating-Point Values

**Syntax** **CMPLTDP** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	23 22						18 17			
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>		<i>src1</i>	
3	1			5						5		5	
<i>src1</i>	<i>x</i>	1	0	1	0	1	0	1	0	0	0	<i>s</i>	<i>p</i>
5	1									1		1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	dp	.S1, .S2
<i>src2</i>	xdp	
<i>dst</i>	sint	

**Description** Compares *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Special cases of inputs:

<b>Input</b>		<b>Output</b>	<b>FAUCR Bits</b>	
<b>src1</b>	<b>src2</b>		<b>UNORD</b>	<b>INVAL</b>
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/-denormalized	+/-0	0	0	0
+/-0	+/-denormalized	0	0	0
+/-0	+/-0	0	0	0
+/-denormalized	+/-denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	0	0	0
-infinity	-infinity	0	0	0
-infinity	other	1	0	0



**Note**—No configuration bits other than those in the preceding table are set, except the NaNn and DENn bits when appropriate.

**Execution**

```
if (cond) {
    if (src1 < src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>
Read	<i>src1_l, src2_l</i>	<i>src1_h, src2_h</i>
Written		<i>dst</i>
Unit in use	.S	.S

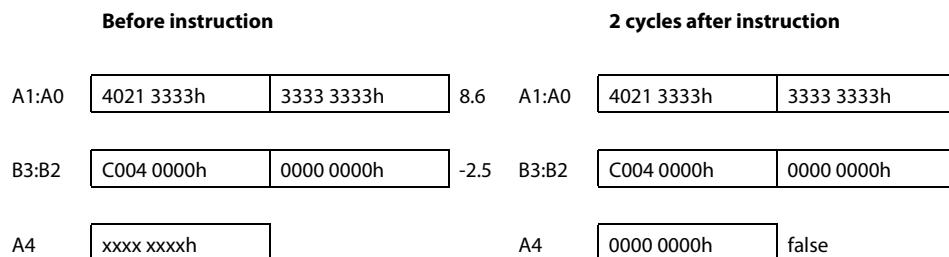
**Instruction Type** DP compare

**Delay Slots** 1

**Functional Unit Latency** 2

**See Also** [CMPEQDP](#), [CMPGTDP](#), [CMPLT](#), [CMPLTSP](#), [CMPLTU](#)

**Example** CMPLTDP .S1X A1:A0,B3:B2,A4



## 4.55 CMPLTSP

Compare for Less Than, Single-Precision Floating-Point Values

**Syntax** **CMPLTSP** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	23 22					18 17						
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>						
3	1			5					5				5		
<i>src1</i>	<i>x</i>	1	1	1	0	1	0	1	0	0	0	<i>s</i>	<i>p</i>	1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sp	.S1, .S2
<i>src2</i>	xsp	
<i>dst</i>	sint	

**Description** Compares *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

Special cases of inputs:

<b>Input</b>		<b>FAUCR Bits</b>		
<b>src1</b>	<b>src2</b>	<b>Output</b>	<b>UNORD</b>	<b>INVAL</b>
NaN	don't care	0	1	1
don't care	NaN	0	1	1
NaN	NaN	0	1	1
+/-denormalized	+/-0	0	0	0
+/-0	+/-denormalized	0	0	0
+/-0	+/-0	0	0	0
+/-denormalized	+/-denormalized	0	0	0
+infinity	+infinity	0	0	0
+infinity	other	0	0	0
-infinity	-infinity	0	0	0
-infinity	other	1	0	0



**Note**—No configuration bits other than those in the preceding table are set, except the NaNn and DENn bits when appropriate.

**Execution**

```
if (cond) {
    if (src1 < src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [CMPEQSP](#), [CMPGTSP](#), [CMPLT](#), [CMPLTDP](#), [CMPLTU](#)

**Example** CMPLTSP .S1 A1,A2,A3

**Before instruction**

**1 cycle after instruction**

A1	<table border="1"><tr><td>C020 0000h</td></tr></table>	C020 0000h	-2.5	A1	<table border="1"><tr><td>C020 0000h</td></tr></table>	C020 0000h
C020 0000h						
C020 0000h						
A2	<table border="1"><tr><td>4109 999Ah</td></tr></table>	4109 999Ah	8.6	A2	<table border="1"><tr><td>4109 999Ah</td></tr></table>	4109 999Ah
4109 999Ah						
4109 999Ah						
A3	<table border="1"><tr><td>xxxxxxxx</td></tr></table>	xxxxxxxx		A3	<table border="1"><tr><td>0000 0001h</td></tr></table>	0000 0001h
xxxxxxxx						
0000 0001h						

## 4.56 CMPLTU

Compare for Less Than, Unsigned Integers

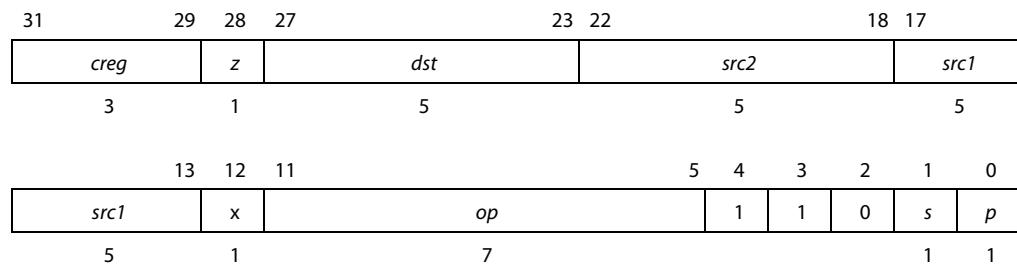
**Syntax** **CMPLTU** (.unit) *src1*, *src2*, *dst*

unit = .L1 or .L2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L2c Lx1c	<a href="#">Figure D-7</a> <a href="#">Figure D-10</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i> <i>src2</i> <i>dst</i>	uint xuint uint	.L1, .L2	1011111
<i>src1</i> <i>src2</i> <i>dst</i>	ucst4 xuint uint	.L1, .L2	1011110
<i>src1</i> <i>src2</i> <i>dst</i>	xuint ulong uint	.L1, .L2	1011101
<i>src1</i> <i>src2</i> <i>dst</i>	ucst4 ulong uint	.L1, .L2	1011100

**Description** Performs an unsigned comparison of *src1* to *src2*. If *src1* is less than *src2*, then 1 is written to *dst*; otherwise, 0 is written to *dst*.

**Execution**

```
if (cond) {
    if (src1 < src2), 1 → dst
    else 0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPLT](#), [CMPLT2](#), [CMPLTU4](#)

**Examples** [Example 1](#)

CMPLTU .L1 A1,A2,A3

Before instruction			1 cycle after instruction		
A1	0000289Ah	10,394 <sup>1</sup>	A1	0000289Ah	
A2	FFFFF35Eh	4,294,964,062 <sup>1</sup>	A2	FFFFF35Eh	
A3	xxxxxxxxh		A3	0000 0001h	true

1. Unsigned 32-bit integer

**Example 2**

CMPLTU .L1 14,A1,A2

**Before instruction**
**1 cycle after instruction**

A1	<table border="1"><tr><td>0000000Fh</td></tr></table>	0000000Fh	15 <sup>1</sup>	A1	<table border="1"><tr><td>0000000Fh</td></tr></table>	0000000Fh
0000000Fh						
0000000Fh						
A2	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A2	<table border="1"><tr><td>0000 0001h</td></tr></table>	0000 0001h
xxxxxxxxh						
0000 0001h						

1. Unsigned 32-bit integer

**Example 3**

CMPLTU .L1 A1,A5:A4,A2

**Before instruction**
**1 cycle after instruction**

A1	<table border="1"><tr><td>003B8260h</td></tr></table>	003B8260h	3,900,000 <sup>1</sup>	A1	<table border="1"><tr><td>003B8260h</td></tr></table>	003B8260h		
003B8260h								
003B8260h								
A2	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A2	<table border="1"><tr><td>00000000h</td></tr></table>	00000000h		
xxxxxxxxh								
00000000h								
A5:A4	<table border="1"><tr><td>00000000h</td><td>003A0002h</td></tr></table>	00000000h	003A0002h	3,801,090 <sup>2</sup>	A5:A4	<table border="1"><tr><td>00000000h</td><td>003A0002h</td></tr></table>	00000000h	003A0002h
00000000h	003A0002h							
00000000h	003A0002h							

1. Unsigned 32-bit integer  
 2. Unsigned 40-bit (long) integer

## 4.57 CMPLTU4

Compare for Less Than, Unsigned, Packed 8-Bit

**Syntax** **CMPLTU4 (.unit) src2, src1, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		18	17
3				z					23	22							
	1								5					5			5
		13	12	11	10	9	8	7	6	5	4	3	2	1	0		
		src1	x	0	1	0	1	0	1	1	0	0	0	s	p	1	1
		5		1													

Opcode map field used...	For operand type...	Unit
src1	u4	.S1, .S2
src2	xu4	
dst	bv4	

**Description**

The **CMPLTU4** instruction is a pseudo-operation that performs less-than comparisons on packed 8-bit data. Each unsigned 8-bit value in *src2* is compared against the corresponding unsigned 8-bit value in *src1*, returning a 1 if the byte in *src2* is less than the corresponding byte in *src1* or a 0 if it is not less than. The comparison results are packed into the four least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0 to 3, starting with the least-significant byte, and moving towards the most-significant byte. The comparison results for byte 0 are written to bit 0 of the result. Similarly, the results for byte 1 to 3 are written to bits 1 to 3 of the result, respectively, as shown in the diagram below. The remaining bits of *dst* are cleared to 0.

The assembler uses the operation **CMPGTU4** (.unit) *src1, src2, dst* to perform this task (see [CMPGTU4](#)).

**Execution**

```
if (cond) {
    if (ubyte0(src2) < ubyte0(src1)), 1 → dst0 else 0 → dst0;
    if (ubyte1(src2) < ubyte1(src1)), 1 → dst1 else 0 → dst1;
    if (ubyte2(src2) < ubyte2(src1)), 1 → dst2 else 0 → dst2;
    if (ubyte3(src2) < ubyte3(src1)), 1 → dst3 else 0 → dst3;
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [CMPEQ4](#), [CMPGT](#), [CMPLT](#), [CMPLT2](#), [CMPLTU](#), [XPND4](#)

**Examples**

**Example 1**

CMPLTU4 .S1 A4,A3,A5; assembler treats as CMPGTU4 A3,A4,A5

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A3	25 3A 1C E4h	37 58 28 228	A3	25 3A 1C E4h	
A4	02 B8 4E 76h	2 184 78 118	A4	02 B8 4E 76h	
A5	xxxx xxxxh		A5	0000 0009h	true, false, false, true

**Example 2**

CMPLTU4 .S2 B8,B2,B13; assembler treats as CMPGTU4 B2,B8,B13

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
B2	89 F2 3A 37h	137 242 58 55	B2	89 F2 3A 37h	
B8	04 8F 17 89h	4 143 23 137	B8	04 8F 17 89h	
B13	xx xx xx xxh		B13	0000 000Eh	true, true, true, false

**Example 3**

CMPLTU4 .S2 B8,B2,B13; assembler treats as CMPGTU4 B2,B8,B13

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
B2	12 33 9D 51h	18 51 157 81	B2	12 33 9D 51h	
B8	75 67 24 C5h	117 103 36 197	B8	75 67 24 C5h	
B13	xx xx xx xxh		B13	0000 0002h	false, false, true, false

## 4.58 CMPY

Complex Multiply Two Pairs, Signed, Packed 16-Bit

**Syntax** **CMPY (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Opcode**

31	30	29	28	27	dst					src2					src1			
5	5	5	5	5	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					src1	x	0	0	1	0	1	0	1	1	0	0	s	p

Opcode map field used...	For operand type...	Unit
src1	s2	.M1, .M2
src2	xs2	
dst	dint	

**Description**

Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed results are written to a 64-bit register pair.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The result is written to *dst\_o*.

The product of the upper halfword of *src1* and the lower halfword of *src2* is added to the product of the lower halfword of *src1* and the upper halfword of *src2*. The result is written to *dst\_e*.

If the result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the result is written to *dst\_e*.

This instruction executes unconditionally.



**Note**—In the overflow case, where all four halfwords in *src1* and *src2* are 8000h, the saturation value 7FFF FFFFh is written into the 32-bit *dst\_e* register.

**Execution**

```
sat((lsb16(src1) × msb16(src2)) + (msb16(src1) × lsb16(src2))) → dst_e
(msb16(src1) × msb16(src2)) - (lsb16(src1) × lsb16(src2)) → dst_o
```

**Instruction Type**

Four-cycle

**Delay Slots**

3

**See Also** [CMPYR](#), [CMPYR1](#), [DOTP2](#), [DOTPN2](#)

**Examples**

**Example 1**

```
CMPY .M1 A0,A1,A3:A2
```

<b>Before instruction</b>		<b>4 cycles after instruction<sup>1</sup></b>			
A0	<table border="1"><tr><td>0008 0004h</td></tr></table>	0008 0004h	A2	<table border="1"><tr><td>0000 0034h</td></tr></table>	0000 0034h
0008 0004h					
0000 0034h					
A1	<table border="1"><tr><td>0009 0002h</td></tr></table>	0009 0002h	A3	<table border="1"><tr><td>0000 0040h</td></tr></table>	0000 0040h
0009 0002h					
0000 0040h					

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 2

CMPY .M2X B0,A1,B3:B2

<b>Before instruction</b>		<b>4 cycles after instruction<sup>1</sup></b>			
B0	<table border="1"><tr><td>7FFF 7FFFh</td></tr></table>	7FFF 7FFFh	B2	<table border="1"><tr><td>FFFF 8001h</td></tr></table>	FFFF 8001h
7FFF 7FFFh					
FFFF 8001h					
A1	<table border="1"><tr><td>7FFF 8000h</td></tr></table>	7FFF 8000h	B3	<table border="1"><tr><td>7FFE 8001h</td></tr></table>	7FFE 8001h
7FFF 8000h					
7FFE 8001h					

1. CSR.SAT and SSR.M2 unchanged by operation

### Example 3

CMPY .M1 A0,A1,A3:A2

<b>Before instruction</b>		<b>4 cycles after instruction<sup>1</sup></b>			
A0	<table border="1"><tr><td>8000 8000h</td></tr></table>	8000 8000h	A2	<table border="1"><tr><td>7FFF FFFFh</td></tr></table>	7FFF FFFFh
8000 8000h					
7FFF FFFFh					
A1	<table border="1"><tr><td>8000 8000h</td></tr></table>	8000 8000h	A3	<table border="1"><tr><td>0000 0000h</td></tr></table>	0000 0000h
8000 8000h					
0000 0000h					

1. CSR.SAT and SSR.M1 unchanged by operation

4.59 CMPY32R1

## Complex Multiply With Rounding, Signed Complex 32-bit (32-bit Real/32-bit Imaginary)

**Syntax**    **CMPY32R1** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode**      Opcode for .M Unit, Compound Results, new opcode space

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	01000

<b>Description</b>	The CMPY32R1 instruction performs one complex multiply between two 64-bit complex numbers. The 64-bit complex number is contained in a register pair. The Odd register in the pair (the most significant word) represents the real component of the complex number as a 32-bit signed quantity. The Even register in the pair represents the imaginary component of the complex number.
--------------------	---

After multiplying and adding the 32-bit numbers together, they are shifted right by 31, rounded and saturated to 32-bits Intermediate results are calculated at 64-bits.

The saturation condition of `0x80000000:0x80000000 * 0x80000000:0x80000000` yields a result of `0x00000000:0x7FFFFFFF`

The CMPY32R1 instruction is functionally equivalent to following C sequence:

**Execution**      sat(src1\_o x src2\_e + src1\_e x src2\_o + (1<<30))>>31 -> dst\_e  
 src1\_o x src2\_o - src1\_e x src2\_e + (1<<30))>>31 -> dst\_o

**Instruction Type** 4-cycle

## *Delay Slots*

## ***Functional Unit Latency***

**See Also**      [CMPY](#), [SMPY32](#), [CCMPY32R1](#)

```

Example   CSR == 0x00000000 ; 4
           A3 == 0x00000000
           A2 == 0x00000000
           A1 == 0x00000000
           A0 == 0x00000000
           CMPY32R1 .M A3:A2,A1:A0,A15:A14
           A15 == 0x00000000
           A14 == 0x00000000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x00000000
A2 == 0x08000400
A1 == 0x00000000
A0 == 0x09000200
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0xffff6ffff98

```

#### 4.59 CMPY32R1

##### Chapter 4—Instruction Descriptions

```

A14 == 0x00000000
CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x80000000
A2 == 0x80000000
A1 == 0x80000000
A0 == 0x80000000
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x7fffffff

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0x7FFF7FFF
A2 == 0x7FFF8000
A1 == 0x7FFF8000
A0 == 0x7FFF7FFF
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x7fffffff

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0xFFFFFFFF
A2 == 0xFFFFFFFF
A1 == 0xFFFFFFFF
A0 == 0xFFFFFFFF
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x55555555
A2 == 0x55555555
A1 == 0x55555555
A0 == 0x55555555
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x71c71c71

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x01234567
A2 == 0x89ABCDEF
A1 == 0x89ABCDEF
A0 == 0x01234567
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x6d660a7f

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x7fff7fff
A2 == 0x80000000
A1 == 0x7fff7fff
A0 == 0x7fffffff
CMPY32R1 .M A3:A2,A1:A0,A15:A14
A15 == 0x7fffffff
A14 == 0xffffffff

CSR= 0x00000200 ; 4

CSR == 0x00000000 ; 4
B3 == 0x80000000
B2 == 0x80000000
B1 == 0x80000000
B0 == 0x7fffffff
CMPY32R1 .M B3:B2,B1:B0,B15:B14
B15 == 0x7fffffff
B14 == 0x00000001

CSR= 0x00000200 ; 4

```

```
CSR == 0x00000000 ; 4
B3 == 0x80000000
B2 == 0x80000000
B1 == 0x7fffffff
B0 == 0x7fffffff
CMPY32R1 .M B3:B2,B1:B0,B15:B14
B15 == 0x00000000
B14 == 0x80000000

CSR= 0x00000200 ; 4

CSR == 0x00000000 ; 4
B3 == 0x80000000
B2 == 0x7fffffff
B1 == 0x7fffffff
B0 == 0x7fffffff
CMPY32R1 .M B3:B2,B1:B0,B15:B14
B15 == 0x80000000
B14 == 0xffffffff

CSR= 0x00000200 ; 4
```

## 4.60 CMPYR

Complex Multiply Two Pairs, Signed, Packed 16-Bit With Rounding

**Syntax** **CMPYR (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	30	29	28	27	dst					src2					src1			
5	5	5	5	5	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					src1	x	0	0	1	0	1	1	1	1	0	0	s	p

Opcode map field used...	For operand type...	Unit
src1	s2	.M1, .M2
src2	xs2	
dst	s2	

**Description**

Performs two dot-products between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded with saturation, shifted, packed and written to a 32-bit register.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The result is rounded by adding  $2^{15}$  to it. The 16 most-significant bits of the rounded value are written to the upper half of *dst*.

The product of the upper halfword of *src1* and the lower halfword of *src2* is added to the product of the lower halfword of *src1* and the upper halfword of *src2*. The result is rounded by adding  $2^{15}$  to it. The 16 most-significant bits of the rounded value are written to the lower half of *dst*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the result is written to *dst*.

This instruction executes unconditionally.

**Execution**

```

sat((lsb16(src1) × msb16(src2)) + (msb16(src1) × lsb16(src2))) → tmp_e
msb16(sat(tmp_e + 00008000h)) → lsb16(dst)
sat((msb16(src1) × msb16(src2)) - (lsb16(src1) × lsb16(src2))) → tmp_o
msb16(sat(tmp_o + 00008000h)) → msb16(dst)

```

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [CMPY](#), [CMPYR1](#), [DOTP2](#), [DOTPN2](#)

**Examples** [Example 1](#)

CMPYR .M1 A0,A1,A2

**Before instruction**
**4 cycles after instruction<sup>1</sup>**

A0	<table border="1"><tr><td>0800 0400h</td></tr></table>	0800 0400h	A2	<table border="1"><tr><td>0040 0034h</td></tr></table>	0040 0034h
0800 0400h					
0040 0034h					
A1	<table border="1"><tr><td>0900 0200h</td></tr></table>	0900 0200h			
0900 0200h					

1. CSR.SAT and SSR.M1 unchanged by operation

**Example 2**

CMPYR .M2X B0,A1,B2

**Before instruction**
**4 cycles after instruction<sup>1</sup>**

B0	<table border="1"><tr><td>7FFF 7FFFh</td></tr></table>	7FFF 7FFFh	B2	<table border="1"><tr><td>7FFF 0000h</td></tr></table>	7FFF 0000h
7FFF 7FFFh					
7FFF 0000h					
A1	<table border="1"><tr><td>7FFF 8000h</td></tr></table>	7FFF 8000h			
7FFF 8000h					

1. CSR.SAT and SSR.M2 unchanged by operation

**Example 3**

CMPYR .M1 A0,A1,A2

**Before instruction**
**4 cycles after instruction**

A0	<table border="1"><tr><td>8000 8000h</td></tr></table>	8000 8000h	A2	<table border="1"><tr><td>0000 7FFFh</td></tr></table>	0000 7FFFh
8000 8000h					
0000 7FFFh					
A1	<table border="1"><tr><td>8000 8000h</td></tr></table>	8000 8000h			
8000 8000h					
CSR	<table border="1"><tr><td>0001 0100h</td></tr></table>	0001 0100h	CSR <sup>1</sup>	<table border="1"><tr><td>0001 0300h</td></tr></table>	0001 0300h
0001 0100h					
0001 0300h					
SSR	<table border="1"><tr><td>0000 0000h</td></tr></table>	0000 0000h	SS <sup>1</sup>	<table border="1"><tr><td>0000 0010h</td></tr></table>	0000 0010h
0000 0000h					
0000 0010h					

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

**Example 4**

CMPYR .M2 B0,B1,B2

**Before instruction**
**4 cycles after instruction**

B0	<table border="1"><tr><td>8000 8000h</td></tr></table>	8000 8000h	B2	<table border="1"><tr><td>0001 7FFFh</td></tr></table>	0001 7FFFh
8000 8000h					
0001 7FFFh					
B1	<table border="1"><tr><td>8000 8001h</td></tr></table>	8000 8001h			
8000 8001h					
CSR	<table border="1"><tr><td>0001 0100h</td></tr></table>	0001 0100h	CSR <sup>1</sup>	<table border="1"><tr><td>0001 0300h</td></tr></table>	0001 0300h
0001 0100h					
0001 0300h					

## 4.60 CMPYR

## Chapter 4—Instruction Descriptions

SSR      

00000000h
-----------

SSR<sup>1</sup>

00000020h
-----------

1. CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

## 4.61 CMPYR1

Complex Multiply Two Pairs, Signed, Packed 16-Bit With Rounding

**Syntax** **CMPYR1** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	30	29	28	27	dst					src2					src1	
0	0	0	1													
5					5					5					5	
src1	x	0	0	1	1	0	0	1	1	0	0	s	p		1	1
5		1													1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.M1, .M2
<i>src2</i>	<i>xs2</i>	
<i>dst</i>	<i>s2</i>	

**Description**

Performs two dot-products between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded with saturation to 31 bits, shifted, packed and written to a 32-bit register.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The intermediate result is rounded by adding  $2^{14}$  to it. This value is shifted left by 1 with saturation. The 16 most-significant bits of the shifted value are written to the upper half of *dst*.

The product of the upper halfword of *src1* and the lower halfword of *src2* is added to the product of the lower halfword of *src1* and the upper halfword of *src2*. The intermediate result is rounded by adding  $2^{14}$  to it. This value is shifted left by 1 with saturation. The 16 most-significant bits of the shifted value are written to the lower half of *dst*.

If either result saturates in the rounding or shifting process, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

This instruction executes unconditionally.

**Execution**

```
sat((lsb16(src1) × msb16(src2)) + (msb16(src1) × lsb16(src2))) → tmp_e
msb16(sat((tmp_e + 00004000h) << 1)) → lsb16(dst)
sat((msb16(src1) × msb16(src2)) - (lsb16(src1) × lsb16(src2))) → tmp_o
msb16(sat((tmp_e + 00004000h) << 1)) → msb16(dst)
```

**Instruction Type**

Four-cycle

**Delay Slots**

3

**See Also**

[CMPY](#), [CMPYR](#), [DOTP2](#), [DOTPN2](#)

**Examples**

[Example 1](#)

CMPYR1 .M1 A0,A1,A2

<b>Before instruction</b>		<b>4 cycles after instruction<sup>1</sup></b>	
A0	0800 0400h	A2	0080 0068h
A1	0900 0200h		

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 2

CMPYR1 .M2X B0,A1,B2

<b>Before instruction</b>		<b>4 cycles after instruction</b>	
B0	7FFF 7FFFh	B2	7FFFFFFFh
A1	7FFF 8000h		
CSR	00010100h	CSR <sup>1</sup>	00010300h
SSR	00000000h	SSR <sup>1</sup>	00000020h

1. CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

### Example 3

CMPYR1 .M1 A0,A1,A2

<b>Before instruction</b>		<b>4 cycles after instruction</b>	
A0	8000 8000h	A2	00007FFFh
A1	8000 8000h		
CSR	00010100h	CSR <sup>1</sup>	00010300h
SSR	00000000h	SSR <sup>1</sup>	00000010h

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

### Example 4

CMPYR1 .M2 B0,B1,B2

<b>Before instruction</b>		<b>4 cycles after instruction</b>	
B0	C000 C000h	B2	00017FFFh

B1	8000 8001h		
CSR	0001 0100h	CSR <sup>1</sup>	0001 0300h
SSR	0000 0000h	SSR <sup>1</sup>	0000 0020h

1. CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

## 4.62 CMPYSP

Single Precision Complex Floating Point Multiply

**Syntax**    **CMPYSP (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	0	0	0	0	0	s	p	

5                        5                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,qwdst	.M1 or .M2	11110

**Description**    This instruction performs a complex multiply of two Single Precision Floating-Point numbers in a register pair giving a 128-bit output.

- The product of the lower word of src1 and the upper word of src2 to is placed into dst\_0.
- The product of the lower word of src1 and the lower word of src2 is negated and placed into dst\_1.
- The product of the upper word of src1 and the lower word of src2 to is placed into dst\_2.
- The product of the upper word of src1 and the upper word of src2 to is placed into dst\_3.

Special Cases:

- If one source is SNaN or QNaN, the result is a signed NaN\_out and the NANn bit is set. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the XOR to the input signs.
- Signed infinity multiplied by signed infinity or a normalized number (other than signed zero) returns signed infinity. Signed infinity multiplied by signed zero (or denormal) returns a signed NaN\_out and sets the INVAL bit.
- If one or both source are signed zero, the result is signed zero unless the other source is a NaN or signed infinity, in which case the result is signed NaN\_out.
- If signed zero is multiplied by signed infinity, the result is signed NaN\_out and the INVAL bit is set.
- A denormalized source is treated as signed zero and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, OR signed zero. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
- If rounding is performed, the INEX bit is set.

**Execution**

```

src1_e x src2_o -> dst_0
-(src1_e x src2_e) -> dst_1
src1_o x src2_e -> dst_2
src1_o x src2_o -> dst_3

```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** MPYSP, CMPY

**Example**

```

FMCR== 0x00000000
A9 == 0xc0200000
A8 == 0x4059999a
A7 == 0x4109999a
A6 == 0x416b3333
CMPYSP .M A9:A8,A7:A6,A3:A2:A1:A0
A3 == 0xc1ac0000
A2 == 0xc2130000
A1 == 0xc247eb85
A0 == 0x41e9eb86

FMCR= 0x00000080

FMCR== 0x00000000
A9 == 0xc0200000
A8 == 0x40600000
A7 == 0x40500000
A6 == 0x42510000
CMPYSP .M A9:A8,A7:A6,A3:A2:A1:A0
A3 == 0xc1020000
A2 == 0xc302a000
A1 == 0xc336e000
A0 == 0x41360000

FMCR= 0x00000000

FMCR== 0x00000000
A9 == 0x7fc00000
A8 == 0x42510000
A7 == 0x40600000
A6 == 0xc0200000
CMPYSP .M A9:A8,A7:A6,A3:A2:A1:A0
A3 == 0x7fffffff
A2 == 0xffffffff
A1 == 0x4302a000
A0 == 0x4336e000

FMCR= 0x00000001

FMCR== 0x00000000
A9 == 0xffc00000
A8 == 0x42510000
A7 == 0x40600000
A6 == 0xc0200000
CMPYSP .M A9:A8,A7:A6,A3:A2:A1:A0
A3 == 0xffffffff
A2 == 0x7fffffff
A1 == 0x4302a000
A0 == 0x4336e000

FMCR= 0x00000001

FMCR== 0x00000000
B9 == 0x7fc00000
B8 == 0x42510000
B3 == 0x7f900000
B2 == 0xc0200000
CMPYSP .M B9:B8,B3:B2,B7:B6:B5:B4
B7 == 0x7fffffff
B6 == 0xffffffff
B5 == 0x4302a000
B4 == 0x7fffffff

```

```

FMCR= 0x00130000
FMCR== 0x00000000
B9 == 0xff900033
B8 == 0x7f800000
B3 == 0x7fc00000
B2 == 0xc0200000
CMPYSP .M B9:B8,B3:B2,B7:B6:B5:B4
B7 == 0xffffffff
B6 == 0xffffffff
B5 == 0x7f800000
B4 == 0x7fffffff

FMCR= 0x00330000
FMCR== 0x00000000
B9 == 0x7f800000
B8 == 0x00802000
B3 == 0x7fc00000
B2 == 0xc0200000
CMPYSP .M B9:B8,B3:B2,B7:B6:B5:B4
B7 == 0xffffffff
B6 == 0xff800000
B5 == 0x01202800
B4 == 0x7fffffff

FMCR= 0x00220000
FMCR== 0x00000000
B9 == 0x7f800000
B8 == 0x00802000
B3 == 0x00000000
B2 == 0xc0200000
CMPYSP .M B9:B8,B3:B2,B7:B6:B5:B4
B7 == 0xffffffff
B6 == 0xff800000
B5 == 0x01202800
B4 == 0x00000000

FMCR= 0x00300000
FMCR== 0x00000000
B9 == 0x00000000
B8 == 0x00802000
B3 == 0x7f800000
B2 == 0x40600000
CMPYSP .M B9:B8,B3:B2,B7:B6:B5:B4
B7 == 0xffffffff
B6 == 0x00000000
B5 == 0x81603800
B4 == 0x7f800000

FMCR= 0x00300000

```

## 4.63 CROT270

Complex Rotate By 270 Degrees, Signed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    **CROT270** (.unit) *src2, dst*

unit = .L1 or .L2,

**Opcode**    Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	5	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dst	.L1 or .L2	01010

**Description**    Returns the 270 degree rotation of the input complex number. This is the same as multiplying the number by -j.

The input format is 2 packed signed 16-bit numbers, bits 31 through 16 are the real portion of the number, and bits 15 through 0 are the imaginary part.

The real input is returned as the imaginary output, and the imaginary portion is negated with saturation, and returned as the real portion. E.g.,

```
C(31 downto 16) = A(15 downto 0)
C(15 downto 0) = sat( -A(31 downto 16) )
```

**Execution**

```
if(cond) {
  lsb16(src1) -> msb16(dst)
  sat(-msb16(src1)) -> lsb16(dst)
}
else nop
```

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [DCROT270, CROT90](#)

**Example**

```
A0 == 0x12345678
CROT270 .L A0,A15
A15 == 0x5678EDCC
```

## 4.64 CROT90

Complex Rotate By 90 Degrees, Signed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax** **CROT90** (.unit) *src2, dst*

unit = .L1 or .L2,

**Opcode** Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	5	5	4	3	2	1	0
creg	z	dst		src2		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p		

3                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dst	.L1 or .L2	01001

**Description** Returns the 90 degree rotation of the input complex number. This is the same as multiplying the number by j.

The input format is 2 packed signed 16-bit numbers, bits 31 through 16 are the real portion of the number, and bits 15 through 0 are the imaginary part.

The real input is returned as the imaginary output, and the imaginary portion is negated with saturation, and returned as the real portion. E.g.,

```
C(31 downto 16) = sat( - A(15 downto 0) )
C(15 downto 0) = A(31 downto 16)
```

**Execution**

```
if(cond) {
sat(-lsb16(src1)) -> msb16(dst)
msb16(src1) -> lsb16(dst)
}
else nop
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DCROT90](#), [CROT270](#)

**Example**

```
A0 == 0x12345678
CROT90 .L A0,A15
A15 == 0xA9881234
```

## 4.65 DADD

2-Way SIMD Addition, Packed Signed 32-bit

**Syntax**    **DADD (.unit) src1, src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode**    Opcode for .S Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	opfield	1	0	0	0	s	p		
3			5		5		5		6								

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	scst5,xdwop2,dwdst	.S1 or .S2	100001

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	0	0	0	s	p	
				5			5		5		5							

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	000111

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0	
creg	z		dst		src2		src1	x	opfield	1	1	0	s	p			
3			5		5		5			7							

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	scst5,xdwop2,dwdst	.L1 or .L2	0100010

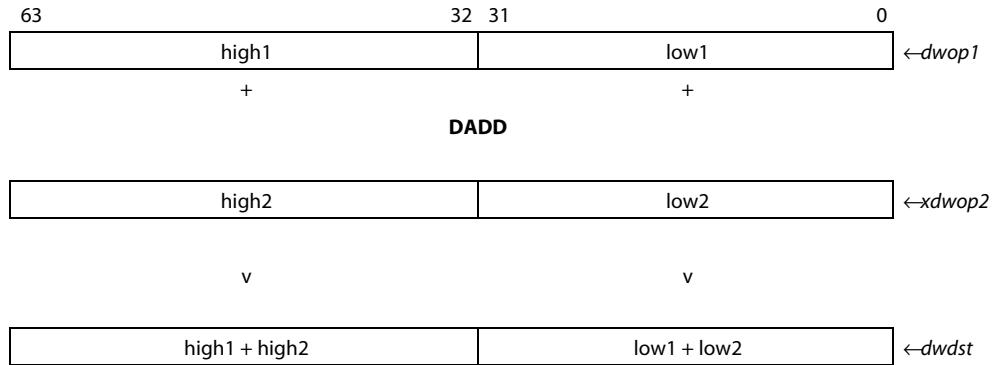
**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	1	0	s	p	
				5			5		5		7						

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0000011

**Description**

The DADD instruction performs two 32-bit additions of the packed 32-bit numbers contained in the two source register pairs. The addition results are returned as two 32-bit results packed into *dwdst*.

**Execution**

```
if(cond) {
  src1_e + src2_e -> dst_e
  src1_o + src2_o -> dst_o
}
else nop
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [ADD](#), [DADD2](#)

**Example**

```
A1 == 0x00000010
A0 == 0x00050011
DADD .L -16,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00050001
```

```
A1 == 0x44444444
A0 == 0x7fffffff
A3 == 0xcccccc444
A2 == 0x00000001
DADD .L A1:A0,A3:A2,A15:A14
A15 == 0x11110888
A14 == 0x80000000
```

## 4.66 DADD2

4-Way SIMD Addition, Packed Signed 16-bit

**Syntax**    **DADD2 (.unit) src1, src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	0	0	0	s	p	

5                        5                        5                        6

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	000001

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

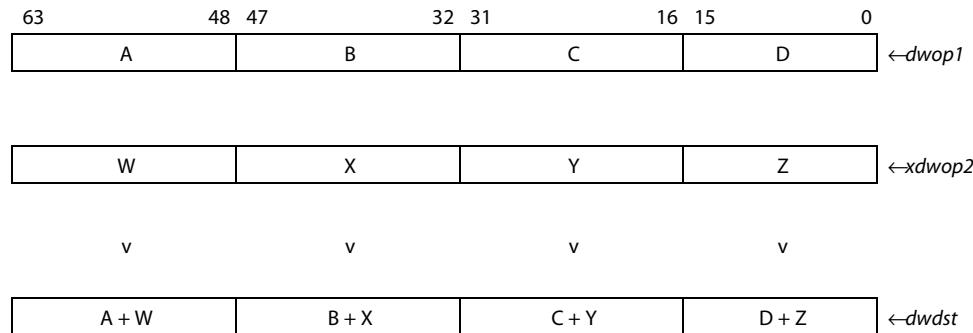
31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	1	0	s	p	

5                        5                        5                        7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0000101

**Description**

The DADD2 instruction performs four 16-bit additions of the packed 16-bit numbers contained in the two 64-bit wide source registers. The addition results are returned as four 16-bit results packed into *dst*.

**Execution**

```

lsb16(src1_e) + lsb16(src2_e) -> lsb16(dst_e)
lsb16(src1_o) + lsb16(src2_o) -> lsb16(dst_o)
msb16(src1_e) + msb16(src2_e) -> msb16(dst_e)
msb16(src1_o) + msb16(src2_o) -> msb16(dst_o)

```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DADD](#), [DSADD](#), [DSADD2](#), [ADD2](#)

**Example**

```

A1 == 0x44444444
A0 == 0x7fff0002
A3 == 0xcccccc444
A2 == 0x7ff00005
DADD2 .L A1:A0,A3:A2,A15:A14
A15 == 0x11100888
A14 == 0xffffef0007

```

## 4.67 DADDSP

2-Way SIMD Single Precision Floating Point Addition

**Syntax**    **DADDSP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	1	0	s	p	

5                        5                        5                        7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0111100

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	opfield	1	0	0	0	s	p	

5                        5                        5                        6

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	101100

**Description**    Performs a SIMD single-precision floating point ADD on the pairs of numbers in dwop1 and xdwop2.

The values in even registers of dwop1 and xdwop2 are added. The result is placed in the even register in the destination register pair dwdst. The values in odd registers of dwop1 and xdwop2 are added. The result is placed in the even register in the destination register pair dwdst.

The following are equivalent:

DADDSP A1:A0, A3:A2, A5:A4

and

```

FADDSP A1, A3, A5
|| FADDSP A0, A2, A4

```

**Execution**    `src1_e + src2_e -> dst_e`  
`src1_o + src2_o -> dst_o`

**Instruction Type**    3-cycle

**Delay Slots**    2

**Functional Unit Latency**    1

**See Also**    [ADDDP](#), [SUBDP](#), [SUBSP](#), [FADDDP](#), [FSUBDP](#), [FSUBSP](#)

**Example**

```

A3 == 0x41200000
A2 == 0xC1200000
A5 == 0x42300000
A4 == 0x42300000
DADDSP .L A3:A2,A5:A4,A1:A0
A1 == 0x42580000
A0 == 0x42080000

; [10] + [44] -> [54]; [-10] + [44] -> [34];
A3 == 0x41200000
A2 == 0xC1200000
A5 == 0xC2300000
A4 == 0xC2300000
DADDSP .L A3:A2,A5:A4,A1:A0
A1 == 0xC2080000
A0 == 0xC2580000

; [10] + [-44] -> [-34]; [-10] + [-44] -> [-54];

```

## 4.68 DAPYS2

4-Way SIMD Apply Sign Bits to Operand

**Syntax** **DAPYS2 (.unit) src1, src2, dst**

unit = .L1 or .L2,

**Opcode** Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opfield	1	1	0	s	p

5                    5                    5                    7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0111000

**Description** The DAPYS2 instruction uses the sign-bits of the 16-bit packed quantities from dwop1 to conditionally negate the packed 16-bit quantities from xdwop2. If the sign-bit is set, then the corresponding field in xdwop2 will be negated, otherwise it will pass through unchanged. The boundary case of 0x8000 will saturate to 0x7FFF.

The DAPYS2 instruction can be used to return the absolute value of the packed 16-bit quantities by sending the same data on to both operands. For example:

DAPYS .L A1:A0, A1:A0, A9:A8

After execution of this instruction, A9:A8 will contain the four absolute values of the contents of A1:A0

**Execution**

```

if(sn(lsb16(src1_e)))
    sat(-lsb16(src2_e)) -> lsb16(dst_e)
else
    lsb16(src2_e) -> lsb16(dst_e)

if(sn(msb16(src1_e)))
    sat(-msb16(src2_e)) -> msb16(dst_e)
else
    msb16(src2_e) -> msb16(dst_e)

if(sn(lsb16(src1_o)))
    sat(-lsb16(src2_o)) -> lsb16(dst_o)
else
    lsb16(src2_o) -> lsb16(dst_o)

if(sn(msb16(src1_o)))
    sat(-msb16(src2_o)) -> msb16(dst_o)
else
    msb16(src2_o) -> msb16(dst_o)

```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [ABS2](#)

**Example**

```
A1 == 0x44444444
A0 == 0x7fffffff
A3 == 0xcccccc444
A2 == 0x00000001
DAPYS2 .L A1:A0,A3:A2,A15:A14
A15 == 0xcccccc444
A14 == 0x0000ffff
```

## 4.69 DAVG2

## 4-Way SIMD Average, Signed, Packed 16-bit

**Syntax**      **DAVG2** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**      Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	0		opfield		1	1	0	0	s	p
			5			5		5				5							

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	10011

**Description** The DAVG2 instruction performs an averaging operation on packed 16-bit data. For each pair of signed 16-bit values provided in *dwop1* and *xdwop2*, DAVG2 calculates the average of the two values as an signed 16-bit quantity in the corresponding position of *dwdst*.

The averaging and rounding operation itself is performed by adding 1 to the sum of the two 16 bit numbers. The result is then right-shifted by 1 to produce the final 16-bit result.

63	48	47	32	31	16	15	0	
i3		i2		i1		i0		←op1

j3	j2	j1	j0	↔op2
v	v	v	v	
$(i3 + j3 + 1) \gg 1$	$(i2 + j2 + 1) \gg 1$	$(i1 + j1 + 1) \gg 1$	$(i0 + j0 + 1) \gg 1$	↔dst

```

Execution    (((lsb16(src1_e) + lsb16(src2_e) + 1)>>1) -> lsb16(dst_e))
    (((msb16(src1_e) + msb16(src2_e) + 1)>>1) -> msb16(dst_e))
    (((lsb16(src1_o) + lsb16(src2_o) + 1)>>1) -> lsb16(dst_o))
    (((msb16(src1_o) + msb16(src2_o) + 1)>>1) -> msb16(dst_o))

```

***Instruction Type***      2-cycle

## *Delay Slots*

### ***Functional Unit Latency***

**See Also** DAVGNRU4

```

Example    A3 == 0xc001c001
              A2 == 0x40004000
              A1 == 0x3fff3fff
              A0 == 0x40004000
              DAVG2 .M A3:A2,A1:A0,A7:A6
              A7 == 0x00000000
              A6 == 0x40004000

              A3 == 0x3fff3fff
              A2 == 0x0000c000
              A1 == 0x3fff3fff
              A0 == 0x0000c000

```

```
DAVG2 .M A3:A2,A1:A0,A7:A6
A7 == 0x3fff3fff
A6 == 0xc000c000
```

## 4.70 DAVGNR2

4-Way SIMD Average Without Rounding, Signed Packed 16-bit

**Syntax** **DAVGNR2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	0		opfield	1	1	0	0	s	p

5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwo1,xdwop2,dwdst	.M1 or .M2	10001

**Description**

The DAVGNR2 instruction performs an averaging operation on packed 16-bit data. For each pair of signed 16-bit values provided in dwop1 and xdwop2, DAVGNR2 calculates the average of the two values as an signed 16-bit quantity in the corresponding position of dwdst.

The averaging operation itself is performed without rounding: first we compute the sum of the two 16-bit numbers being averaged. The result is then right-shifted by 1 and sign extended to produce a 16-bit result. The intermediate results are kept at full precision internally, so that no overflow conditions exist.

63	48	47	32	31	16	15	0
i3		i2		i1		i0	
							←op1
j3		j2		j1		j0	
v		v		v		v	
(i3 + j3) >> 1		(i2 + j2) >> 1		(i1 + j1) >> 1		(i0 + j0) >> 1	←dst

**Execution**

```
((lsb16(src1_e) + lsb16(src2_e))>>1) -> lsb16(dst_e)
((msb16(src1_e) + msb16(src2_e))>>1) -> msb16(dst_e)
((lsb16(src1_o) + lsb16(src2_o))>>1) -> lsb16(dst_o)
((msb16(src1_o) + msb16(src2_o))>>1) -> msb16(dst_o)
```

**Instruction Type** 2-cycle

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [DAVGNRU4](#)

**Example**

```
A3 == 0xc001c001
A2 == 0x40004000
A1 == 0x3fff3fff
A0 == 0x40004000
DAVGNR2 .M A3:A2,A1:A0,A7:A6
A7 == 0x00000000
A6 == 0x40004000
```

```
A3 == 0x3fff3fff
A2 == 0xc000c000
```

```
A1 == 0x3fff3fff
A0 == 0xc000c000
DAVGNR2 .M A3:A2,A1:A0,A7:A6
A7 == 0x3fff3fff
A6 == 0xc000c000
```

## 4.71 DAVGNRU4

8-Way SIMD Average Without Rounding, Unsigned Packed 8-bit

**Syntax** DAVGNRU4 (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	0	opfield	1	1	0	0	s	p				

5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwo1,xdwop2,dwdst	.M1 or .M2	10000

**Description** The DAVGNRU4 instruction performs an averaging operation between unsigned packed 8-bit quantities. The values in *dwo1* and *xdwop2* are treated as unsigned packed 8-bit quantities, and the results are written in an unsigned packed 8-bit format.

For each pair of 8-bit quantities in *dwo1* and *xdwop2*, the average of the unsigned 8-bit value from *dwo1* and the unsigned 8-bit value from *xdwop2* is calculated to produce an unsigned 8-bit result. The result is placed in the corresponding position in *dwdst*.

The averaging operation is performed without rounding -- the two numbers are merely added together and the result shifted right by 1.

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0			
i7		i6		i5		i4		i3		i2		i1		i0		←dwo1		
j7		j6		j5		j4		j3		j2		j1		j0		←xdwop2		
v		v		v		v		v		v		v		v				
(i7+j7) >>1	(i6+j6) >>1	(i5+j5) >>1	(i4+j4) >>1	(i3+j3) >>1	:	(i2+j2) >>1	(i1+j1) >>1	i0+j0) >>1							←dwdst			

**Execution**

```
((ubyte0(src1_e) + ubyte0(src2_e))>>1) -> ubyte0(dst_e)
((ubyte1(src1_e) + ubyte1(src2_e))>>1) -> ubyte1(dst_e)
((ubyte2(src1_e) + ubyte2(src2_e))>>1) -> ubyte2(dst_e)
((ubyte3(src1_e) + ubyte3(src2_e))>>1) -> ubyte3(dst_e)
((ubyte0(src1_o) + ubyte0(src2_o))>>1) -> ubyte0(dst_o)
((ubyte1(src1_o) + ubyte1(src2_o))>>1) -> ubyte1(dst_o)
((ubyte2(src1_o) + ubyte2(src2_o))>>1) -> ubyte2(dst_o)
((ubyte3(src1_o) + ubyte3(src2_o))>>1) -> ubyte3(dst_o)
```

**Instruction Type** 2-cycle

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** AVGU4, DAVG2

**Example**

A3 == 0x1A2E5F4E
A2 == 0xFBFCFDDE
A1 == 0x9EF26E3F
A0 == 0x03020201

```
DAVGNRU4 .M A3:A2,A1:A0,A15:A14
A15 == 0x5C906646
A14 == 0x7f7f7f7f
```

## 4.72 DAVGU4

8-Way SIMD Average, Unsigned Packed 8-bit

**Syntax** **DAVGU4 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	0	opfield	1	1	0	0	s	p				

5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwp1,xdwop2,dwdst	.M1 or .M2	10010

**Description** The DAVGU4 instruction performs an averaging operation between unsigned packed 8-bit quantities and rounding. The values in *dwp1* and *xdwop2* are treated as unsigned packed 8-bit quantities, and the results are written in an unsigned packed 8-bit format.

For each pair of 8-bit quantities in *dwp1* and *xdwop2*, the average of the unsigned 8-bit value from *dwp1* and the unsigned 8-bit value from *xdwop2* is calculated to produce an unsigned 8-bit result. The result is placed in the corresponding position in *dwdst*.

The averaging and rounding operation itself is performed by adding 1 to the sum of the two unsigned 8-bit numbers being averaged. The result is then right-shifted by 1 to produce a the final unsigned 8-bit result. The intermediate results are kept at full precision internally, so that no overflow conditions exist.

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0			
i7		i6		i5		i4		i3		i2		i1		i0		←dwp1		
j7		j6		j5		j4		j3		j2		j1		j0		←xdwop2		
v		v		v		v		v		v		v		v				
(i7+j7+1)>>1	(i6+j6+1)>>1	(i5+j5+1)>>1	(i4+j4+1)>>1	(i3+j3+1)>>1	(i2+j2+1)>>1	(i1+j1+1)>>1	(i0+j0+1)>>1									←dwdst		

**Execution**

```
((ubyte0(src1_e) + ubyte0(src2_e) + 1)>>1) -> ubyte0(dst_e)
((ubyte1(src1_e) + ubyte1(src2_e) + 1)>>1) -> ubyte1(dst_e)
((ubyte2(src1_e) + ubyte2(src2_e) + 1)>>1) -> ubyte2(dst_e)
((ubyte3(src1_e) + ubyte3(src2_e) + 1)>>1) -> ubyte3(dst_e)
((ubyte0(src1_o) + ubyte0(src2_o) + 1)>>1) -> ubyte0(dst_o)
((ubyte1(src1_o) + ubyte1(src2_o) + 1)>>1) -> ubyte1(dst_o)
((ubyte2(src1_o) + ubyte2(src2_o) + 1)>>1) -> ubyte2(dst_o)
((ubyte3(src1_o) + ubyte3(src2_o) + 1)>>1) -> ubyte3(dst_o)
```

**Instruction Type** 2-cycle

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [AVGU4](#), [DAVG2](#)

**Example**

```
A3 == 0x1A2E5F4E
A2 == 0xFBFCFDDE
A1 == 0x9EF26E3F
A0 == 0x03020201
DAVGU4 .M A3:A2,A1:A0,A15:A14
A15 == 0x5C906747
A14 == 0x7f7f8080
```

## 4.73 DCCMPY

## 2-Way SIMD Complex Multiply With Conjugate, Packed Complex Signed 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**      **DCCMPY** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcodes**      Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	0		opfield		1	1	0	0	s	p
			5			5		5				5							

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	00011

**Description** This instruction performs two complex multiplies on two pairs of packed complex numbers (16-bit real, 16-bit imaginary). It first performs complex conjugate operation on the two complex numbers on the 2nd operand, then it multiplies two complex numbers in the first register pairs with the corresponding two complex numbers in the second input register pair. The final results are 32-bit complex number (32-bit real, 32-bit imaginary) and placed in the destination register pair.

The DCCMPY instruction is functionally equivalent to following instruction sequence:

```

DAPYS2 src2_h:src2_l, cst_h:cst_l, tmp_h:tmp_l
      ; where cst_h:cst_l contains "0x0000FFFF:0x0000FFFF"
||   CMPY  src1_h, tmp_h, dst_3:dst_2
||   CMPY  src1_l, tmp_l, dst_1:dst_0

```

```

Execution   ((msb16(src1_e) x lsb16(src2_e)) - (lsb16(src1_e) x msb16(src2_e))) -> dst_0
               ((msb16(src1_e) x msb16(src2_e)) + (lsb16(src1_e) x lsb16(src2_e))) -> dst_1
               ((msb16(src1_o) x lsb16(src2_o)) - (lsb16(src1_o) x msb16(src2_o))) -> dst_2
               ((msb16(src1_o) x msb16(src2_o)) + (lsb16(src1_o) x lsb16(src2_o))) -> dst_3

```

**Instruction Type** 4-cycle

## *Delay Slots*

### ***Functional Unit Latency***

**See Also** [DCMPY](#)

```

Example      A5 == 0x0009FFFF
                A4 == 0x0009FFFF
                A7 == 0xFFFFF0007
                A6 == 0xFFFFF0007
                DCCMPY .M A5:A4,A7:A6,A15:A14:A13:A12
                A15 == 0xFFFFFFFFE9
                A14 == 0xFFFFFFFFC3
                A13 == 0xFFFFFFFFE9
                A12 == 0xFFFFFFFFC3

```

## 4.74 DCCMPYR1

2-Way SIMD Complex Multiply With Conjugate and Rounding, Packed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax** **DCCMPYR1** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	0		opfield	1	1	0	0	s	p

5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwp1,xdwop2,dwdst	.M1 or .M2	01110

**Description** This instruction performs two complex multiplies on two pairs of packed 16-bit complex numbers. It first performs complex conjugate operation on the two 16-bit complex numbers on the 2nd operand, then it multiplies two complex numbers in the first register pairs with the corresponding two complex numbers in the second input register pair. The final results are 16-bit complex number and placed in the destination register pair.

The DCCMPYR1 instruction is approximately equivalent to following instruction sequence:

```
DAPYS2 src2_h:src2_l, tmp_h:tmp_l
CMPYR1 src1_h, tmp_h, dst_h
|| CMPYR1 src1_l, tmp_l, dst_l
```

**Execution**

```
((msb16(src1_e) x lsb16(src2_e))-(lsb16(src1_e) x msb16(src2_e)))-> tmp0_e
((msb16(src1_e) x msb16(src2_e))+(lsb16(src1_e) x lsb16(src2_e)))-> tmp0_o
((msb16(src1_o) x lsb16(src2_o))-(lsb16(src1_o) x msb16(src2_o)))-> tmp1_e
((msb16(src1_o) x msb16(src2_o))+(lsb16(src1_o) x lsb16(src2_o)))-> tmp1_o
msb16(sat((tmp0_e + 00004000h) << 1)) -> lsb16(dst_e)
msb16(sat((tmp0_o + 00004000h) << 1)) -> msb16(dst_e)
msb16(sat((tmp1_e + 00004000h) << 1)) -> lsb16(dst_o)
msb16(sat((tmp1_o + 00004000h) << 1)) -> msb16(dst_o)
```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [DCMPYR1](#), [CMPYR1](#)

**Example**

```
CSR == 0x00000000 ; 4
A3 == 0x00000000
A2 == 0x00000000
A1 == 0x00000000
A0 == 0x00000000
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000

CSR= 0x00000000
```

```

CSR == 0x00000000 ; 4
A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80007fff
A0 == 0x80007fff
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00017fff
A14 == 0x00017fff

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80008000
A0 == 0x80008000
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x7fff0000
A14 == 0x7fff0000

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0x08000400
A2 == 0x09000200
A1 == 0x09000200
A0 == 0x08000400
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00a00028
A14 == 0x00a0ffd8

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x7FFF7FFF
A2 == 0x7FFF8000
A1 == 0x7FFF8000
A0 == 0x7FFF7FFF
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0xffff7fff
A14 == 0xffff8000

CSR= 0x00000200

CSR == 0x00000000 ; 4
A3 == 0xFFFFFFFF
A2 == 0xFFFFFFFF
A1 == 0xFFFFFFFF
A0 == 0xFFFFFFFF
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x55555555
A2 == 0x55555555
A1 == 0x55555555
A0 == 0x55555555
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x71c60000
A14 == 0x71c60000

CSR= 0x00000000

CSR == 0x00000000 ; 4
A3 == 0x01234567
A2 == 0x89ABCDEF
A1 == 0x89ABCDEF
A0 == 0x01234567
DCCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0xe3cec049
A14 == 0xe3ce3fb7

CSR= 0x00000000

CSR == 0x00000000 ; 4
B3 == 0x80008000
B2 == 0x80008000
B1 == 0x80007fff

```

```
B0 == 0x80007fff
DCCMPYR1 .M B3:B2,B1:B0,B15:B14 ; added for a & b_path cov.
B15 == 0x00017fff
B14 == 0x00017fff

CSR= 0x00000200

CSR == 0x00000000 ; 4
B3 == 0x80008000
B2 == 0x80008000
B1 == 0x80008000
B0 == 0x80008000
DCCMPYR1 .M B3:B2,B1:B0,B15:B14
B15 == 0x7fff0000
B14 == 0x7fff0000

CSR= 0x00000200
```

## 4.75 DCMPEQ2

2-Way SIMD Compare If Equal, Packed 16-bit

**Syntax**    **DCMPEQ2** (.unit) *src1, src2, dst*

unit = .S1 or .S2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

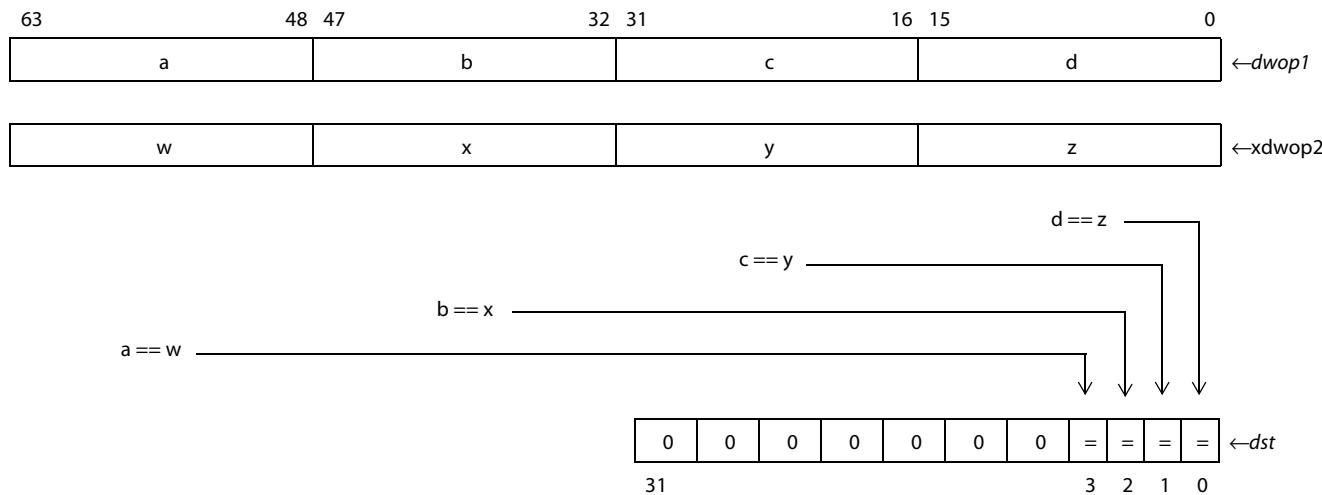
31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x		opfield	1	0	0	0	s	p	

5                        5                        5                        6

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwp1,xdwop2,dst	.S1 or .S2	011101

**Description**

The DCMPEQ2 instruction performs equality comparisons on packed 16-bit data. Each 16-bit value in *op1* is compared against the corresponding 16-bit value in *xop2*, returning a 1 if equal or 0 if not equal. The equality results are packed into the four least-significant bits of *dwdst*.



**Execution**

```

if(lsb16(src1_e) == lsb16(src2_e)) 1 -> dst0
else 0 -> dst0
if(msb16(src1_e) == msb16(src2_e)) 1 -> dst1
else 0 -> dst1
if(lsb16(src1_o) == lsb16(src2_o)) 1 -> dst2
else 0 -> dst2
if(msb16(src1_o) == msb16(src2_o)) 1 -> dst3
else 0 -> dst3

```

---

<b>Instruction Type</b>	Single-cycle
<b>Delay Slots</b>	0
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">DCMPGT2</a> , <a href="#">DCMPEQ4</a> , <a href="#">DXPND2</a>

**Example**

```

A1 == 0x00000000
A0 == 0x3fff0000
A3 == 0x00000000
A2 == 0x00003fff
DCMPEQ2 .S A1:A0,A3:A2,A15
A15 == 0x0000000C

A1 == 0x3ffffc000
A0 == 0x0000c001
A3 == 0x3ffffc000
A2 == 0x3ffffc001
DCMPEQ2 .S A1:A0,A3:A2,A15
A15 == 0x0000000d

A1 == 0xc0010000
A0 == 0x44444444
A3 == 0xc0010110
A2 == 0xcccc4444
DCMPEQ2 .S A1:A0,A3:A2,A15
A15 == 0x00000009

```

## 4.76 DCMPEQ4

4-Way SIMD Compare If Equal, Packed 8-bit

**Syntax**    **DCMPEQ4** (.unit) *src1, src2, dst*

unit = .S1 or .S2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

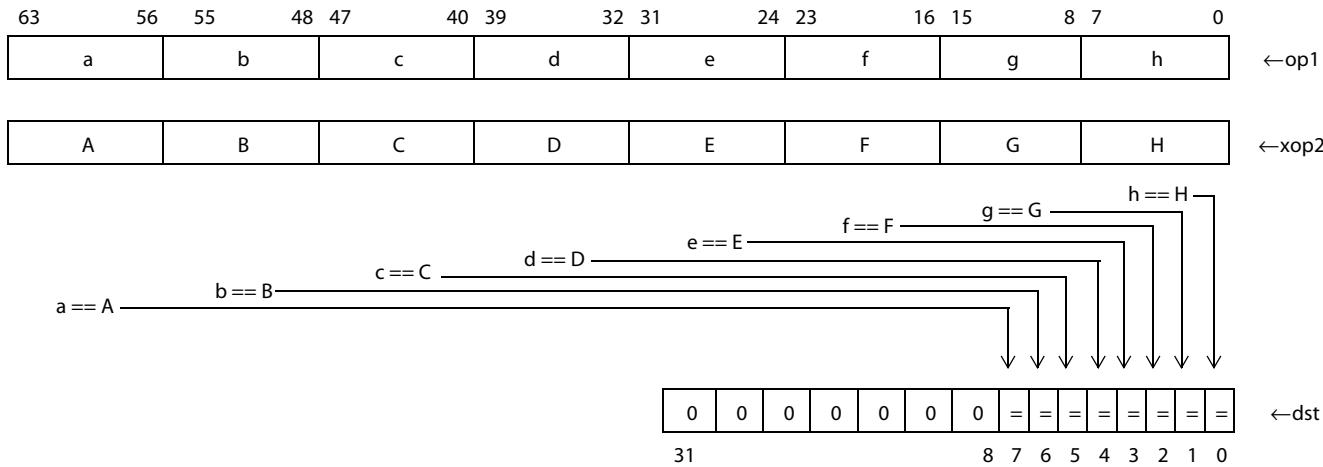
31	30	29	28	27	23	22	18	17	13	12	11	6	5	2	1	0
0	0	0	1	dst		src2		src1	x		opfield		1 0 0 0	s	p	

5                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dst	.S1 or .S2	011100

**Description**    The DCMPEQ4 instruction performs equality comparisons on packed 8-bit data. Each 8-bit value in *dwop1* is compared against the corresponding 8-bit value in *xdwop2*, returning a 1 if equal or 0 if not equal. The equality results are packed into the eight least-significant bits of *dst*.

The 8-bit values in each input are numbered from 0..7 starting with the least-significant byte, working towards the most- significant byte. The comparison results for byte 0 are written to bit 0 of the result. Likewise, the results for byte 1..7 are written to bits 1..7 of the result, respectively, as shown in the diagram below.



```
Execution    if(sbyte0(src1_e) == sbyte0(src2_e)) 1 -> dst0
                  else 0 -> dst0
                  if(sbyte1(src1_e) == sbyte1(src2_e)) 1 -> dst1
                  else 0 -> dst1
                  if(sbyte2(src1_e) == sbyte2(src2_e)) 1 -> dst2
                  else 0 -> dst2
                  if(sbyte3(src1_e) == sbyte3(src2_e)) 1 -> dst3
                  else 0 -> dst3
                  if(sbyte0(src1_o) == sbyte0(src2_o)) 1 -> dst4
                  else 0 -> dst4
                  if(sbyte1(src1_o) == sbyte1(src2_o)) 1 -> dst5
                  else 0 -> dst5
                  if(sbyte2(src1_o) == sbyte2(src2_o)) 1 -> dst6
                  else 0 -> dst6
                  if(sbyte3(src1_o) == sbyte3(src2_o)) 1 -> dst7
                  else 0 -> dst7
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DCMPEQ4](#), [DCMPGTU4](#), [DXPND4](#)

**Example**

```
A1 == 0xabfffff
A0 == 0xff00ff00
A3 == 0x00ffff00
A2 == 0x00ffff00
DCMPEQ4 .S A1:A0,A3:A2,A15
A15 == 0x00000063

A1 == 0xff0000ff
A0 == 0x023a4e1f
A3 == 0x002e3aff
A2 == 0x023b4e1f
DCMPEQ4 .S A1:A0,A3:A2,A15
A15 == 0x0000001b

A1 == 0x44444444
A0 == 0xff918aee
A3 == 0xcccccc444
A2 == 0x01665a1e
DCMPEQ4 .S A1:A0,A3:A2,A15
A15 == 0x00000010
```

## 4.77 DCMPGT2

2-Way SIMD Compare If Greater-Than, Packed 16-bit

**Syntax**    **DCMPGT2** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

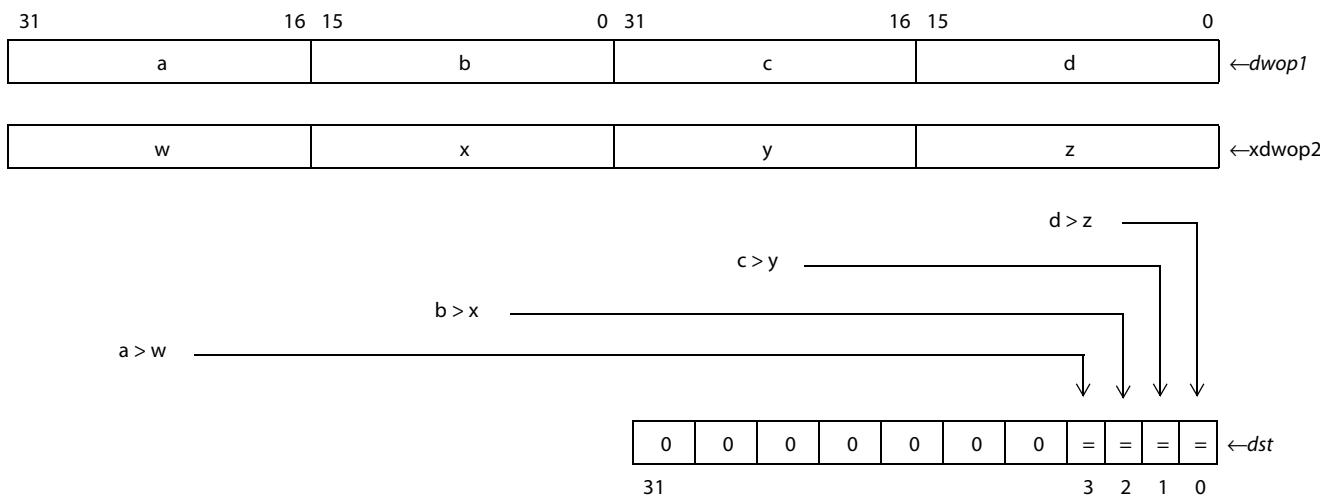
**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23 22	18 17	13	12	11	6 5	2	1	0
0	0	0	1	dst	src2	src1	x		opfield	1 0 0 0	s	p	

5                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dst	.S1 or .S2	010100

**Description**    The DCMPGT2 instruction performs greater-than comparisons on packed 16-bit data. Each signed 16-bit value in *dwop1* is compared against the corresponding signed 16-bit value in *xdwop2*, returning a 1 if the value from *dwop1* is greater than the value from *xdwop2*, or 0 otherwise. The comparison results are packed into the four least-significant bits of *dst*.



**Execution**

```

if(lsb16(src1_e) > lsb16(src2_e)) 1 -> dst0
else 0 -> dst0
if(msb16(src1_e) > msb16(src2_e)) 1 -> dst1
else 0 -> dst1
if(lsb16(src1_o) > lsb16(src2_o)) 1 -> dst2
else 0 -> dst2

```

---

```
if(msb16(src1_o) > msb16(src2_o)) 1 -> dst3
else 0 -> dst3
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [CMPGT](#), [CMPEQ2](#), [CMPEQ4](#), [CMPGTU4](#), [XPND2](#)

**Example**

```
A1 == 0x7fff7fff
A0 == 0x7fff7fff
A3 == 0x7fff7fff
A2 == 0x7fff7fff
DCMPGT2 .S A1:A0,A3:A2,A11
A11 == 0x00000000

A1 == 0x7fff7fff
A0 == 0x7fff7fff
A3 == 0x80008000
A2 == 0x80008000
DCMPGT2 .S A1:A0,A3:A2,A11
A11 == 0x0000000f

A1 == 0x80007fff
A0 == 0x80007fff
A3 == 0x7fff8000
A2 == 0x7fff8000
DCMPGT2 .S A1:A0,A3:A2,A11
A11 == 0x00000005

A1 == 0x7fff8000
A0 == 0x7fff8000
A3 == 0x80007fff
A2 == 0x80007fff
DCMPGT2 .S A1:A0,A3:A2,A11
A11 == 0x0000000a

A1 == 0x80008000
A0 == 0x80008000
A3 == 0x7fff7fff
A2 == 0x7fff7fff
DCMPGT2 .S A1:A0,A3:A2,A11
A11 == 0x00000000
```

4.78 DCMPGTU4

## 4-Way SIMD Compare If Greater-Than, Unsigned Packed 8-bit

**Syntax**      DCMPGTU4 (.unit) *src1*, *src2*, *dst*

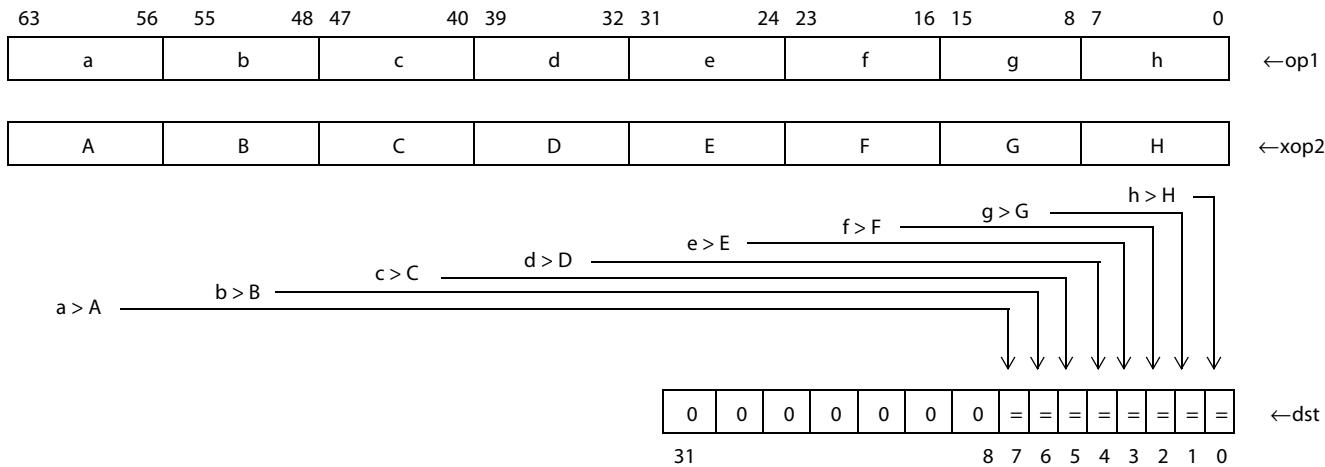
unit = .S1 or .S2

**Opcode**      Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1000	s	p					
			5		5	5		6		4						

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dst	.S1 or .S2	010101

**Description** The DCMPGTU4 instruction performs greater-than comparisons on packed 8-bit data. Each unsigned 8-bit value in *dwop1* is compared against the corresponding unsigned 8-bit value in *xdwop2*, returning a 1 if the value from *op1* is greater than the value in *xop2* or 0 otherwise. The comparison results are packed into the eight least-significant bits of *dst*.



```
Execution    if(ubyte0(src1_e) > ubyte0(src2_e)) 1 -> dst_0  
                  else 0 -> dst_0  
                  if(ubyte1(src1_e) > ubyte1(src2_e)) 1 -> dst_1  
                  else 0 -> dst_1  
                  if(ubyte2(src1_e) > ubyte2(src2_e)) 1 -> dst_2  
                  else 0 -> dst_2  
                  if(ubyte3(src1_e) > ubyte3(src2_e)) 1 -> dst_3
```

```

else 0 -> dst3
if(ubyte0(src1_o) > ubyte0(src2_o)) 1 -> dst4
else 0 -> dst4
if(ubyte1(src1_o) > ubyte1(src2_o)) 1 -> dst5
else 0 -> dst5
if(ubyte2(src1_o) > ubyte2(src2_o)) 1 -> dst6
else 0 -> dst6
if(ubyte3(src1_o) > ubyte3(src2_o)) 1 -> dst7
else 0 -> dst7

```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DCMPEQ4](#), [DCMPGT2](#), [DCMPEQ2](#), [DXPND4](#)

**Example**

```

A1 == 0x000000ff
A0 == 0x000000ff
A3 == 0x000000fe
A2 == 0x000000fe
DCMPGTU4 .S A1:A0,A3:A2,A11
A11 == 0x00000011

A1 == 0x0000ffff
A0 == 0x0000ffff
A3 == 0x0000feff
A2 == 0x0000feff
DCMPGTU4 .S A1:A0,A3:A2,A11
A11 == 0x00000022

A1 == 0xffffffff
A0 == 0xffffffff
A3 == 0xfefffffe
A2 == 0xfefffffe
DCMPGTU4 .S A1:A0,A3:A2,A11
A11 == 0x000000bb

A1 == 0xffffffff
A0 == 0xffffffff
A3 == 0xfefefefe
A2 == 0xfefefefe
DCMPGTU4 .S A1:A0,A3:A2,A11
A11 == 0x000000ff

A1 == 0xabcdefac
A0 == 0xabcdefac
A3 == 0xbcdfaceb
A2 == 0xbcdfaceb
DCMPGTU4 .S A1:A0,A3:A2,A11
A11 == 0x00000022

```

## 4.79 DCMPY

2-Way SIMD Complex Multiply, Packed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    DCMPY (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	2	1	0
0	0	0	1		dst		src2		src1	x	0	opfield		1100	s	p	

5                        5                        5                        5                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	00010

**Description**    This instruction performs two complex multiplies on two pairs of packed complex numbers.

The DCMPY instruction is functionally equivalent to following instruction sequence:

```
CMPY  src1_h, src2_h, dst_3:dst_2
||  CMPY  src1_l, src2_l, dst_1:dst_0
```

**Execution**    sat(lsb16(src1\_e) x msb16(src2\_e))+(msb16(src1\_e) x lsb16(src2\_e))-> dst\_0
(msb16(src1\_e) x msb16(src2\_e))- (lsb16(src1\_e) x lsb16(src2\_e))-> dst\_1
sat(lsb16(src1\_o) x msb16(src2\_o))+(msb16(src1\_o) x lsb16(src2\_o))-> dst\_2
(msb16(src1\_o) x msb16(src2\_o))- (lsb16(src1\_o) x lsb16(src2\_o))-> dst\_3

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [DCCMPY](#)

**Example**

```
A5 == 0x0009FFFE
A4 == 0x0009FFFE
A7 == 0xFFFFF007
A6 == 0xFFFFF007
DCMPY .M A5:A4,A7:A6,A15:A14:A13:A12
A15 == 0x00000005
A14 == 0x00000041
A13 == 0x00000005
A12 == 0x00000041
```

## 4.80 DCMPYR1

2-Way SIMD Complex Multiply With Rounding, Packed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    **DCMPYR1** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	2	1	0
0	0	0	1		dst		src2		src1	x	0	opfield		1100	s	p	

5                        5                        5                        5                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	01101

**Description**    This instruction performs two complex multiplies on two pairs of packed complex numbers.

The CMPYR1 instruction is functionally equivalent to following instruction sequence:

```
CMPYR1    src1_h, src2_h, dst_h
||  CMPYR1    src1_l, src2_l, dst_l
```

**Execution**

```
((msb16(src1_e) x lsb16(src2_e))+(lsb16(src1_e) x msb16(src2_e))) -> tmp0_e
((msb16(src1_e) x msb16(src2_e))-(lsb16(src1_e) x lsb16(src2_e))) -> tmp0_o
((msb16(src1_o) x lsb16(src2_o))+(lsb16(src1_o) x msb16(src2_o))) -> tmp1_e
((msb16(src1_o) x msb16(src2_o))-(lsb16(src1_o) x lsb16(src2_o))) -> tmp1_o
msb16(sat((tmp0_e + 00004000h) << 1)) -> lsb16(dst_e)
msb16(sat((tmp0_o + 00004000h) << 1)) -> msb16(dst_e)
msb16(sat((tmp1_e + 00004000h) << 1)) -> lsb16(dst_o)
msb16(sat((tmp1_o + 00004000h) << 1)) -> msb16(dst_o)
```

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [DCCMPYR1](#), [CMPYR1](#)

**Example**

```
A3 == 0x00000000
A2 == 0x00000000
A1 == 0x00000000
A0 == 0x00000000
DCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000
```

```

A3 == 0x80008000
A2 == 0x80008000
A1 == 0x80008000
A0 == 0x80008000
DCMPYR1 .M A3:A2,A1:A0,A15:A14 ; added for a & b_path cov.
A15 == 0x00007fff
A14 == 0x00007fff

A3 == 0x08000400
A2 == 0x09000200
A1 == 0x09000200
A0 == 0x08000400
DCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00800068
A14 == 0x00800068

A3 == 0x7FFF7FFF
A2 == 0x7FFF8000
A1 == 0x7FFF8000
A0 == 0x7FFF7FFF
DCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x7fffffff
A14 == 0x7fffffff

A3 == 0xFFFFFFFF
A2 == 0xFFFFFFFF
A1 == 0xFFFFFFFF
A0 == 0xFFFFFFFF
DCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x00000000
A14 == 0x00000000

A3 == 0x55555555
A2 == 0x55555555
A1 == 0x55555555
A0 == 0x55555555
DCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x000071c6
A14 == 0x000071c6

A3 == 0x01234567
A2 == 0x89ABCDEF
A1 == 0x89ABCDEF
A0 == 0x01234567
DCMPYR1 .M A3:A2,A1:A0,A15:A14
A15 == 0x1a18bf65
A14 == 0x1a18bf65

```

## 4.81 DCROT270

2-Way SIMD Rotate Complex Number By 270 Degrees, Packed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    **DCROT270** (.unit) *src2, dst*

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	2	1	0
creg	z		dst		src2		opfield	x		0011010110		s	p

3                        5                        5                        5                        10

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xdwop,dwdst	.L1 or .L2	11010

**Description**    Performs two rotate by 270 degree operations on the input vector of complex numbers. Equivalent to executing the CROT270 instruction twice.

DCROT270 A1:A0, A3:A2

is equivalent to executing

```
CROT270 A1, A3
|| CROT270 A0, A2
```

**Execution**

```
if(cond) {
    lsb16(src1_e) -> msb16(dst_e)
    sat(-msb16(src1_e)) -> lsb16(dst_e)
    lsb16(src1_o) -> msb16(dst_o)
    sat(-msb16(src1_o)) -> lsb16(dst_o)
}
else nop
```

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [CROT270](#), [DCROT90](#)

**Example**

```
A1 == 0x12345678
A0 == 0x80005532
DCROT270 .L A1:A0,A15:A14
A15 == 0x5678EDCC
A14 == 0x55327fff
```

## 4.82 DCROT90

2-Way SIMD Rotate Complex Number By 90 Degrees, Packed Complex 16-bit (16-bit Real/16-bit Imaginary)

**Syntax**    DCROT90 (.unit) *src2*, *dst*

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	2	1	0
creg	z		dst		src2		opfield	x		0011010110		s	p

3                         5                         5                         5                         10

Opcode map field used...	For operand type...	Unit	Opfield
src2, dst	xdwop,dwdst	.L1 or .L2	11001

**Description**    Performs two rotate by 90 degrees operations on the input vector of complex numbers. Equivalent to executing the CROT90 instruction twice

DCROT90 A1:A0, A3:A2

is equivalent to executing

```
CROT90 A1, A3
|| CROT90 A0, A2
```

**Execution**

```
if(cond) {
    sat(-lsb16(src1_e)) -> msb16(dst_e)
    msb16(src1_e) -> lsb16(dst_e)
    sat(-lsb16(src1_o)) -> msb16(dst_o)
    msb16(src1_o) -> lsb16(dst_o)
}
else nop
```

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [CROT90](#), [DCROT270](#)

**Example**

```
A1 == 0x12345678
A0 == 0x55328000
DCROT90 .L A1:A0,A15:A14
A15 == 0xA9881234
A14 == 0x7fff5532
```

## 4.83 DDOTP4

Double Dot Product, Signed, Packed 16-Bit and Signed, Packed 8-Bit

**Syntax**    **DDOTP4 (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	30	29	28	27			23	22											18
0	0	0	1		<i>dst</i>				<i>src2</i>										
					5					5									
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
		<i>src1</i>	x	0	1	1	0	0	0	1	1	0	0	s	p		1	1	
			5	1															

Opcode map field used...	For operand type...	Unit
<i>src1</i>	ds2	.M1, .M2
<i>src2</i>	xs4	
<i>dst</i>	dint	

**Description**

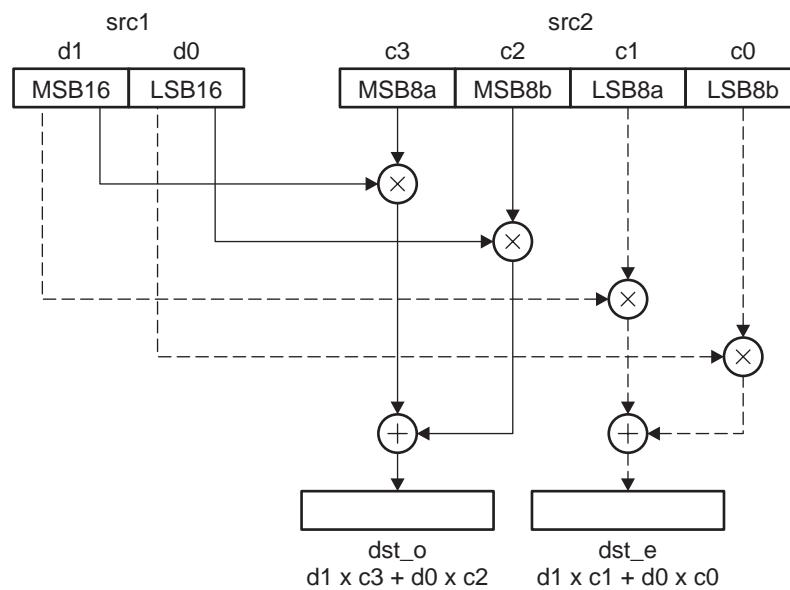
Performs two DOTP2 operations simultaneously.

The lower byte of the lower halfword of *src2* is sign-extended to 16 bits and multiplied by the lower halfword of *src1*. The upper byte of the lower halfword of *src2* is sign-extended to 16 bits and multiplied by the upper halfword of *src1*. The two products are added together and the result is then written to *dst\_e*.

The lower byte of the upper halfword of *src2* is sign-extended to 16 bits and multiplied by the lower halfword of *src1*. The upper byte of the upper halfword of *src2* is sign-extended to 16 bits and multiplied by the upper halfword of *src1*. The two products are added together and the result is then written to *dst\_o*.

There are no saturation cases possible.

This instruction executes unconditionally.



**Execution**

$$(msb16(src1) \times msb8(lsb16(src2))) + (lsb16(src1) \times lsb8(lsb16(src2))) \rightarrow dst\_e$$

$$(msb16(src1) \times msb8(msb16(src2))) + (lsb16(src1) \times lsb8(msb16(src2))) \rightarrow dst\_o$$

**Instruction Type** Four-cycle

**Delay Slots** 3

**Examples** **Example 1**

DDOTP4 .M1 A4,A5,A9:A8

Before instruction			4 cycles after instruction		
A4	0005 0003h	5, 3	A8	0000001Bh	$(5 \times 3) + (3 \times 4) = 27$
A5	0102 0304h	1, 2, 3, 4	A9	0000000Bh	$(5 \times 1) + (3 \times 2) = 11$

**Example 2**

DDOTP4 .M1X A4,B5,A9:A8

Before instruction			4 cycles after instruction		
A4	8000 8000h		A8	FF810000h	
B5	8080 7F7Fh		A9	00800000h	

## 4.84 DDOTP4H

2-Way SIMD Dot Product, Signed by Signed Packed 16-bit

**Syntax**    **DDOTP4H** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	2	1	0
0	0	0	1	dst	src2		src1	x	0	opfield		1 1 0 0	s	p			

5                        5                        5                        5                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	01000

**Description**

The DDOTP4H instruction returns two dot-products between four sets of packed 16-bit values. The values in dwop1 and xdwop2 are treated as signed packed 16-bit quantities

For each pair of 16-bit quantities in the low 2 words of qwop1 and qwop2, the signed 16-bit value from qwop1 is multiplied with the signed 16-bit value from qwop2. The four products are summed together, and the resulting dot product is written to the low 32-bits of dwdst.

And for each pair of 16-bit quantities in the high 2 words of qwop1 and qwop2, the signed 16-bit value from qwop1 is multiplied with the signed 16-bit value from qwop2. The four products are summed together, and the resulting dot product is written to the high 32-bits of dwdst.

The result of each dot product is saturated to 32-bits, and the sat bits are set in CSR and SSR

DDOTP4H instruction is functionally equivalent to following instruction sequence:

```
    DOTP4H  src1_h, src2_h, dst_3:dst_2
||  DOTP4H  src1_l, src2_l, dst_1:dst_0
```

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0	
a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0									←qwop1
dotp4h																
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0									←qwop2
=																
a_7 * b_7 + a_6 * b_6 + a_5 * b_5 + a_4 * b_4								a_3 * b_3 + a_2 * b_2 + a_1 * b_1 + a_0 * b_0								←dwdst

<b>Execution</b>	TBD
<b>Instruction Type</b>	4-cycle
<b>Delay Slots</b>	3
<b>Functional Unit Latency</b>	1
<b>See Also</b>	
<b>Example</b>	<pre>A3 == 0x321089AB A2 == 0x321089AB A1 == 0x321089AB A0 == 0x321089AB A7 == 0x87654321 A6 == 0x87654321 A5 == 0x87654321 A4 == 0x87654321 DDOTP4H .M A3:A2:A1:A0,A7:A6:A5:A4,A15:A14 A15 &lt;== 0x92c560b6 A14 &lt;== 0x92c560b6</pre>

## 4.85 DDOTPH2

Double Dot Product, Two Pairs, Signed, Packed 16-Bit

**Syntax**   **DDOTPH2** (.unit) *src1\_o:src1\_e, src2, dst\_o:dst\_e*  
 unit = .M1 or .M2

**Compatibility**

Opcode												src2			
31	30	29	28	27	dst					src2				18	
					5					5					
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0
		<i>src1</i>	x	0	1	0	1	1	1	1	1	0	0	s	p
		5		1									1		1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	ds2	.M1, .M2
<i>src2</i>	xs2	
<i>dst</i>	dint	

**Description**

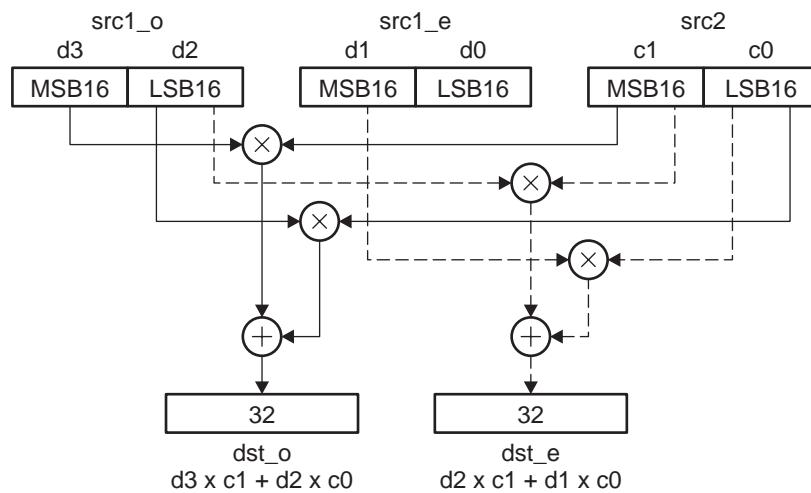
Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1\_e*, *src1\_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are written to a 64-bit register pair.

The product of the lower halfwords of *src1\_o* and *src2* is added to the product of the upper halfwords of *src1\_o* and *src2*. The result is then written to *dst\_o*.

The product of the upper halfword of *src2* and the lower halfword of *src1\_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1\_e*. The result is then written to *dst\_e*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst\_o:dst\_e*.

This instruction executes unconditionally.



**Execution**  $\text{sat}((\text{msb16}(\text{src1\_o}) \times \text{msb16}(\text{src2})) + (\text{lsb16}(\text{src1\_o}) \times \text{lsb16}(\text{src2}))) \rightarrow \text{dst\_o}$

$\text{sat}((\text{lsb16}(\text{src1\_o}) \times \text{msb16}(\text{src2})) + (\text{msb16}(\text{src1\_e}) \times \text{lsb16}(\text{src2}))) \rightarrow \text{dst\_e}$

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DDOTPL2](#), [DDOTPH2R](#), [DDOTPL2R](#)

**Examples** [Example 1](#)

DDOTPH2 .M1 A5:A4,A6,A9:A8

Before instruction			4 cycles after instruction <sup>1</sup>		
A4	0005 0003h	5,3	A8	00000021h	$(4 \times 7) + (5 \times 1) = 33$
A5	0002 0004h	2,4	A9	00000012h	$(2 \times 7) + (4 \times 1) = 18$
A6	0007 0001h	7,1			

1. CSR.SAT and SSR.M1 unchanged by operation

**Example 2**

DDOTPH2 .M1 A5:A4,A6,A9:A8

Before instruction			4 cycles after instruction		
A4	8000 5678h		A8	7FFFFFFFh	
A5	1234 8000h		A9	36E60000h	
A6	8000 8000h				

CSR	<span style="border: 1px solid black; padding: 2px;">00010100h</span>	CSR <sup>1</sup>	<span style="border: 1px solid black; padding: 2px;">00010300h</span>
SSR	<span style="border: 1px solid black; padding: 2px;">00000000h</span>	SSR <sup>1</sup>	<span style="border: 1px solid black; padding: 2px;">00000010h</span>

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

### Example 3

DDOTPH2 .M2X B5:B4,A6,B9:B8

<b>Before instruction</b>		<b>4 cycles after instruction<sup>1</sup></b>	
B4	<span style="border: 1px solid black; padding: 2px;">46B4 16BAh</span>	B8	<span style="border: 1px solid black; padding: 2px;">F41B4AFFh</span>
B5	<span style="border: 1px solid black; padding: 2px;">BBAE D169h</span>	B9	<span style="border: 1px solid black; padding: 2px;">F3B4FAADh</span>
A6	<span style="border: 1px solid black; padding: 2px;">340B F73Bh</span>		

1. CSR.SAT and SSR.M2 unchanged by operation

## 4.86 DDOTPH2R

Double Dot Product With Rounding, Two Pairs, Signed, Packed 16-Bit

**Syntax** **DDOTPH2R** (.unit) *src1\_o:src1\_e, src2, dst*

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	30	29	28	27			23	22											18
0	0	0	1		<i>dst</i>				<i>src2</i>										
					5					5									
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
	<i>src1</i>		x	0	1	0	1	0	1	1	1	0	0	s	p				
			5	1										1	1				

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>ds2</i>	.M1, .M2
<i>src2</i>	<i>xs2</i>	
<i>dst</i>	<i>s2</i>	

**Description**

Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1\_e*, *src1\_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded, shifted right by 16 and packed into a 32-bit register.

The product of the lower halfwords of *src1\_o* and *src2* is added to the product of the upper halfwords of *src1\_o* and *src2*. The result is rounded by adding  $2^{15}$  to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 most-significant bits of *dst*.

The product of the upper halfword of *src2* and the lower halfword of *src1\_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1\_e*. The result is rounded by adding  $2^{15}$  to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 least-significant bits of *dst*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

This instruction executes unconditionally.

**Execution**

```
msb16(sat((msb16(src1_o) × msb16(src2)) +
(lsb16(src1_o) × lsb16(src2)) + 00008000h)) → msb16(dst)

msb16(sat((lsb16(src1_o) × msb16(src2)) +
(msb16(src1_e) × lsb16(src2)) + 00008000h)) → lsb16(dst)
```

**Instruction Type**

Four-cycle

**Delay Slots**

3

**See Also**

[DDOTPH2](#), [DDOTPL2](#), [DDOTPL2R](#)

**Examples**

[Example 1](#)

DDOTPH2R .M1 A5:A4,A6,A8

	<b>Before instruction</b>	<b>4 cycles after instruction<sup>1</sup></b>
A4	46B4 16BAh	A8
A5	BBAE D169h	F3B5 F41Bh
A6	340B F73Bh	

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 2

DDOTPH2R .M1 A5:A4,A6,A8

	<b>Before instruction</b>	<b>4 cycles after instruction</b>
A4	8000 5678h	A8
A5	1234 8000h	36E6 7FFFh
A6	8000 8001h	
CSR	00010100h	CSR <sup>1</sup>
SSR	00000000h	SSR <sup>1</sup>

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

### Example 3

DDOTPH2R .M2 B5:B4,B6,B8

	<b>Before instruction</b>	<b>4 cycles after instruction</b>
B4	8000 8000h	B8
B5	8000 8000h	7FFF 7FFFh
B6	8000 8001h	
CSR	00010100h	CSR <sup>1</sup>
SSR	00000000h	SSR <sup>1</sup>

1. CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

## 4.87 DDOTPL2

Double Dot Product, Two Pairs, Signed, Packed 16-Bit

**Syntax** **DDOTPL2 (.unit) src1\_o:src1\_e, src2, dst\_h\_o:dst\_e**

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	30	29	28	27	dst					src2					18
0	0	0	1												5
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0
	src1	x	0	1	0	1	1	1	0	1	1	0	0	s	p

5                    1

1                    1

Opcode map field used...	For operand type...	Unit
src1	ds2	.M1, .M2
src2	xs2	
dst	dint	

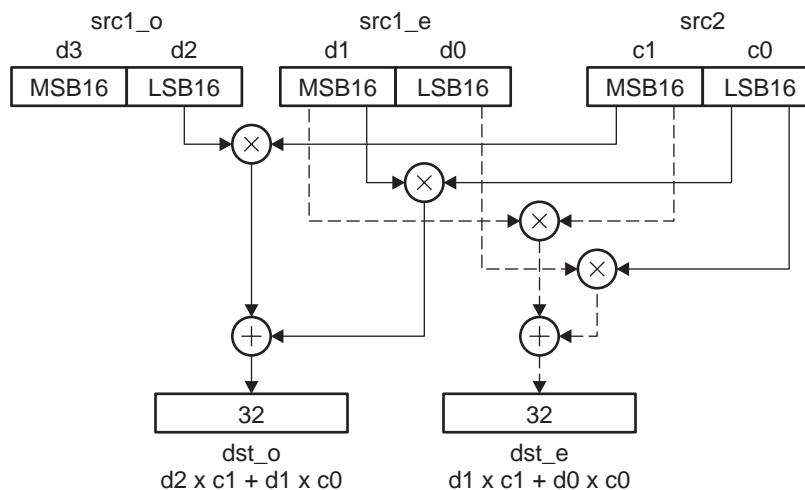
**Description**

Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1\_e*, *src1\_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are written to a 64-bit register pair.

The product of the lower halfwords of *src1\_e* and *src2* is added to the product of the upper halfwords of *src1\_e* and *src2*. The result is then written to *dst\_e*.

The product of the upper halfword of *src2* and the lower halfword of *src1\_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1\_e*. The result is then written to *dst\_o*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst\_o:dst\_e*.



<b>Execution</b>	$\text{sat}((\text{msb16(src1\_e)} \times \text{msb16(src2)}) + (\text{lsb16(src1\_e)} \times \text{lsb16(src2)})) \rightarrow \text{dst\_esat}((\text{lsb16(src1\_o)} \times \text{msb16(src2)}) + (\text{msb16(src1\_e)} \times \text{lsb16(src2)})) \rightarrow \text{dst\_o}$
<b>Instruction Type</b>	Four-cycle
<b>Delay Slots</b>	3
<b>See Also</b>	<a href="#">DDOTPH2</a> , <a href="#">DDOTPL2R</a> , <a href="#">DDOTPH2R</a>
<b>Examples</b>	<b>Example 1</b>

DDOTPL2 .M1 A5:A4,A6,A9:A8

<b>Before instruction</b>			<b>4 cycles after instruction<sup>1</sup></b>		
A4	<span style="border: 1px solid black; padding: 2px;">0005 0003h</span>	5 , 3	A8	<span style="border: 1px solid black; padding: 2px;">00000026h</span>	$(4 \times 7) + (5 \times 1) = 33$
A5	<span style="border: 1px solid black; padding: 2px;">0002 0004h</span>	2 , 4	A9	<span style="border: 1px solid black; padding: 2px;">00000021h</span>	$(2 \times 7) + (4 \times 1) = 18$
A6	<span style="border: 1px solid black; padding: 2px;">0007 0001h</span>	7 , 1			

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 2

DDOTPL2 .M1 A5:A4,A6,A9:A8

<b>Before instruction</b>			<b>4 cycles after instruction<sup>1</sup></b>		
A4	<span style="border: 1px solid black; padding: 2px;">46B4 16BAh</span>		A8	<span style="border: 1px solid black; padding: 2px;">0D984C9Ah</span>	
A5	<span style="border: 1px solid black; padding: 2px;">BBAE D169h</span>		A9	<span style="border: 1px solid black; padding: 2px;">F41B4AFFh</span>	
A6	<span style="border: 1px solid black; padding: 2px;">340B F73Bh</span>				

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 3

DDOTPL2 .M1 A5:A4,A6,A9:A8

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
A4	<span style="border: 1px solid black; padding: 2px;">8000 5678h</span>		A8	<span style="border: 1px solid black; padding: 2px;">14C40000h</span>	
A5	<span style="border: 1px solid black; padding: 2px;">1234 8000h</span>		A9	<span style="border: 1px solid black; padding: 2px;">7FFFFFFFh</span>	
A6	<span style="border: 1px solid black; padding: 2px;">8000 8000h</span>				

CSR	<span style="border: 1px solid black; padding: 2px;">00010100h</span>	CSR <sup>1</sup>	<span style="border: 1px solid black; padding: 2px;">00010300h</span>
SSR	<span style="border: 1px solid black; padding: 2px;">00000000h</span>	SSR <sup>1</sup>	<span style="border: 1px solid black; padding: 2px;">00000010h</span>

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

## 4.88 DDOTPL2R

Double Dot Product With Rounding, Two Pairs, Signed Packed 16-Bit

**Syntax** **DDOTPL2R (.unit) src1\_o:src1\_e, src2, dst**

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	30	29	28	27			23	22											18
0	0	0	1		<i>dst</i>				<i>src2</i>										
					5					5									
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
	<i>src1</i>		x	0	1	0	1	0	0	1	1	0	0	s	p				
			5	1										1	1				

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>ds2</i>	.M1, .M2
<i>src2</i>	<i>xs2</i>	
<i>dst</i>	<i>s2</i>	

**Description** Returns two dot-products between two pairs of signed, packed 16-bit values. The values in *src1\_e*, *src1\_o*, and *src2* are treated as signed, packed 16-bit quantities. The signed results are rounded, shifted right by 16 and packed into a 32-bit register.

The product of the lower halfwords of *src1\_e* and *src2* is added to the product of the upper halfwords of *src1\_e* and *src2*. The result is rounded by adding  $2^{15}$  to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 least-significant bits of *dst*.

The product of the upper halfword of *src2* and the lower halfword of *src1\_o* is added to the product of the lower halfword of *src2* and the upper halfword of *src1\_e*. The result is rounded by adding  $2^{15}$  to it and saturated if appropriate. The 16 most-significant bits of the result are written to the 16 most-significant bits of *dst*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

**Execution**

```
msb16(sat((msb16(src1_e) × msb16(src2)) +
(lsb16(src1_e) × lsb16(src2)) + 00008000h)) → lsb16(dst)

msb16(sat((lsb16(src1_o) × msb16(src2)) +
(msb16(src1_e) × lsb16(src2)) + 00008000h)) → msb16(dst)
```

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DDOTPH2R](#), [DDOTPL2](#), [DDOTPH2](#)

**Examples** [Example 1](#)

DDOTPL2R .M1 A5:A4,A6,A8

		<b>Before instruction</b>	<b>4 cycles after instruction<sup>1</sup></b>
A4		46B4 16BAh	A8
A5		BBAE D169h	
A6		340B F73Bh	

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 2

DDOTPL2R .M1 A5:A4,A6,A8

		<b>Before instruction</b>	<b>4 cycles after instruction</b>
A4		8000 5678h	A8
A5		1234 8000h	
A6		8000 8001h	
CSR		00010100h	CSR <sup>1</sup>
SSR		00000000h	SSR <sup>1</sup>

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

### Example 3

DDOTPL2R .M2 B5:B4,B6,B8

		<b>Before instruction</b>	<b>4 cycles after instruction</b>
B4		8000 8000h	B8
B5		8000 8000h	
B6		8000 8001h	
CSR		00010100h	CSR <sup>1</sup>
SSR		00000000h	SSR <sup>1</sup>

1. CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

## 4.89 DDOTPSU4H

2-Way SIMD Dot Product, Signed By Unsigned Packed 16-bit

**Syntax** **DDOTPSU4H** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	0		opfield	1	1	0	0	s	p

5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	01001

**Description** The DDOTPSU4H instruction returns two dot-products between four sets of packed 16-bit values.

The values in qwop1 are treated as signed packed 16-bit quantities, while the values in qwop2 are treated as unsigned 16-bit quantities.

For each pair of 16-bit quantities in the low 2 words of qwop1 and qwop2, the signed 16-bit value from qwop1 is multiplied with the unsigned 16-bit value from qwop2. The four products are summed together, and the resulting dot product is written to the low 32-bits of dwdst.

And for each pair of 16-bit quantities in the high 2 words of qwop1 and qwop2, the signed 16-bit value from qwop1 is multiplied with the unsigned 16-bit value from qwop2. The four products are summed together, and the resulting dot product is written to the high 32-bits of dwdst.

The result of each dot product is saturated to 32-bits, and the sat bits are set in CSR and SSR

The DDOTPSU4H instruction is functionally equivalent to following instruction sequence:

```
    DOTPSU4H src1_h, src2_h, dst_3:dst_2
||  DOTPSU4H src1_l, src2_l, dst_1:dst_0
```

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0						
a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0									←qwop1					
dotpsu4h								dotpsu4h													
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0									←qwop2					
=								=													
$a_7 * b_7 + a_6 * b_6 + a_5 * b_5 + a_4 * b_4$								$a_3 * b_3 + a_2 * b_2 + a_1 * b_1 + a_0 * b_0$											←dwdst		

<b>Execution</b>	TBD
<b>Instruction Type</b>	4-cycle
<b>Delay Slots</b>	3
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">DOTPSU4H</a> , <a href="#">DOTP2</a> , <a href="#">DMPYU4</a>

**Example**

```

CSR == 0x00000000
A3 == 0x7f7f7f7f
A2 == 0x7f7f7f7f
A1 == 0x7f7f7f7f
A0 == 0x7f7f7f7f
A7 == 0xffffffff
A6 == 0xffffffff
A5 == 0xffffffff
A4 == 0xffffffff
DDOTPSU4H .M A3:A2:A1:A0,A7:A6:A5:A4,A15:A14
A15 <== 0x7fffffff
A14 <== 0x7fffffff
CSR<== 0x10010200
CSR == 0x00000000

```

**4.90 DEAL**

## Deinterleave and Pack

**Syntax**      **DEAL** (.unit) *src2*, *dst*

unit = .M1 or .M2

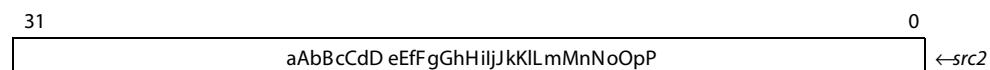
## *Opcode*

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>src2</i>	xuint	.M1, .M2
<i>dst</i>	uint	

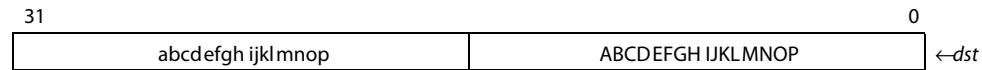
#### *Description*

Performs a deinterleave and pack operation on the bits in *src2*. The odd and even bits of *src2* are extracted into two separate, 16-bit quantities. These 16-bit quantities are then packed such that the even bits are placed in the lower halfword, and the odd bits are placed in the upper halfword.

As a result, bits 0, 2, 4, ..., 28, 30 of  $src2$  are placed in bits 0, 1, 2, ..., 14, 15 of  $dst$ . Likewise, bits 1, 3, 5, ..., 29, 31 of  $src2$  are placed in bits 16, 17, 18, ..., 30, 31 of  $dst$ .



DEAL



**Note**—The **DEAL** instruction is the exact inverse of the **SHFL** instruction (see **SHFL3**).

**Execution**      if (cond) {  
 $src2_{31,29,27\dots 1} \rightarrow dst_{31,30,29\dots 16}$   
 $src2_{30,28,26\dots 0} \rightarrow dst_{15,14,13\dots 0}$

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>
Read	<i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [SHFL](#)

**Example** DEAL .M1 A1,A2

**Before instruction**

A1	<table border="1"><tr><td>9E52 6E30h</td></tr></table>	9E52 6E30h
9E52 6E30h		
A2	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh
xxxx xxxxh		

**2 cycles after instruction**

A1	<table border="1"><tr><td>9E52 6E30h</td></tr></table>	9E52 6E30h
9E52 6E30h		
A2	<table border="1"><tr><td>B174 6CA4h</td></tr></table>	B174 6CA4h
B174 6CA4h		

## 4.91 DINT

Disable Interrupts and Save Previous Enable State

**Syntax**    **DINT**

unit = none

**Compatibility**

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	s	p

1      1

**Description**

Disables interrupts in the current cycle, copies the contents of the GIE bit in TSR into the SGIE bit in TSR, and clears the GIE bit in both TSR and CSR. The PGIE bit in CSR is unchanged.

The CPU will not service a maskable interrupt in the cycle immediately following the **DINT** instruction. This behavior differs from writes to GIE using the **MVC** instruction. See section 5.2 for details.

The **DINT** instruction cannot be placed in parallel with the following instructions: **MVC reg, TSR; MVC reg, CSR; B IRP; B NRP; NOP n; RINT; SPKERNEL; SPKERNELR; SPLOOP; SPLOOPD; SPLOOPW; SPMASK; or SPMASKR.**

This instruction executes unconditionally.

 **Note**—The use of the **DINT** and **RINT** instructions in a nested manner, like the following code:

DINT

DINT

RINT

RINT

leaves interrupts disabled. The first **DINT** leaves TSR.GIE cleared to 0, so the second **DINT** leaves TSR.SGIE cleared to 0. The **RINT** instructions, therefore, copy zero to TSR.GIE (leaving interrupts disabled).

**Execution**

Disable interrupts in current cycle

GIE bit in TSR → SGIE bit in TSR

0 → GIE bit in TSR

0 → GIE bit in CSR

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**

[RINT](#)

## 4.92 DINTHSP

2-Way SIMD Convert 16-bit Signed Integer to Single Precision Floating Point

### Syntax

Mnemonic	Unit	Operand
DINTHSP	L,S1 or L,S2	xop,dwdst

**Opcode**      Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	2	1	0
creg	z		dst		scst5		opcode	x		0011010110		s	p
3			5		5		5			10			

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.L1 or .L2	10010

**Opcode**      Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	2	1	0
creg	z		dst		src		opcode	x		1111001000		s	p
3			5		5		5			10			

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.S1 or .S2	10010

**Description**      The signed, packed 16-bit, values in src2 are converted to single-precision floating point values and placed in dst\_e and dst\_o.

**Execution**

```
if(cond) {
    sp(slsb16(src2)) -> dst_e
    sp(smsb16(src2)) -> dst_o
}
else nop
```

**Instruction Type**    3-cycle

**Delay Slots**        2

**Functional Unit Latency**    1

**See Also**            DPINT, DPTRUNC, INTDP, INTDPU, INTSP, INTSPU, SPTRUNC

**Example**

```
FADCR== 0x00000000
A2 == 0x19651127
DINTHSP .L A2,A1:A0
A1 == 0x45CB2800
A0 == 0x45893800
```

```
FADCR= 0x00000000
;RMODE(0) ;[6501]->[6501] ;[4391]->[4391] ;RMODE(0) ;
FADCR== 0x00000000
A2 == 0xffffffffde
DINTHSP .L A2,A1:A0
A1 == 0xbff80000
A0 == 0xc2080000

FADCR== 0x00000000
;RMODE(0) ;[-1]->[-1] ;[-34]->[-34] ;RMODE(0)
```

## 4.93 DINTHSP

2-Way SIMD Convert 32-bit Signed Integer to Single Precision Floating Point, Packed Signed 32-bit

**Syntax**    **DINTHSP** (.unit) *src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		scst5		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.L1 or .L2	10111

**Opcode**    Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src		opfield	x	1	1	1	1	0	0	1	0	0	0	0	s	p

3                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.S1 or .S2	10101

**Description**    The signed, packed 32-bit, values in src2 are converted to single-precision floating point values and placed in dst\_e and dst\_o.

**Execution**

```
if(cond) {
    sp(src2_e) -> dst_e
    sp(src2_o) -> dst_o
}
else nop
```

**Instruction Type**    3-cycle

**Delay Slots**    2

**Functional Unit Latency**    1

**See Also**    [DPINT](#), [DPTRUNC](#), [INTDP](#), [INTDPU](#), [INTSP](#), [INTSPU](#), [SPTRUNC](#)

**Example**

## 4.94 DINTHSPU

2-Way SIMD Convert 16-bit Unsigned Integer to Single Precision Floating Point

**Syntax** **DINTHSPU** (.unit) *src2*, *dst*

unit = .L1, .L2, .S1, or .S2

**Opcode** Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	2	1	0
creg	z		dst		src2		opfield	x		0011010110	s	p	

3                    5                    5                    5                    10

Opcode map field used...	For operand type...	Unit	Opfield
src2, dst	xop,dwdst	.L1 or .L2	10011

**Opcode** Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	2	1	0
creg	z		dst		src2		opfield	x		1111001000	s	p	

3                    5                    5                    5                    10

Opcode map field used...	For operand type...	Unit	Opfield
src2, dst	xop,dwdst	.S1 or .S2	10011

**Description** The unsigned, packed 16-bit, values in src2 are converted to single-precision floating point values and placed in dst\_h and dst\_l.

**Execution**

```
if(cond) {
    sp(ulsb16(src2)) -> dst_e
    sp(umsb16(src2)) -> dst_o
}
```

else nop

**Instruction Type** 3-cycle

**Delay Slots** 2

**Functional Unit Latency** 1

**See Also** [DPINT](#), [DPTRUNC](#), [INTDP](#), [INTDPU](#), [INTSP](#), [INTSPU](#), [SPTRUNC](#)

**Example**

```
FADCR== 0x00000000
A2 == 0x19651127
DINTHSPU .L A2,A1:A0
A1 == 0x45CB2800
A0 == 0x45893800
```

```
FADCR= 0x00000000
;RMODE(0);[6501]->[4391];[4391]->[4391];RMODE(0);
FADCR== 0x00000000
A2 == 0xffffffffde
DINTHSPU .L A2,A0
A0 == 0x477FFF00
== 0x477FDE00

FADCR= 0x00000000
;RMODE(0);[2^16-1]->[2^16-1];[2^16-34]->[2^16-34];RMODE(0);
```

## 4.95 DINTSPU

2-Way SIMD Convert 32-bit Unsigned Integer to Single Precision Floating Point,  
Packed Unsigned 32-bit

**Syntax**    **DINTSPU** (.unit) *src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		scst5		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2, dst	xop, dwdst	.L1 or .L2	10110

**Opcode**    Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src		opfield	x	1	1	1	1	0	0	1	0	0	0	s	p	

3                        5                        5

Opcode map field used...	For operand type...	Unit	Opfield
src2, dst	xop, dwdst	.S1 or .S2	10001

**Description**    The unsigned, packed 32-bit, values in src2 are converted to single-precision floating point values and placed in dst\_e and dst\_o.

**Execution**

```
if(cond) {
    sp(src2_e) -> dst_e
    sp(src2_o) -> dst_o
}
else nop
```

**Instruction Type**    3-cycle

**Delay Slots**    2

**Functional Unit Latency**    1

**See Also**    [DPINT](#), [DPTRUNC](#), [INTDP](#), [INTDPU](#), [INTSP](#), [INTSPU](#), [SPTRUNC](#)

**Example**

## 4.96 DMAX2

2-Way SIMD Maximum, Packed Signed 16-bit

**Syntax** **DMAX2 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23 22	18 17	13	12	11	5 4	2	1	0
0	0	0	1		dst	src2	src1	x		opfield	1 1 0	s	p

5                    5                    5                    7                    3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1000010

**Description** The DMAX2 performs four maximum operations on packed signed 16-bit values. For each pair of signed 16-bit values in *dwop1* and *xdwop2*, DMAX2 places the larger value in the corresponding position in *dwdst*.

**Execution**

```
if(lsb16(src1) >= lsb16(src2), lsb16(src1) -> lsb16(dst)
else lsb16(src2) -> lsb16(dst)
if(msb16(src1) >= msb16(src2), msb16(src1) -> msb16(dst)
else msb16(src2) -> msb16(dst)
if(lsb16(src1) >= lsb16(src2), lsb16(src1) -> lsb16(dst)
else lsb16(src2) -> lsb16(dst)
if(msb16(src1) >= msb16(src2), msb16(src1) -> msb16(dst)
else msb16(src2) -> msb16(dst)
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [MIN2](#), [MAXU4](#), [MINU4](#)

**Example**

```
A1 == 0x80007fff
A0 == 0x80007fff
A3 == 0x00008000
A2 == 0x00008000
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x00007fff
A4 == 0x00007fff

A1 == 0x11118001
A0 == 0x11118001
A3 == 0x22228003
A2 == 0x22228003
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x22228003
A4 == 0x22228003
```

```

A1 == 0xfffffffff
A0 == 0xfffffffff
A3 == 0x7fffffff
A2 == 0x7fffffff
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x7fffffff
A4 == 0x7fffffff

A1 == 0x7fffffff
A0 == 0x7fffffff
A3 == 0x8000ffff
A2 == 0x8000ffff
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x7fffffff
A4 == 0x7fffffff

A1 == 0x7fffffff
A0 == 0x7fffffff
A3 == 0x7fffffff
A2 == 0x7fffffff
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x7fffffff
A4 == 0x7fffffff

A1 == 0xffffeffff
A0 == 0xffffeffff
A3 == 0x7fff7ffe
A2 == 0x7fff7ffe
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x7fff7ffe
A4 == 0x7fff7ffe

A1 == 0x43211234
A0 == 0x43211234
A3 == 0x23411324
A2 == 0x23411324
DMAX2 .L A1:A0,A3:A2,A5:A4
A5 == 0x43211324
A4 == 0x43211324

```

## 4.97 DMAXU4

4-Way SIMD Maximum, Packed Unsigned 8-bit

**Syntax** **DMAXU4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0
0	0	0	1	dst		src2		src1	x		opfield		110	s	p	

5                    5                    5                    7                    3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1000011

**Description** The DMAXU4 performs eight maximum operations on packed unsigned 8-bit values. For each pair of unsigned 8-bit values in *dwop1* and *xdwop2*, MAXU4 places the larger value in the corresponding position in *dwdst*.

```
Execution
if(ubyte0(src1_e) >= ubyte0(src2_e), ubyte0(src1_e) -> ubyte0(dst_e)
else ubyte0(src2_e) -> ubyte0(dst_e)
if(ubyte1(src1_e) >= ubyte1(src2_e), ubyte1(src1_e) -> ubyte1(dst_e)
else ubyte1(src2_e) -> ubyte0(dst_e)
if(ubyte2(src1_e) >= ubyte2(src2_e), ubyte2(src1_e) -> ubyte2(dst_e)
else ubyte2(src2_e) -> ubyte2(dst_e)
if(ubyte3(src1_e) >= ubyte3(src2_e), ubyte3(src1_e) -> ubyte3(dst_e)
else ubyte3(src2_e) -> ubyte3(dst_e)
if(ubyte0(src1_o) >= ubyte0(src2_o), ubyte0(src1_o) -> ubyte0(dst_o)
else ubyte0(src2_o) -> ubyte0(dst_o)
if(ubyte1(src1_o) >= ubyte1(src2_o), ubyte1(src1_o) -> ubyte1(dst_o)
else ubyte1(src2_o) -> ubyte0(dst_o)
if(ubyte2(src1_o) >= ubyte2(src2_o), ubyte2(src1_o) -> ubyte2(dst_o)
else ubyte2(src2_o) -> ubyte2(dst_o)
if(ubyte3(src1_o) >= ubyte3(src2_o), ubyte3(src1_o) -> ubyte3(dst_o)
else ubyte3(src2_o) -> ubyte3(dst_o)
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [MINU4](#), [MAX2](#), [MIN2](#)

**Example**

```

A3 == 0xbabebef
A2 == 0xbabebef
A1 == 0xbeefbabe
A0 == 0xbeefbabe
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0xbeefbeef
A4 == 0xbeefbeef

A3 == 0x11223344
A2 == 0x11223344
A1 == 0x44332211
A0 == 0x44332211
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0x44333344
A4 == 0x44333344

A3 == 0x807f0040
A2 == 0x807f0040
A1 == 0x7f801180
A0 == 0x7f801180
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0x80801180
A4 == 0x80801180

A3 == 0xfffffeffe
A2 == 0xfffffeffe
A1 == 0xfeffffeff
A0 == 0xfeffffeff
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0xffffffff
A4 == 0xffffffff

A3 == 0x00000000
A2 == 0x00000000
A1 == 0xfffffff
A0 == 0xfffffff
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0xfffffff
A4 == 0xfffffff

A3 == 0x00ff00ff
A2 == 0x00ff00ff
A1 == 0xff00ff00
A0 == 0xff00ff00
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0xfffffff
A4 == 0xfffffff

A3 == 0xabcdefad
A2 == 0xabcdefad
A1 == 0xbadcfcda
A0 == 0xbadcfcda
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0xbadcfcda
A4 == 0xbadcfcda

A3 == 0x43211234
A2 == 0x43211234
A1 == 0x23411324
A0 == 0x23411324
DMAXU4 .L A3:A2,A1:A0,A5:A4
A5 == 0x43411334
A4 == 0x43411334

```

## 4.98 DMIN2

2-Way SIMD Minimum, Packed Signed 16-bit

**Syntax**    **DMIN2 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0
0	0	0	1		dst		src2		src1	x		opfield		110	s	p

5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1000001

**Description**    The DMIN2 performs four minimum operations on packed signed 16-bit values. For each pair of signed 16-bit values in *dwop1* and *xdwop2*, DMIN2 places the smaller value in the corresponding position in *dwdst*.

```
Execution    if(lsb16(src1_e) <= lsb16(src2_e), lsb16(src1_e) -> lsb16(dst_e)
              else lsb16(src2_e) -> lsb16(dst_e)
              if(msb16(src1_e) <= msb16(src2_e), msb16(src1_e) -> msb16(dst_e)
              else msb16(src2_e) -> msb16(dst_e)
              if(lsb16(src1_o) <= lsb16(src2_o), lsb16(src1_o) -> lsb16(dst_o)
              else lsb16(src2_o) -> lsb16(dst_o)
              if(msb16(src1_o) <= msb16(src2_o), msb16(src1_o) -> msb16(dst_o)
              else msb16(src2_o) -> msb16(dst_o)
```

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [MAX2](#), [MAXU4](#), [MINU4](#)

**Example**

```
A3 == 0x80007fff
A2 == 0x80007fff
A1 == 0x00008000
A0 == 0x00008000
DMIN2 .L A3:A2,A1:A0,A5:A4
A5 == 0x80008000
A4 == 0x80008000

A3 == 0x11118001
A2 == 0x11118001
A1 == 0x22228003
A0 == 0x22228003
DMIN2 .L A3:A2,A1:A0,A5:A2
A5 == 0x11118001
A4 == 0x11118001
```

```

A3 == 0xfffffffff
A2 == 0xfffffffff
A1 == 0x7fffffff
A0 == 0x7fffffff
DMIN2 .L A3:A2,A1:A0,A5:A4
A5 == 0xfffffffff
A4 == 0xfffffffff

A3 == 0x7ffffffe
A2 == 0x7ffffffe
A1 == 0x8000ffff
A0 == 0x8000ffff
DMIN2 .L A3:A2,A1:A0,A5:A4
A5 == 0x8000ffffe
A4 == 0x8000ffffe

A3 == 0x7fffffff
A2 == 0x7fffffff
A1 == 0x7ffffffe
A0 == 0x7ffffffe
DMIN2 .L A3:A2,A1:A0,A5:A4
A5 == 0x7ffffffe
A4 == 0x7ffffffe

A3 == 0xffffeffff
A2 == 0xffffeffff
A1 == 0x7fff7ffe
A0 == 0x7fff7ffe
DMIN2 .L A3:A2,A1:A0,A5:A4
A5 == 0xffffeffff
A4 == 0xffffeffff

A3 == 0x43211234
A2 == 0x43211234
A1 == 0x23411324
A0 == 0x23411324
DMIN2 .L A3:A2,A1:A0,A5:A4
A5 == 0x23411234
A4 == 0x23411234

```

## 4.99 DMINU4

4-Way SIMD Minimum, Packed Unsigned 8-bit

**Syntax** DMINU4 (.unit) *src1, src2, dst*

unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0
0	0	0	1	dst		src2		src1	x		opfield		110	s	p	

5                    5                    5                    7                    3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1001000

**Description** The DMINU4 performs eight minimum operations on packed unsigned 8-bit values. For each pair of unsigned 8-bit values in *dwop1* and *xdwop2*, DMINU4 places the smaller value in the corresponding position in *dwdst*.

```
Execution
if(ubyte0(src1_e) <= ubyte0(src2_e), ubyte0(src1_e) -> ubyte0(dst_e)
else ubyte0(src2_e) -> ubyte0(dst_e)
if(ubyte1(src1_e) <= ubyte1(src2_e), ubyte1(src1_e) -> ubyte1(dst_e)
else ubyte1(src2_e) -> ubyte0(dst_e)
if(ubyte2(src1_e) <= ubyte2(src2_e), ubyte2(src1_e) -> ubyte2(dst_e)
else ubyte2(src2_e) -> ubyte2(dst_e)
if(ubyte3(src1_e) <= ubyte3(src2_e), ubyte3(src1_e) -> ubyte3(dst_e)
else ubyte3(src2_e) -> ubyte3(dst_e)
if(ubyte0(src1_o) <= ubyte0(src2_o), ubyte0(src1_o) -> ubyte0(dst_o)
else ubyte0(src2_o) -> ubyte0(dst_o)
if(ubyte1(src1_o) <= ubyte1(src2_o), ubyte1(src1_o) -> ubyte1(dst_o)
else ubyte1(src2_o) -> ubyte0(dst_o)
if(ubyte2(src1_o) <= ubyte2(src2_o), ubyte2(src1_o) -> ubyte2(dst_o)
else ubyte2(src2_o) -> ubyte2(dst_o)
if(ubyte3(src1_o) <= ubyte3(src2_o), ubyte3(src1_o) -> ubyte3(dst_o)
else ubyte3(src2_o) -> ubyte3(dst_o)
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** MAXU4, MAX2, MIN2

**Example**

```

A1 == 0xffffeffe
A0 == 0xffffeffe
A3 == 0xefffffff
A2 == 0xefffffff
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0xfefefefe
A8 == 0xfefefefe

A1 == 0x00000000
A0 == 0x00000000
A3 == 0xfffffff
A2 == 0xffffffff
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0x00000000
A8 == 0x00000000

A1 == 0x00ff00ff
A0 == 0x00ff00ff
A3 == 0xff00ff00
A2 == 0xff00ff00
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0x00000000
A8 == 0x00000000

A1 == 0xabcdefad
A0 == 0xabcdefad
A3 == 0xbadcfeda
A2 == 0xbadcfeda
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0xabcdefad
A8 == 0xabcdefad

A1 == 0x43211234
A0 == 0x43211234
A3 == 0x23411324
A2 == 0x23411324
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0x23211224
A8 == 0x23211224

A1 == 0x77665544
A0 == 0x77665544
A3 == 0x11223344
A2 == 0x11223344
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0x11223344
A8 == 0x11223344

A1 == 0x33445566
A0 == 0x33445566
A3 == 0x77665544
A2 == 0x77665544
DMINU4 .L A1:A0,A3:A2,A9:A8
A9 == 0x33445544
A8 == 0x33445544

```

## 4.100 DMPY2

4-Way SIMD Multiply, Packed Signed 16-bit

**Syntax**    **DMPY2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	2	1	0
0	0	0	1	dst	src2	src1	x	0	opfield	1100	s	p	5	4			

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,qwdst	.M1 or .M2	00000

**Description**    The DMPY2 instruction performs four 16-bit multiplications between signed packed 16-bit quantities. The values in *dwop1* and *xdwop2* are treated as signed packed 16-bit quantities. The 32-bit results are placed in a 128-bit register quad.

**Execution**

```
lsbl6(src1_e) x lsbl6(src2_e) -> dst_0
msbl6(src1_e) x msbl6(src2_e) -> dst_1
lsbl6(src1_o) x lsbl6(src2_o) -> dst_2
msbl6(src1_o) x msbl6(src2_o) -> dst_3
```

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [MPYSU4](#)

**Example**

```
A5 == 0x12343497
A4 == 0x6a321193
A9 == 0x21ff50a7
A8 == 0xb1746ca4
DMPY2 .M A5:A4,A9:A8,A3:A2:A1:A0
A3 == 0x026ad5cc
A2 == 0x10917e81
A1 == 0xdf6ab0a8
A0 == 0x0775462c

A5 == 0xffff7fff
A4 == 0x80018001
A9 == 0x7fff7fff
A8 == 0x80018001
DMPY2 .M A5:A4,A9:A8,A3:A2:A1:A0
A3 == 0x3fff0001
A2 == 0x3fff0001
A1 == 0x3fff0001
A0 == 0x3fff0001

A5 == 0x7fff8001
A4 == 0x3ccdc333
A9 == 0x80017fff
A8 == 0xc333c333
DMPY2 .M A5:A4,A9:A8,A3:A2:A1:A0
```

```
A3 == 0xc000ffff
A2 == 0xc000ffff
A1 == 0xf18f43d7
A0 == 0x0e70bc29

A5 == 0x87654321
A4 == 0x80008000
A9 == 0x321089ab
A8 == 0x80008000
DMPY2 .M A5:A4,A9:A8,A3:A2:A1:A0
A3 == 0xe86a3050
A2 == 0xe0f8800b
A1 == 0x40000000
A0 == 0x40000000
```

## 4.101 DMPYSP

2-Way SIMD Multiply, Packed Single Precision Floating Point

**Syntax** **DMPYSP (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	2	1	0
0	0	0	1	dst		src2		src1	x		opfield		00000		s	p

5                    5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	11100

**Description**

Special Cases:

1. If one source is SNaN or QNaN, the result is a signed NaN\_out and the NANn bit is set. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the XOR to the input signs.
2. Signed infinity multiplied by signed infinity or a normalized number (other than signed zero) returns signed infinity. Signed infinity multiplied by signed zero (or denormal) returns a signed NaN\_out and sets the INVAL bit.
3. If one or both source are signed zero, the result is signed zero unless the other source is a NaN or signed infinity, in which case the result is signed NaN\_out.
4. If signed zero is multiplied by signed infinity, the result is signed NaN\_out and the INVAL bit is set.
5. A denormalized source is treated as signed zero and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, OR signed zero. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
6. If rounding is performed, the INEX bit is set.

**Execution**

src1\_e x src2\_e -> dst\_e

src1\_o x src2\_o -> dst\_o

**Instruction Type**

4-cycle

**Delay Slots**

3

**Functional Unit Latency**

1

**See Also**

[MPYDP](#)

**Example**

```

FMCR== 0x00000000
A5 == 0x3f800000
A4 == 0xbff800000
A9 == 0xbff800000
A8 == 0x7fc00000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0xbff800000
A0 == 0xffffffff

```

```

FMCR= 0x00000002

FMCR== 0x00000000
A5 == 0xbff800000
A4 == 0x3f800000
A9 == 0x7fc00000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0xffffffff
A0 == 0x3f800000

FMCR= 0x00000002

FMCR== 0x00000000
A5 == 0xbff800000
A4 == 0x3f800000
A9 == 0x7fc00000
A8 == 0x7fc00000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0xffffffff
A0 == 0x7fffffff

FMCR= 0x00000002

FMCR== 0x00000000
A5 == 0x3f800000
A4 == 0xffc00000
A9 == 0x3f800000
A8 == 0xbff800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x3f800000
A0 == 0x7fffffff

FMCR= 0x00000001

FMCR== 0x00000000
A5 == 0xffc00000
A4 == 0x3f800000
A9 == 0xbff800000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x3f800000

FMCR= 0x00000001

FMCR== 0x00000000
A5 == 0xffc00000
A4 == 0x3f800000
A9 == 0xbff800000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x7fffffff

FMCR= 0x00000001

FMCR== 0x00000000
A5 == 0xbff800000
A4 == 0x3f800000
A9 == 0xbff800000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x3f800000
A0 == 0x3f800000

FMCR= 0x00000000

FMCR== 0x00000000
A5 == 0x7f900000
A4 == 0x3f800000
A9 == 0x7f900000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x3f800000

FMCR= 0x00000013

```

```

FMCR== 0x00000000
A5 == 0x3f800000
A4 == 0x7f900000
A9 == 0x7f900000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x7fffffff

FMCR= 0x00000013

FMCR== 0x00000000
A5 == 0x7f900000
A4 == 0x3f800000
A9 == 0x3f800000
A8 == 0x7f900000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x7fffffff

FMCR= 0x00000013

FMCR== 0x00000000
A5 == 0x7f900000
A4 == 0x7f900000
A9 == 0x7f900000
A8 == 0xff900000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0xffffffff

FMCR= 0x00000013

FMCR== 0x00000000
A5 == 0x3356bf94
A4 == 0x3f800000
A9 == 0x43ff8000
A8 == 0x3f800000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x37d65434
A0 == 0x3f800000

FMCR= 0x00000080

FMCR== 0x00000000
A5 == 0x3356bf94
A4 == 0x43ff8000
A9 == 0x43ff8000
A8 == 0x3356bf94
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x37d65434
A0 == 0x37d65434

FMCR= 0x00000080

FMCR== 0x00000000
A5 == 0x3f800000
A4 == 0x3356bf94
A9 == 0x3f800000
A8 == 0x43ff8000
DMPYSP .M A5:A4,A9:A8,A1:A0
A1 == 0x3f800000
A0 == 0x37d65434

FMCR= 0x00000080

FMCR== 0x00000000
B5 == 0x3356bf94
B4 == 0x3f800000
B9 == 0x43ff8000
B8 == 0x3f800000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0x37d65434
B0 == 0x3f800000

FMCR= 0x00800000

FMCR== 0x00000000
B5 == 0x3356bf94
B4 == 0x43ff8000
B9 == 0x43ff8000

```

```

B8 == 0x3356bf94
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0x37d65434
B0 == 0x37d65434

FMCR= 0x00800000

FMCR== 0x00000000
B5 == 0x3f800000
B4 == 0x3356bf94
B9 == 0x3f800000
B8 == 0x43ff8000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0x3f800000
B0 == 0x37d65434

FMCR= 0x00800000
FMCR== 0x00000000
B5 == 0x7fc00000
B4 == 0x7f900000
B9 == 0xbff80000
B8 == 0x3f800000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0xffffffff
B0 == 0x7fffffff

FMCR= 0x00110000

FMCR== 0x00000000
B5 == 0xbff800000
B4 == 0x3f800000
B9 == 0x7fc00000
B8 == 0x7f900000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0xffffffff
B0 == 0x7fffffff

FMCR= 0x00120000

FMCR== 0x00000000
B5 == 0x7fc00000
B4 == 0x7f900000
B9 == 0xff900000
B8 == 0x3f800000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0xffffffff
B0 == 0x7fffffff

FMCR= 0x00130000

FMCR== 0x00000000
B5 == 0xff900000
B4 == 0x3f800000
B9 == 0x7fc00000
B8 == 0x7f900000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0xffffffff
B0 == 0x7fffffff

FMCR= 0x00130000

FMCR== 0x00000000
B5 == 0x7fc00000
B4 == 0x3f800000
B9 == 0xbff800000
B8 == 0x7f900000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0xffffffff
B0 == 0x7fffffff

FMCR= 0x00130000

FMCR== 0x00000000
B5 == 0x3f800000
B4 == 0x7fc00000
B9 == 0x7f900000
B8 == 0xbff800000
DMPYSP .M B5:B4,B9:B8,B1:B0
B1 == 0x7fffffff
B0 == 0xffffffff

```

FMCR= 0x00130000

4.102 DMPYSU4

## 4-Way SIMD Multiply Signed By Unsigned, Packed 8-bit

**Syntax**    **DMPYSU4** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**      Opcode for .M Unit, 32-bit

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,qwdst	.M1 or .M2	11000

**Description** For each 8-bit quantity in dwop1 and xdwop2, DMPYSU4 performs a signed 8-bit by unsigned 8-bit multiply between the value from dwop1 and xdwop2, producing a signed 16-bit result. The eight signed 16-bit results are packed into a 128-bit register quad.

This instruction is exactly equivalent to executing two MPYSU4 instructions in parallel.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	$\leftarrow op1$
MYPSU4				MYPSU4				
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	$\leftarrow xop2$
=				=				
a_7 * b_7	a_6 * b_6	a_5 * b_5	a_4 * b_4	a_3 * b_3	a_2 * b_2	a_1 * b_1	a_0 * b_0	

<b>Execution</b>	sbyte0 (src1_e) x byte0 (src2_e) -> lsb16(dst_0)
	sbyte1 (src1_e) x byte1 (src2_e) -> msb16(dst_0)
	sbyte2 (src1_e) x byte2 (src2_e) -> lsb16(dst_1)
	sbyte3 (src1_e) x byte3 (src2_e) -> msb16(dst_1)
	sbyte0 (src1_o) x byte0 (src2_o) -> lsb16(dst_2)
	sbyte1 (src1_o) x byte1 (src2_o) -> msb16(dst_2)
	sbyte2 (src1_o) x byte2 (src2_o) -> lsb16(dst_3)
	sbyte3 (src1_o) x byte3 (src2_o) -> msb16(dst_3)

**Instruction Type** 4-cycle

*Delay Slots* 3

## ***Functional Unit Latency***

**See Also** MPYU4, DOTPU4

### *Example*

## 4.103 DMPYU2

4-Way SIMD Multiply Unsigned by Unsigned, Packed 16-bit

**Syntax** **DMPYU2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	2	1	0
0	0	0	1	dst		src2		src1	x		opfield		00000		s	p

5                    5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,qwdst	.M1 or .M2	00001

**Description** The DMPYU2 instruction performs four 16-bit multiplications between unsigned packed 16-bit quantities. The values in *dwop1* and *xdwop2* are treated as unsigned packed 16-bit quantities. The 32-bit results are placed in a 128-bit register quad.

**Execution**

```

ulsb16(src1_e) x ulsb16(src2_e) -> dst_0
umsb16(src1_e) x umsb16(src2_e) -> dst_1
ulsb16(src1_o) x ulsb16(src2_o) -> dst_2
umsb16(src1_o) x umsb16(src2_o) -> dst_3

```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** MPYSU4

**Example**

```

A5 == 0x12343497
A4 == 0x6a321193
A9 == 0x21ff50a7
A8 == 0xb1746ca4
DMPYU2 .M A5:A4,A9:A8,A3:A2:A1:A0
A3 == 0x026ad5cc
A2 == 0x10917e81
A1 == 0x499cb0a8
A0 == 0x0775462c

A5 == 0xffff7fff
A4 == 0x80018001
A9 == 0xffff7fff
A8 == 0x80018001
DMPYU2 .M A5:A4,A9:A8,A3:A2:A1:A0
A3 == 0x3fff0001
A2 == 0x3fff0001
A1 == 0x40010001
A0 == 0x40010001

A5 == 0x7fff8001
A4 == 0x3ccdc333
A9 == 0x80017fff
A8 == 0xc333c333
DMPYU2 .M A5:A4,A9:A8,A3:A2:A1:A0

```

```
A3 == 0x3fffffff
A2 == 0x3fffffff
A1 == 0x2e5c43d7
A0 == 0x94d6bc29

A5 == 0x87654321
A4 == 0x80008000
A9 == 0x321089ab
A8 == 0x80008000
DMPYU2 .M A5:A4,A9:A8,A3:A2:A1:A0
A3 == 0x1a7a3050
A2 == 0x2419800b
A1 == 0x40000000
A0 == 0x40000000
```

4.104 DMPYU4

## 4-Way SIMD Multiply Unsigned By Unsigned, Packed 8-bit

**Syntax**      **DMPYU4** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode**      Opcode for .M Unit, 32-bit

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1, src2, dst	dwop1, xdwop2, qwdst	.M1 or .M2	00100

**Description** For each 8-bit quantity in dwop1 and xdwop2, DMPYU4 performs eight unsigned 8-bit by unsigned 8-bit multiplies between the values from dwop1 and xdwop2, producing eight signed 16-bit results packed into a 128-bit register quad.

This instruction is exactly equivalent to executing two MPYU4 instructions in parallel.

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	$\leftarrow op1$
MYPUI4				MYPUI4				
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	$\leftarrow xop2$
=				=				
a_7 * b_7	a_6 * b_6	a_5 * b_5	a_4 * b_4	a_3 * b_3	a_2 * b_2	a_1 * b_1	a_0 * b_0	

<b>Execution</b>	byte0 (src1_e)	x byte0 (src2_e)	-> lsb16(dst_0)
	byte1 (src1_e)	x byte1 (src2_e)	-> msb16(dst_0)
	byte2 (src1_e)	x byte2 (src2_e)	-> lsb16(dst_1)
	byte3 (src1_e)	x byte3 (src2_e)	-> msb16(dst_1)
	byte0 (src1_o)	x byte0 (src2_o)	-> lsb16(dst_2)
	byte1 (src1_o)	x byte1 (src2_o)	-> msb16(dst_2)
	byte2 (src1_o)	x byte2 (src2_o)	-> lsb16(dst_3)
	byte3 (src1_o)	x byte3 (src2_o)	-> msb16(dst_3)

***Instruction Type***      4-cycle

*Delay Slots* 3

## ***Functional Unit Latency***

**See Also** MPYSU4

### *Example*

## 4.105 DMV

Move Two Independent Registers to a Register Pair

**Syntax**    **DMV (.unit) src1, src2, dst**

unit = .S1, .S2, .L1, or .L2

Mnemonic	Unit	Operand
DMV	L,S1 or L,S2	op1,xop2,dwdst

**Opcode**    Opcode for .S Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	2	1	0
creg	z		dst		src2		src1	x		11		opfield		1100		s	p
3			5		5		5			2		4		4			

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dwdst	.S1 or .S2	1011

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0	
creg	z		dst		src2		src1	x		opfield		110		s	p	
3			5		5		5			7		3				

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dwdst	.L1 or .L2	1101100

**Description**    This instruction moves two registers into a register pair. This performs 2 moves at once and is required when performing large amounts of double word processing

**Execution**

```
if(cond){
  0 + src2_e -> dst_e
  0 + src2_o -> dst_o
} else nop
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also**

**Example**

```
A0 == 0x87654321
A1 == 0x12345678
DMV .L A0,A1,A3:A2
A3 == 0x87654321
A2 == 0x12345678
```

## 4.106 DMVD

Move Two Independent Registers to a Register Pair, Delayed

**Syntax**    **DMVD (.unit) src1, src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode**    Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	2	1	0
0	0	0	1		dst		src2		src1	x	11		opfield		1100	s	p	

5                        5                        5                        2                        4                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dwdst	.S1 or .S2	0011

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0	
0	0	0	1		dst		src2		src1	x		opfield		110	s	p	

5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dwdst	.L1 or .L2	1101100

**Description**    This instruction moves two registers into a register pair. This performs 2 moves at once and is useful when performing large amounts of double word processing. The writeback is delayed by 4 clocks to reduce register pressure.

**Execution**

```
if(cond){
    0 + src2_e -> dst_e
    0 + src2_o -> dst_o
}
else nop
```

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**

**Example**

```
A1 == 0x87654321
A6 == 0x12345678
DMVD .L A1,A6,A3:A2
A3 == 0x87654321
A2 == 0x12345678
```

## 4.107 DOTP2

Dot Product, Signed, Packed 16-Bit

**Syntax**    **DOTP2 (.unit) src1, src2, dst**

or

**DOTP2 (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22		18
creg	z	dst		src2		
3	1	5		5		
17	13	12	11	10	6	5
src1	x	0	op		1	1
5	1		5		0	s
					1	p

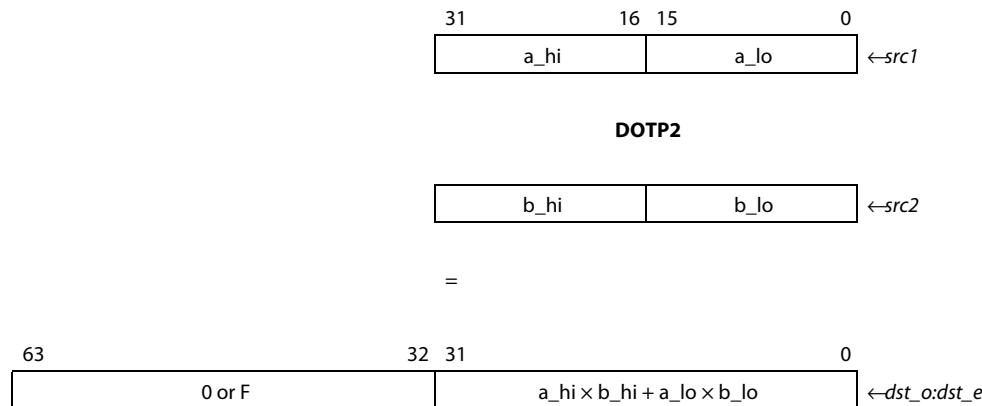
Opcode map field used...	For operand type...	Unit	Opfield
src1	s2	.M1, .M2	01100
src2	xs2		
dst	int		
src1	s2	.M1, .M2	01011
src2	xs2		
dst	sllong		

**Description**

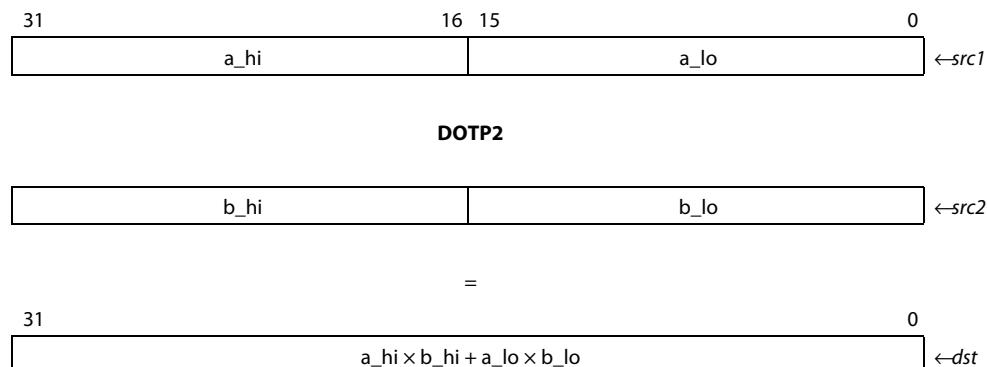
Returns the dot-product between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed result is written either to a single 32-bit register, or sign-extended into a 64-bit register pair.

The product of the lower halfwords of *src1* and *src2* is added to the product of the upper halfwords of *src1* and *src2*. The result is then written to the *dst*.

If the result is sign-extended into a 64-bit register pair, the upper word of the register pair always contains either all 0s or all 1s, depending on whether the result is positive or negative, respectively.



The 32-bit result version returns the same results that the 64-bit result version does in the lower 32 bits. The upper 32-bits are discarded.



**Note**—In the overflow case, where all four halfwords in *src1* and *src2* are 8000h, the value 8000 0000h is written into the 32-bit *dst* and 0000 0000 8000 0000h is written into the 64-bit *dst*.

## ***Execution***

```
if (cond) (lsb16(src1) × lsb16(src2)) + (msb16(src1) × msb16(src2)) → dst
```

```
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src1, src2</i>		
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTPN2](#)

**Examples** [Example 1](#)

```
DOTP2 .M1 A5,A6,A8
```

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
A5	6A32 1193h	27186 4499	A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812	A6	B174 6CA4h	
A8	xxxx xxxxh		A8	E6DF F6D4h	-421,529,900

**Example 2**

```
DOTP2 .M1 A5,A6,A9:A8
```

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
A5	6A32 1193h	27186 4499	A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812	A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	FFFF FFFFh	E6DF F6D4h -421,529,900

**Example 3**

```
DOTP2 .M2 B2,B5,B8
```

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B2	1234 3497h	4660 13463	B2	1234 3497h	
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	

B8	xxxx xxxxh	B8	12FC 544Dh	318,526,541
----	------------	----	------------	-------------

### Example 4

DOTP2 .M2 B2,B5,B9:B8

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B2	1234 3497h	4660 13463	B2	1234 3497h	
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	0000 0000h	12FC 544Dh

318,526,541

## 4.108 DOTP4H

Dot Product, Signed by Signed, Packed 16-bit

**Syntax** **DOTP4H (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, 2 src fixed hdr

31	29	28	27	23	22	18	17	13	12	11	10	6	5	4	5	2	1	0
creg	z		dst		src2		src1	x	0		opfield	1	1	0	0	s	p	

3                    5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dst	.M1 or .M2	00110
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	00111

**Description** The DOTP4H instruction returns the dot-product between a vector of four 16-bit two sets of packed 16-bit values.

The values in dwop1 and xdwop2 are treated as signed packed 16-bit quantities

For each pair of 16-bit quantities in op1 and xop2, the signed 16-bit value from op1 is multiplied with the signed 16-bit value from xop2. The four products are summed together, and the resulting dot product is written to either a 32-bit result or to a 64-bit signed result.

For the 32-bit destination form, the result is saturated to 32-bits, and the sat bits are set in CSR and SSR.

$$\begin{array}{cccccccccc}
 & a_3 & a_2 & a_1 & a_0 & a_3 & a_2 & a_1 & a_0 & \leftarrow op1 \\
 & \text{DDOTP4H} \\
 & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & \leftarrow xop2 \\
 & = \\
 & a_3 * b_3 + a_2 * b_2 + a_1 * b_1 + a_0 * b_0 & \leftarrow dst
 \end{array}$$

<b>Execution</b>	TBD
<b>Instruction Type</b>	4-cycle
<b>Delay Slots</b>	3
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">DOTPU4</a> , <a href="#">DOTP2</a> , <a href="#">MPYU4</a>
<b>Example</b>	<pre>A3 == 0x321089AB A2 == 0x321089AB A1 == 0x87654321 A0 == 0x87654321 DOTP4H .M A3:A2,A1:A0,A14:A13 A14 &lt;== 0xFFFFFFFF A13 &lt;== 0x92C560B6  CSR == 0x10010000 A3 == 0x7FFF7FFF A2 == 0x7FFF7FFF A1 == 0x7FFF7FFF A0 == 0x7FFF7FFF DOTP4H .M A3:A2,A1:A0 ; Maximum Positive inputs . A14 &lt;== 0xFFFFFFFF  CSR&lt;== 0x10010200</pre>

## 4.109 DOTPN2

Dot Product With Negate, Signed, Packed 16-Bit

**Syntax** **DOTPN2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22											18
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>										
3		1			5											5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
<i>src1</i>	<i>x</i>	0	0	1	0	0	1	1	1	1	0	0	<i>s</i>	<i>p</i>		1	1	
5		1														1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.M1, .M2
<i>src2</i>	<i>xs2</i>	
<i>dst</i>	<i>int</i>	

**Description** Returns the dot-product between two pairs of signed, packed 16-bit values where the second product is negated. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The signed result is written to a single 32-bit register.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The result is then written to *dst*.

31		16 15		0
	<i>a_hi</i>		<i>a_lo</i>	$\leftarrow \text{src1}$

**DOTPN2**

<i>b_hi</i>		<i>b_lo</i>	$\leftarrow \text{src2}$
-------------	--	-------------	--------------------------

=

31		0
	<i>a_hi</i> $\times$ <i>b_hi</i> - <i>a_lo</i> $\times$ <i>b_lo</i>	$\leftarrow \text{dst}$

**Execution**

Note that unlike **DOTP2**, no overflow case exists for this instruction.

```
if (cond) (msb16(src1) × msb16(src2)) - (lsb16(src1) × lsb16(src2)) → dst
```

---

else nop

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTP2](#)
**Examples** [Example 1](#)

DOTPN2 .M1 A5,A6,A8

**Before instruction**

A5	3629 274Ah	13865 10058
A6	325C 8036h	12892 -32714
A8	xxxx xxxxh	

**4 cycles after instruction**

A5	3629 274Ah
A6	325C 8036h
A8	1E44 2F20h 507,784,992

**Example 2**

DOTPN2 .M2 B2,B5,B8

**Before instruction**

B2	3FF6 5010h	16374 20496
B5	B1C3 0244h	-20029 580
B8	xxxx xxxxh	

**4 cycles after instruction**

B2	3FF6 5010h
B5	B1C3 0244h
B8	EBBE 6A22h -339,842,526

## 4.110 DOTPNRSU2

Dot Product With Negate, Shift and Round, Signed by Unsigned, Packed 16-Bit

**Syntax** **DOTPNSR2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22			18				
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>						
3		1		5				5						
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	0	0	1	1	1	1	1	0	0	<i>s</i>	<i>p</i>	
5		1									1	1		

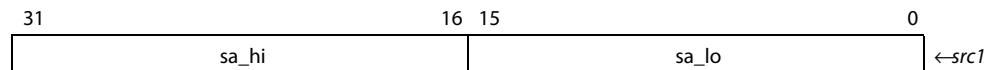
Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.M1, .M2
<i>src2</i>	<i>xu2</i>	
<i>dst</i>	<i>int</i>	

**Description**

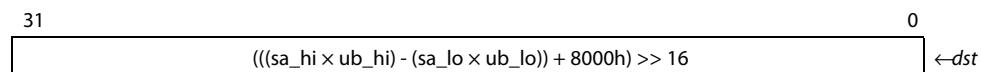
Returns the dot-product between two pairs of packed 16-bit values, where the second product is negated. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed, packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned, packed 16-bit quantities. The results are written to *dst*.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The value  $2^{15}$  is then added to this sum, producing an intermediate 33-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

The intermediate results of the **DOTPNSR2** instruction are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.

**DOTPNSU2**

=



**Execution**

```

if (cond) {
    int33 = (smsb16(src1)  $\times$  umsb16(src2)) -
            (slsb16(src1)  $\times$  ulsb16(src2)) + 8000h;
    int33 >> 16  $\rightarrow dst$ 
}
else nop

```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read		src1, src2		
Written				dst
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTP2](#), [DOTPN2](#), [DOTPRS2](#)

**Examples** [Example 1](#)

DOTPNRSU2 .M1 A5, A6, A8

Before instruction			4 cycles after instruction		
A5	<span style="border: 1px solid black; padding: 2px;">3629 274Ah</span>	13865 10058 signed	A5	<span style="border: 1px solid black; padding: 2px;">3629 274Ah</span>	
A6	<span style="border: 1px solid black; padding: 2px;">325C 8036h</span>	12892 32822 unsigned	A6	<span style="border: 1px solid black; padding: 2px;">325C 8036h</span>	
A8	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>		A8	<span style="border: 1px solid black; padding: 2px;">FFFF F6FAh</span>	-2310 (signed)

**Example 2**

DOTPNRSU2 .M2 B2, B5, B8

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B2	3FF6 5010h	16374 20496 signed	B2	3FF6 5010h	
B5	B1C3 0244h	45507 580 unsigned	B5	B1C3 0244h	
B8	xxxx xxxxh		B8	0000 2BB4h	11188 (signed)

### Example 3

DOTPNRSU2 .M2 B12, B23, B11

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B12	7FFF 8000h	32767 -32768 signed	B12	7FFF 8000h	
B23	FFFF FFFFh	65535 65535 unsigned	B23	FFFF FFFFh	
B11	xxxx xxxxh		B11	xxxx xxxxh	Overflow occurs; result undefined

## 4.111 DOTPNRUS2

Dot Product With Negate, Shift and Round, Unsigned by Signed, Packed 16-Bit

**Syntax** **DOTPNSRUS2 (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22						18				
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>				
3	1			5						5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	0	0	1	1	1	1	1	1	0	0	<i>s</i>	<i>p</i>
5	1											1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.M1, .M2
<i>src2</i>	<i>xu2</i>	
<i>dst</i>	<i>int</i>	

**Description**

The **DOTPNSRUS2** pseudo-operation performs the dot-product between two pairs of packed 16-bit values, where the second product is negated. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed, packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned, packed 16-bit quantities. The results are written to *dst*. The assembler uses the **DOTPNSRUS2 src1, src2, dst** instruction to perform this task.

The product of the lower halfwords of *src1* and *src2* is subtracted from the product of the upper halfwords of *src1* and *src2*. The value  $2^{15}$  is then added to this sum, producing an intermediate 32 or 33-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

The intermediate results of the **DOTPNSRUS2** pseudo-operation are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.

**Execution**

```
if (cond) {
    int33 = (smsb16(src1) × umsb16(src2)) -
            (slsb16(src1) × ulsb16(src2)) + 8000h;
    int33 >> 16 → dst
```

```

        }
else nop

```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTP2](#), [DOTPN2](#), [DOTPNRSU2](#), [DOTPRSU2](#)

## 4.112 DOTPRSU2

Dot Product With Shift and Round, Signed by Unsigned, Packed 16-Bit

**Syntax** **DOTPRSU2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

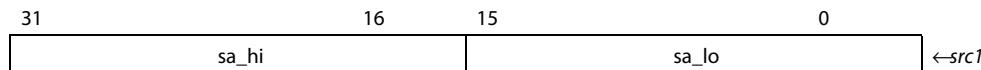
31	29	28	27			23	22			18	
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>				
3	1			5				5			
17	13	12	11	10	9	8	7	6	5	4	
<i>src1</i>	<i>x</i>	0	0	1	1	0	1	1	1	0	
5	1									1	
										1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.M1, .M2
<i>src2</i>	<i>xu2</i>	
<i>dst</i>	<i>int</i>	

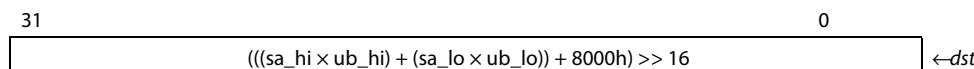
**Description** Returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product and performs an additional round and shift step. The values in *src1* are treated as signed packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned packed 16-bit quantities. The results are written to *dst*.

The product of the lower halfwords of *src1* and *src2* is added to the product of the upper halfwords of *src1* and *src2*. The value  $2^{15}$  is then added to this sum, producing an intermediate 32 or 33-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

The intermediate results of the **DOTPRSU2** instruction are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.

**DOTPRSU2**

=



**Note**—Certain combinations of operands for the **DOTPRSU2** instruction results in an overflow condition. If an overflow does occur, the result is undefined. Overflow can be avoided if the sum of the two products plus the rounding term is less than or equal to  $2^{31}-1$  for a positive sum and greater than or equal to  $-2^{31}$  for a negative sum.

The intermediate results of the **DOTPRSU2** instruction are maintained to 33-bit precision, ensuring that no overflow may occur during the adding and rounding steps.

**Execution**

```
if (cond) {
    int33 = (smsb16(src1)  $\times$  umsb16(src2)) +
            (slsb16(src1)  $\times$  ulsb16(src2)) + 8000h;
    int33  $\gg$  16  $\rightarrow$  dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		<i>.M</i>		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTP2](#), [DOTPN2](#), [DOTPNRSU2](#)

**Examples** [Example 1](#)

`DOTPRSU2 .M1 A5, A6, A8`

**Before instruction**

A5 3629 274Ah 13865 10058  
signed

**4 cycles after instruction**

A5 3629 274Ah

A6	<span style="border: 1px solid black; padding: 2px;">325C 8036h</span>	12892 32822 unsigned	A6	<span style="border: 1px solid black; padding: 2px;">325C 8036h</span>
A8	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>		A8	<span style="border: 1px solid black; padding: 2px;">0000 1E55h</span>

### Example 2

DOTPRSU2 .M2 B2, B5, B8

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B2	<span style="border: 1px solid black; padding: 2px;">B1C3 0244h</span>	-20029 580 signed	B2	<span style="border: 1px solid black; padding: 2px;">B1C3 0244h</span>	20029 580 signed
B5	<span style="border: 1px solid black; padding: 2px;">3FF6 5010h</span>	16374 20496 unsigned	B5	<span style="border: 1px solid black; padding: 2px;">3FF6 5010h</span>	16374 20496 unsigned
B8	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>		B8	<span style="border: 1px solid black; padding: 2px;">FFFF ED29h</span>	-4823 (signed)

### Example 3

DOTPRSU2 .M2 B12, B23, B11

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B12	<span style="border: 1px solid black; padding: 2px;">7FFF 7FFFh</span>	32767 32767 signed	B12	<span style="border: 1px solid black; padding: 2px;">7FFF 7FFFh</span>	
B23	<span style="border: 1px solid black; padding: 2px;">FFFF FFFFh</span>	65535 65535 unsigned	B23	<span style="border: 1px solid black; padding: 2px;">FFFF FFFFh</span>	
B11	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>		B11	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>	Overflow occurs; result undefined

## 4.113 DOTPRUS2

Dot Product With Shift and Round, Unsigned by Signed, Packed 16-Bit

**Syntax** **DOTPRUS2 (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22											18
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>										
3		1			5											5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
<i>src1</i>	<i>x</i>	0	0	1	1	0	1	1	1	1	0	0	<i>s</i>	<i>p</i>		1	1	
5		1														1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.M1, .M2
<i>src2</i>	<i>xu2</i>	
<i>dst</i>	<i>int</i>	

**Description**

The **DOTPRUS2** pseudo-operation returns the dot-product between two pairs of packed 16-bit values. This instruction takes the result of the dot-product, and performs an additional round and shift step. The values in *src1* are treated as signed packed 16-bit quantities; whereas, the values in *src2* are treated as unsigned packed 16-bit quantities. The results are written to *dst*. The assembler uses the **DOTPRSU2 (.unit) src1, src2, dst** instruction to perform this task.

The product of the lower halfwords of *src1* and *src2* is added to the product of the upper halfwords of *src1* and *src2*. The value  $2^{15}$  is then added to this sum, producing an intermediate 32-bit result. The intermediate result is signed shifted right by 16, producing a rounded, shifted result that is sign extended and placed in *dst*.

The intermediate results of the **DOTPRUS2** pseudo-operation are maintained to a 33-bit precision, ensuring that no overflow may occur during the subtracting and rounding steps.

**Execution**

```

if (cond) {
    int32 = (umsb16(src2) < smsb16(src1)) +
            (ulsb16(src2) < slsb16(src1)) + 8000h;
    int32 >> 16 → dst
}
else nop
if (cond) {
    int33 = (umsb16(src2) < smsb16(src1)) +
            (ulsb16(src2) < slsb16(src1)) + 8000h;
    int33 >> 16 → dst
}

```

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src1, src2</i>		
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTP2](#), [DOTPN2](#), [DOTPNRSU2](#), [DOTPRSU2](#)

## 4.114 DOTPSU4

Dot Product, Signed by Unsigned, Packed 8-Bit

**Syntax**    **DOTPSU4 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22											18
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>										
3		1			5											5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
<i>src1</i>	<i>x</i>	0	0	0	0	1	0	1	1	0	0	0	<i>s</i>	<i>p</i>	1	1		
5		1													1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s4</i>	.M1, .M2
<i>src2</i>	<i>xu4</i>	
<i>dst</i>	<i>int</i>	

**Description**    Returns the dot-product between four sets of packed 8-bit values. The values in *src1* are treated as signed packed 8-bit quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data. The signed result is written into *dst*.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot product is written as a signed 32-bit result to *dst*.

31	24	23	16	15	8	7	0
<i>sa_3</i>		<i>sa_2</i>		<i>sa_1</i>		<i>sa_0</i>	← <i>src1</i>

**DOTPSU4**

<i>ub_3</i>	<i>ub_2</i>	<i>ub_1</i>	<i>ub_0</i>	← <i>src2</i>
-------------	-------------	-------------	-------------	---------------

=

31	0
(sa_3 × ub_3) + (sa_2 × ub_2) + (sa_1 × ub_1) + (sa_0 × ub_0)	

←*dst*

**Execution**    if (cond) {  
                     (sbyte0(*src1*) × ubyte0(*src2*)) +  
                     (sbyte1(*src1*) × ubyte1(*src2*)) +  
                     (sbyte2(*src1*) × ubyte2(*src2*)) +  
                     (sbyte3(*src1*) × ubyte3(*src2*)) → *dst*

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src1, src2</i>		
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTPU4](#)

**Examples** [Example 1](#)

DOTPSU4 .M1 A5, A6, A8

**Before instruction**

A5 

6A 32 11 93h
--------------

 106 50 17 -109  
signed

A6 

B1 74 6C A4h
--------------

 177 116 108 164  
unsigned

A8 

xxxx xxxxh
------------

**4 cycles after instruction**

A5 

6A 32 11 93h
--------------

A6 

B1 74 6C A4h
--------------

A8 

0000 214Ah
------------

 8522 (signed)

**Example 2**

DOTPSU4 .M2 B2, B5, B8

**Before instruction**

B2 

3F F6 50 10h
--------------

 63 -10 80 16  
signed

B5 

C3 56 02 44h
--------------

 195 86 2 68  
unsigned

B8 

xxxx xxxxh
------------

**4 cycles after instruction**

B2 

3F F6 50 10h
--------------

B5 

C3 56 02 44h
--------------

B8 

0000 3181h
------------

 12,673 (signed)

## 4.115 DOTPSU4H

Dot Product, Signed by Unsigned, Packed 16-bit

**Syntax** ***DOTPSU4H*** (.unit) *src1*, *src2*, *dst*

*unit* = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
creg	Z		dst		src2		src1	x	0		opfield	1	1	0	0	s	p	

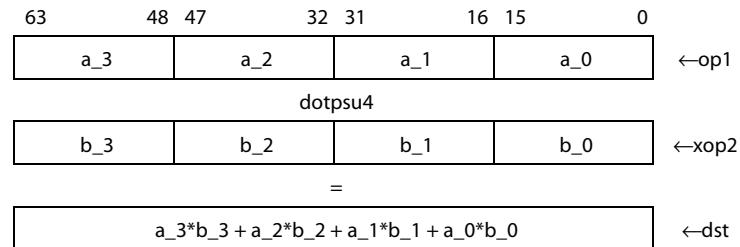
3                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dst	.M1 or .M2	00101
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	00010

**Description** The DOTPSU4H instruction returns the dot-product between four sets of packed 16-bit values. This is essentially a multiply and add operation. The values in *dwop1* are treated as signed packed 16-bit quantities, whereas the values in *xdwop2* are treated as unsigned packed 16-bit quantities.

For each pair of 16-bit quantities in *op1* and *xop2*, the signed 16-bit value from *op1* is multiplied with the unsigned 16-bit value from *xop2*. The four products are summed together, and the resulting dot product is written to either a 32-bit result or to a 64-bit signed result.

For the 32-bit destination form, the result is saturated to 32-bits, and the sat bits are set in CSR and SSR



---

<b>Execution</b>	TBD
<b>Instruction Type</b>	4-cycle
<b>Delay Slots</b>	3
<b>Functional Unit Latency</b>	1
<b>See Also</b>	
<b>Example</b>	<pre> CSR == 0x00000000 A1 == 0x7fff7fff A0 == 0x7fff7fff A3 == 0xfffffffff A2 == 0xfffffffff DOTPSU4H .M A1:A0,A3:A2,A11:A10 A11 == 0x00000001 A10 == 0xffffa0004  CSR= 0x10010000  CSR == 0x00000000 B1 == 0xfffffffff B0 == 0xfffffffff B3 == 0xfffffffff B2 == 0xfffffffff DOTPSU4H .M B1:B0,B3:B2,B11:B10 B11 == 0xfffffff B10 == 0xfffffc0004  CSR= 0x10010000  CSR == 0x00000000  CSR == 0x00000000 A1 == 0x7f7f7f7f A0 == 0x7f7f7f7f A3 == 0xfffffffff A2 == 0xfffffffff DOTPSU4H .M A1:A0,A3:A2 A11 == 0x7fffffff  CSR= 0x10010200 ; 1fdfa0204 -&gt; 7fffffff  CSR == 0x00000000 </pre>

## 4.116 DOTPUS4

Dot Product, Unsigned by Signed, Packed 8-Bit

**Syntax** **DOTPUS4 (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22											18
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>										
3		1			5											5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
<i>src1</i>	<i>x</i>	0	0	0	0	1	0	1	1	0	0	0	<i>s</i>	<i>p</i>		1	1	
5		1														1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s4</i>	.M1, .M2
<i>src2</i>	<i>xu4</i>	
<i>dst</i>	<i>int</i>	

**Description**

The **DOTPUS4** pseudo-operation returns the dot-product between four sets of packed 8-bit values. The values in *src1* are treated as signed packed 8-bit quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data. The signed result is written into *dst*. The assembler uses the **DOTPSU4 (.unit)src1, src2, dst** instruction to perform this task (see ).

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot-product is written as a signed 32-bit result to *dst*.

**Execution**

```
if (cond) {  
    (ubyte0(src2) × sbyte0(src1)) +  
    (ubyte1(src2) × sbyte1(src1)) +  
    (ubyte2(src2) × sbyte2(src1)) +  
    (ubyte3(src2) × sbyte3(src1)) → dst  
}  
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTPU4](#), [DOTPSU4](#)

## 4.117 DOTPU4

Dot Product, Unsigned, Packed 8-Bit

**Syntax**    **DOTPUI4 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22											18
creg	z			dst		src2												
3	1			5		5												
src1	x	0	0	0	1	1	0	1	1	0	0	0	s	p				
5	1												1	1				

Opcode map field used...	For operand type...	Unit
src1	u4	.M1, .M2
src2	xu4	
dst	uint	

**Description**    Returns the dot-product between four sets of packed 8-bit values. The values in both *src1* and *src2* are treated as unsigned, 8-bit packed data. The unsigned result is written into *dst*.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*. The four products are summed together, and the resulting dot-product is written as a 32-bit result to *dst*.

31	24 23	16 15	8 7	0
ua_3	ua_2	ua_1	ua_0	←src1

**DOTPUI4**

ub_3	ub_2	ub_1	ub_0	←src2
------	------	------	------	-------

=

31	0
(ua_3 × ub_3) + (ua_2 × ub_2) + (ua_1 × ub_1) + (ua_0 × ub_0)	

**Execution**    if (cond) {  
                     (ubyte0(src1) × ubyte0(src2)) +  
                     (ubyte1(src1) × ubyte1(src2)) +  
                     (ubyte2(src1) × ubyte2(src2)) +  
                     (ubyte3(src1) × ubyte3(src2)) → dst

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src1, src2</i>		
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [DOTPSU4](#)

**Example** DOTPU4 .M1 A5, A6, A8

**Before instruction**

A5	<span style="border: 1px solid black; padding: 2px;">6A 32 11 93h</span>	106 50 17 147 unsigned
----	--	---------------------------

A6	<span style="border: 1px solid black; padding: 2px;">B1 74 6C A4h</span>	177 116 108 164 unsigned
----	--	-----------------------------

A8	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>	A8 <span style="border: 1px solid black; padding: 2px;">0000 C54Ah</span> 50,506 (unsigned)
----	--	---

**4 cycles after instruction**

A5	<span style="border: 1px solid black; padding: 2px;">6A 32 11 93h</span>
----	--

A6	<span style="border: 1px solid black; padding: 2px;">B1 74 6C A4h</span>
----	--

A8	<span style="border: 1px solid black; padding: 2px;">0000 C54Ah</span>	50,506 (unsigned)
----	--	-------------------

## 4.118 DPACK2

Parallel PACK2 and PACKH2 Operations

**Syntax**    **DPACK2 (.unit) src1, src2, dst\_o:dst\_e**

unit = .L1 or .L2

**Compatibility**

**Opcode**

31	30	29	28	27				24	23	22t									18
0	0	0	1		dst			0			src2								
					4							5							
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
		src1	x	0	1	1	0	1	0	0	1	1	0	s	p				
			5	1										1	1				

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

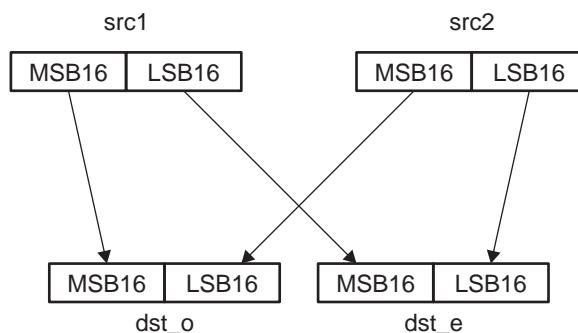
**Description**

Executes a **PACK2** instruction in parallel with a **PACKH2** instruction.

The **PACK2** function of the **DPACK2** instruction takes the lower halfword from *src1* and the lower halfword from *src2*, and packs them both into *dst\_e*. The lower halfword of *src1* is placed in the upper halfword of *dst\_e*. The lower halfword of *src2* is placed in the lower halfword of *dst\_e*.

The **PACKH2** function of the **DPACK2** instruction takes the upper halfword from *src1* and the upper halfword from *src2*, and packs them both into *dst\_o*. The upper halfword of *src1* is placed in the upper halfword of *dst\_o*. The upper halfword of *src2* is placed in the lower halfword of *dst\_o*.

This instruction executes unconditionally.



<b>Execution</b>	$lsb16(src1) \rightarrow msb16(dst\_e)$ $lsb16(src2) \rightarrow lsb16(dst\_e)$ $msb16(src1) \rightarrow msb16(dst\_o)$ $msb16(src2) \rightarrow lsb16(dst\_o)$
<b>Instruction Type</b>	Single-cycle
<b>Delay Slots</b>	0
<b>Example</b>	DPACK2 .L1 A0,A1,A3:A2



## 4.119 DPACKH2

2-Way SIMD Pack 16 MSBs Into Upper and Lower Register Halves

**Syntax** **DPACKH2** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode** Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0
0	0	0	1	dst		src2		src1	x	opfield		110		s	p	

5                    5                    5                    7                    3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0011110

**Opcode** Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	2	1	0
0	0	0	1	dst		src2		src1	x	opfield		1000		s	p	

5                    5                    5                    6                    4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	001001

**Description** The DPACKH2 instruction takes the high half-words from each of the words in *dwop1* and *xdwop2* and packs them both into *dwdst*. The upper half-word of each word in *dwop1* is placed in the upper half-word of each word in *dwdst*. The upper half-word of each word in *xdwop2* is placed in the lower half-word of each word in *dwdst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as [ADD2](#).

63	48	47	31	30	16	15	0	
QRSTUVWXYZABCDEF		xxxxxxxxxxxxxx		ABCDEFGHIJKLMNP		xxxxxxxxxxxxxx		← op1
qrstuvwxyzabcdef		xxxxxxxxxxxxxx		abcdefghijklmnp		xxxxxxxxxxxxxx		← op2
v		v		v		v		
QRSTUVWXYZABCDEF		qrstuvwxyzabcdef		ABCDEFGHIJKLMNP		abcdefghijklmnp		← dst

**Execution**

```

msb16(src2_e) -> lsb16(dst_e)
msb16(src1_e) -> msb16(dst_e)
msb16(src2_o) -> lsb16(dst_o)

```

```
msb16(src1_o)->msb16(dst_o)
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DPACKL2](#), [DPACKHL2](#), [DPACKHL2](#)

**Example**

```
A1 == 0xabcd1111
A0 == 0xabcdabcd
A3 == 0xbabe2222
A2 == 0xc0def00d
DPACKH2 .L A1:A0,A3:A2,A5:A4
A5 == 0xabcdccba
A4 == 0xabcdabab
```

## 4.120 DPACKH4

2-Way SIMD Pack Four High Bytes Into Four 8-Bit Halfwords

**Syntax**    **DPACKH4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23 22	18 17	13	12	11	5 4	2	1	0
0	0	0	1		dst	src2	src1	x		opfield	110	s	p

5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1101001

**Description**    The DPACKH4 instruction behaves precisely two PACKH4 instructions executed in parallel. E.g., the instruction

DPACKH4 A7:A6, A5:A4, A3:A2

is precisely the same as

```
PACKH4 A7,A5,A3
|| PACKH4 A6,A4,A2
```

**Execution**

```
byte3(src1_e) -> byte3(dst_e)
byte1(src1_e) -> byte2(dst_e)
byte3(src2_e) -> byte1(dst_e)
byte1(src2_e) -> byte0(dst_e)
byte3(src1_o) -> byte3(dst_o)
byte1(src1_o) -> byte2(dst_o)
byte3(src2_o) -> byte1(dst_o)
byte1(src2_o) -> byte0(dst_o)
```

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [DPACKL4](#), [DSPACKU4](#)

**Example**

```
A5 == 0xbe55ef55
A4 == 0x44556677
A3 == 0xde55ad55
A2 == 0x00112233
DPACKH4 .L A5:A4,A3:A2,A1:A0
A1 == 0xbeefdead
A0 == 0x44660022
```

## 4.121 DPACKHL2

2-Way SIMD Pack 16 MSB Into Upper and 16 LSB Into Lower Register Halves

**Syntax**    **DPACKHL2 (.unit) src1, src2, dst**

unit = .S1, .S2, .L1 or .L2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	2	1	0
0	0	0	1	dst		src2		src1	x		opfield		1000		s	p

5                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	001000

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0
0	0	0	1	dst		src2		src1	x		opcode		110		s	p

5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0011100

**Description**

The DPACKHL2 instruction takes the high half-words from the words in *dwop1* and low half-words from *xdwop2* and packs them both into the words in *dwdst*. The upper half-word of each word in *dwop1* is placed in the upper half-word of each word in *dwdst*. The lower half-word of each word of *xdwop2* is placed in the lower half-word of each word of *dwdst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as [ADD2](#).

63	48	47	31	30	16	15	0	
QRSTUVWXYZABCDEF		xxxxxxxxxxxxxxxx		ABCDEFGHIJKLMNPQ		xxxxxxxxxxxxxxxx		← op1
xxxxxxxxxxxxxxxx		qrstuvwxyzabcdef		xxxxxxxxxxxxxxxx		abcdefghijklmnpq		← op2
v		v		v		v		
QRSTUVWXYZABCDEF		qrstuvwxyzabcdef		ABCDEFGHIJKLMNPQ		abcdefghijklmnpq		← dst

<b>Execution</b>	<code>lsb16(src2_e)-&gt;lsb16(dst_e); msb16(src1_e)-&gt;msb16(dst_e); lsb16(src2_o)-&gt;lsb16(dst_o); msb16(src1_o)-&gt;msb16(dst_o);</code>
<b>Instruction Type</b>	Single-cycle
<b>Delay Slots</b>	0
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">DPACKL2</a> , <a href="#">DPACKLH2</a> , <a href="#">DPACKH2</a>
<b>Example</b>	<code>A3 == 0xceea1111 A2 == 0xddbabacf A1 == 0x2222bcae A0 == 0xc0def00d DPACKHL2 .L A3:A2,A1:A0,A15:A14 A15 == 0xabefefbe A14 == 0xdebaaf00d</code>

## 4.122 DPACKL2

2-Way SIMD Pack 16 LSBs Into Upper and Lower Register Halves

**Syntax**    **DPACKL2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	110		s	3				

5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0000000

**Opcode**    Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	2	1	0
0	0	0	1	dst	src2	src1	x	11	opfield	1100		s	4	4				

5                        5                        5                        2                        4                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	1111

**Description**    The DPACKL2 instruction takes the low half-words from each of the words in *dwop1* and *xdwop2* and packs them both into *dwdst*. The lower half-word of each word in *dwop1* is placed in the low half-word of each word in *dwdst*. The lower half-word of each word in *xdwop2* is placed in the upper half-word of each word in *dwdst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as [ADD2](#).

63	48	47	31	30	16	15	0	
xxxxxxxxxxxxxx		QRSTUUVWXYZABCDEF		xxxxxxxxxxxxxx		ABCDEFGHJKLMNOP		←op1
xxxxxxxxxxxxxx		qrstuvvxyzabcdef		xxxxxxxxxxxxxx		abcdefghijklmnp		←op2
v		v		v		v		
QRSTUUVWXYZABCDEF		qrstuvvxyzabcdef		ABCDEFGHJKLMNOP		abcdefghijklmnp		←dst

**Execution**

```

lsb16(src2_e) -> lsb16(dst_e);
lsb16(src1_e) -> msb16(dst_e);
lsb16(src2_o) -> lsb16(dst_o);
lsb16(src1_o) -> msb16(dst_o);

```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DPACKLH2](#), [DPACKHL2](#), [DPACKH2](#), [DSPACKU4](#)

**Example**

```
A1 == 0xaaaaabebe
A0 == 0xdeadbabf
A3 == 0xccccbbaf
A2 == 0xc0def00d
DPACKL2 .L A1:A0,A3:A2,A5:A4
A5 == 0xcabebab
A4 == 0xbeefc00d
```

## 4.123 DPACKL4

2-Way SIMD Pack Four Low Bytes Into Four 8-bit Halfwords

**Syntax**    **DPACKL4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23 22	18 17	13	12	11	5 4	2	1	0
0	0	0	1	dst	5	src2	5	src1	x	opfield	110	s	p

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1101000

**Description**    The DPACKL4 instruction behaves precisely two PACKL4 instruction executed in parallel. E.g., the instruction

DPACKL4 A7:A6, A5:A4, A3:A2

is precisely the same as

```
PACKL4 A7,A5,A3
|| PACKL4 A6,A4,A2
```

**Execution**

```
byte2(src1_e) -> byte3(dst_e);
byte0(src1_e) -> byte2(dst_e);
byte2(src2_e) -> byte1(dst_e);
byte0(src2_e) -> byte0(dst_e);
byte2(src1_o) -> byte3(dst_o);
byte0(src1_o) -> byte2(dst_o);
byte2(src2_o) -> byte1(dst_o);
byte0(src2_o) -> byte0(dst_o);
```

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [PACKL4](#), [DPACKH4](#), [DSPACKU4](#)

**Example**

```
A7 == 0x55de55ad
A6 == 0x89012345
A3 == 0x55be55ef
A2 == 0x01234567
DPACKL4 .L A7:A6,A3:A2,A1:A0
A1 == 0xdeafbacf
A0 == 0x01452367
```

## 4.124 DPACKLH2

2-Way SIMD Pack 16 LSB Into Upper and 16 MSB Into Lower Register Halves

**Syntax** **DPACKLH2** (.unit) *src1, src2, dst*

unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src — same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	1	0	s	p		

4                    5                    5                    5                    7                    3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0011011

**Opcode** Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opcode	1	0	0	0	s	p		

4                    5                    5                    5                    6                    4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	010000

**Description** The DPACKLH2 instruction takes the low half-words from each of the words in *dwop1* and high half-words from each of the words in *xdwop2* and packs them both into *dwdst*. The lower half-word of *dwop1* is placed in the upper half-word of each word in *dwdst*. The upper half-word of each word in *xdwop2* is placed in the lower half-word of each word in *dwdst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as [ADD2](#).

63	48	47	31	30	16	15	0	
xxxxxxxxxxxxxx		QRSTUVWXYZABCDEF		xxxxxxxxxxxxxx		ABCDEFGHIJKLMNPQ		←op1
qrstuvwxyzabcdef		xxxxxxxxxxxxxx		abcdefghijklmnop		xxxxxxxxxxxxxx		←op2
v		v		v		v		
QRSTUVWXYZABCDEF		qrstuvwxyzabcdef		ABCDEFGHIJKLMNPQ		abcdefghijklmnop		←dst

---

<b>Execution</b>	$\text{msb16}(\text{src2\_e}) \rightarrow \text{lsb16}(\text{dst\_e})$ $\text{lsb16}(\text{src1\_e}) \rightarrow \text{msb16}(\text{dst\_e})$ $\text{msb16}(\text{src2\_o}) \rightarrow \text{lsb16}(\text{dst\_o})$ $\text{lsb16}(\text{src1\_o}) \rightarrow \text{msb16}(\text{dst\_o})$
<b>Instruction Type</b>	Single-cycle
<b>Delay Slots</b>	0
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">DPACKL2</a> , <a href="#">DPACKHL2</a> , <a href="#">DPACKH2</a>
<b>Example</b>	<pre>A1 == 0xaaaaabacb A0 == 0xcbaaffabd A3 == 0xbeef1111 A2 == 0xc0def00d DPACKLH2 .L A1:A0,A3:A2,A5:A4 A5 == 0xabcfefdeb A4 == 0xbabfc0de</pre>

## 4.125 DPACKLH4

2-Way SIMD Pack High Bytes of Four Half-Words to Packed 8-bit, and Low Bytes Into Packed 8-bits

**Syntax**    **DPACKLH4** (.unit) *src1, src2, dst*

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	1	0	s	p		

4                        5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dwdst	.L1 or .L2	1101010

**Description**    The DPACKH4 instruction behaves precisely as PACKH4 and PACKL4 instructions executed in parallel on the same data set. E.g., the instruction

DPACKH4 A6, A4, A3:A2

is precisely the same as

```
PACKH4 A6,A4,A3
|| PACKL4 A6,A4,A2
```

**Execution**    byte2(src1) -> byte3(dst\_e);
byte0(src1) -> byte2(dst\_e);
byte2(src2) -> byte1(dst\_e);
byte0(src2) -> byte0(dst\_e);
byte3(src1) -> byte3(dst\_o);
byte1(src1) -> byte2(dst\_o);
byte3(src2) -> byte1(dst\_o);
byte1(src2) -> byte0(dst\_o);

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [DPACKL4](#), [DPACKH4](#)

**Example**

```
A4 == 0x44556677
A2 == 0x8899aab
DPACKLH4 .L A4,A2,A1:A0
A1 == 0x446688aa
A0 == 0x557799bb
```

## 4.126 DPACKX2

Parallel PACKLH2 Operations

**Syntax**    **DPACKX2 (.unit) src1, src2, dst\_o:dst\_e**  
 unit = .L1 or .L2

**Compatibility**

Opcode									
31	30	29	28	27	24	23	22	18	
0	0	0	1		dst	0		src2	
4									
17	13	12	11	10	9	8	7	6	5 4 3 2 1 0
src1	x	0	1	1	0	0	1	1	1 0 s p
5	1								1 1

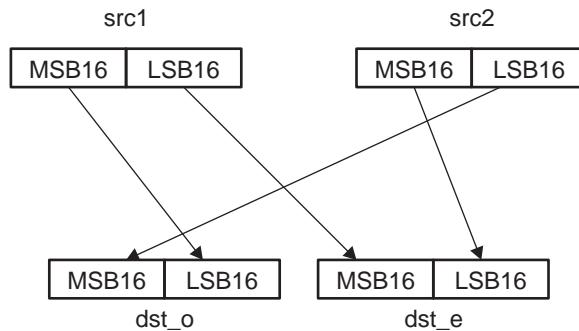
Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

**Description**    Executes two **PACKLH2** instructions in parallel.

One **PACKLH2** function of the **DPACKX2** instruction takes the lower halfword from *src1* and the upper halfword from *src2*, and packs them both into *dst\_e*. The lower halfword of *src1* is placed in the upper halfword of *dst\_e*. The upper halfword of *src2* is placed in the lower halfword of *dst\_e*.

The other **PACKLH2** function of the **DPACKX2** instruction takes the upper halfword from *src1* and the lower halfword from *src2*, and packs them both into *dst\_o*. The upper halfword of *src1* is placed in the lower halfword of *dst\_o*. The lower halfword of *src2* is placed in the upper halfword of *dst\_o*.

This instruction executes unconditionally.



<b>Execution</b>	$lsb16(src1) \rightarrow msb16(dst\_e)$ $msb16(src2) \rightarrow lsb16(dst\_e)$ $msb16(src1) \rightarrow lsb16(dst\_o)$ $lsb16(src2) \rightarrow msb16(dst\_o)$
<b>Instruction Type</b>	Single-cycle
<b>Delay Slots</b>	0
<b>Examples</b>	<b>Example 1</b>

DPACKX2 .L1 A0,A1,A3:A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	87654321h	A2	43211234h
A1	12345678h	A3	56788765h

### Example 2

DPACKX2 .L1X A0,B0,A3:A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	3FFF8000h	A2	80004000h
B0	40007777h	A3	77773FFFh

## 4.127 DPINT

Convert Double-Precision Floating-Point Value to Integer

**Syntax**    **DPINT (.unit) src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22						18	17	16	
				dst						src2		0	0
3				5						5			
15	14	13	12	11	10	9	8	7	6	5	4	3	2
0	0	0	x	0	0	0	1	0	0	0	1	1	0
				1							1		1

Opcode map field used...	For operand type...	Unit
src2	dp	.L1, .L2
dst	sint	

**Description**

The 64-bit double-precision value in *src2* is converted to an integer and placed in *dst*. The operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.



**Note—**

- 1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.
- 2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than  $2^{31}-1$  or less than  $-2^{31}$ .
- 3) If *src2* is denormalized, 0000 0000h is placed in *dst* and the INEX and DEN2 bits are set.
- 4) If rounding is performed, the INEX bit is set.

**Execution**

if (cond) int(src2) → dst

---

else nop

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src2_l, src2_h</i>		
Written				<i>dst</i>
Unit in use		.L		

**Instruction Type** Four-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [DPSP](#), [DPTRUNC](#), [INTDP](#), [SPINT](#)
**Example** DPINT .L1 A1:A0,A4

**Before instruction**

A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	3333 3333h
A4	xxxx xxxxh			A4	0000 0009h	9

**4 cycles after instruction**

## 4.128 DPSP

Convert Double-Precision Floating-Point Value to Single-Precision Floating-Point Value

**Syntax**    **DPSP** (.unit) *src2*, *dst*

unit = .L1 or .L2

**Opcode**

31	29	28	27			23	22			18	17	16
<i>creg</i>		<i>z</i>		<i>dst</i>		<i>src2</i>		0	0			
3		1		5			5					
15	14	13	12	11	10	9	8	7	6	5	4	3
0	0	0	x	0	0	0	1	0	0	1	1	1
			1							0	s	p
										1		1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	dp	.L1, .L2
<i>dst</i>	sp	

**Description**

The double-precision 64-bit value in *src2* is converted to a single-precision value and placed in *dst*. The operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.



**Note—**

- 1) If rounding is performed, the INEX bit is set.
- 2) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 3) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 4) If *src2* is a signed denormalized number, signed 0 is placed in *dst* and the INEX and DEN2 bits are set.
- 5) If *src2* is signed infinity, the result is signed infinity and the INFO bit is set.
- 6) If overflow occurs, the INEX and OVER bits are set and the results are set as follows (LFPN is the largest floating-point number):

Overflow Output Rounding Mode				
Result Sign	Nearest Even	Zero	+Infinity	Infinity
+	+infinity	+LFPN	+infinity	+LFPN
-	-infinity	-LFPN	-LFPN	-infinity

- 7) If underflow occurs, the INEX and UNDER bits are set and the results are set as follows (SPFN is the smallest floating-point number):

Underflow Output Rounding Mode				
Result Sign	Nearest Even	Zero	+Infinity	Infinity
+	+0	+0	+SFPN	+0
-	-0	-0	-0	-SFPN

**Execution**    if (cond)    sp(src2) → dst  
                   else nop

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read		src2_l, src2_h		
Written				dst
Unit in use	.L			

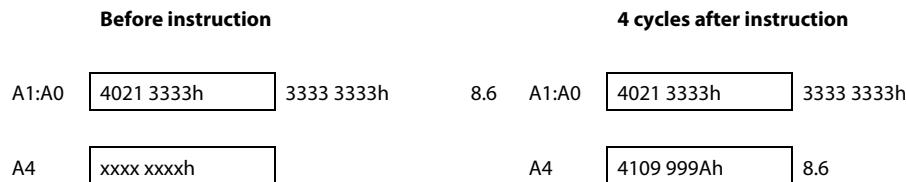
**Instruction Type** Four-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [DPINT](#), [DPTRUNC](#), [INTSP](#), [SPDP](#)

**Example** DPSP .L1 A1:A0,A4



## 4.129 DPTRUNC

Convert Double-Precision Floating-Point Value to Integer With Truncation

**Syntax**    **DPTRUNC (.unit) src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27						23	22						18	17	16
				<i>dst</i>					<i>src2</i>					0	0			
3				1			5				5							
0	0	0	x	0	0	0	0	0	0	1	1	1	0	s	p	1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	dp	.L1, .L2
<i>dst</i>	sint	

**Description**

The 64-bit double-precision value in *src2* is converted to an integer and placed in *dst*. This instruction operates like **DPINT** except that the rounding modes in the floating-point adder configuration register (FADCR) are ignored; round toward zero (truncate) is always used. The 64-bit operand is read in one cycle by using the *src2* port for the 32 MSBs and the *src1* port for the 32 LSBs.



**Note—**

- 1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.
- 2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than  $2^{31}-1$  or less than  $-2^{31}$ .
- 3) If *src2* is denormalized, 0000 0000h is placed in *dst* and the INEX and DEN2 bits are set.
- 4) If rounding is performed, the INEX bit is set.

**Execution**

if (cond) int(*src2*) → *dst*

---

else nop
**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src2_l, src2_h</i>		
Written				<i>dst</i>
Unit in use		.L		

**Instruction Type** Four-cycle**Delay Slots** 3**Functional Unit Latency** 1**See Also** [DPINT](#), [DPSP](#), [SPTRUNC](#)**Example** DPTRUNC .L1 A1:A0,A4**Before instruction**

A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	3333 3333h
A4	xxxx xxxxh			A4	0000 0008h	8

**4 cycles after instruction**

## 4.130 DSADD

2-Way SIMD Addition With Saturation, Packed Signed 32-bit

**Syntax**    **DSADD (.unit) src1, src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	0	0	0	s	p				

4                         5                         5                         6                         4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	100000

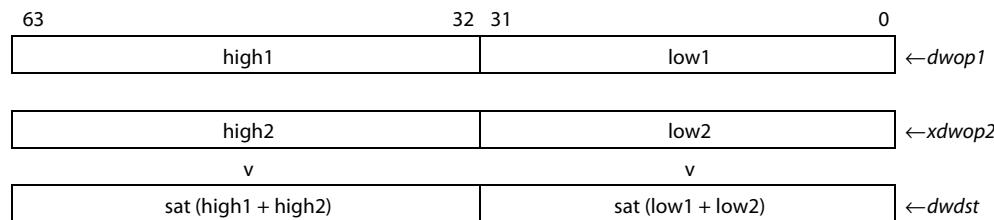
**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	1	0	s	p				

4                         5                         5                         7                         3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0010011

**Description**    The DADD instruction performs two saturating 32-bit additions of the packed 32-bit numbers contained in the two source register pairs. The addition results are returned as two 32-bit results packed into *dwdst*.



**Execution**     $\text{sat}(\text{src1}_e + \text{src2}_e) \rightarrow \text{dst}_e$   
 $\text{sat}(\text{src1}_o + \text{src2}_o) \rightarrow \text{dst}_o$

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DADD](#), [SADD](#), [DADD2](#)

**Example**

```
A1 == 0x44444444
A0 == 0x7fffffff
A3 == 0xcccccc444
A2 == 0x00000001
DSADD .L A1:A0,A3:A2,A15:A14
A15 == 0x11110888
A14 == 0x7fffffff
```

## 4.131 DSADD2

4-Way SIMD Addition with Saturation, Packed Signed 16-bit

**Syntax**    **DSADD2 (.unit) src1, src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode**    Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	1	1	opfield	1	1	0	0	s	p		

4                                 5                                5                                2                                4                                4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	0000

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

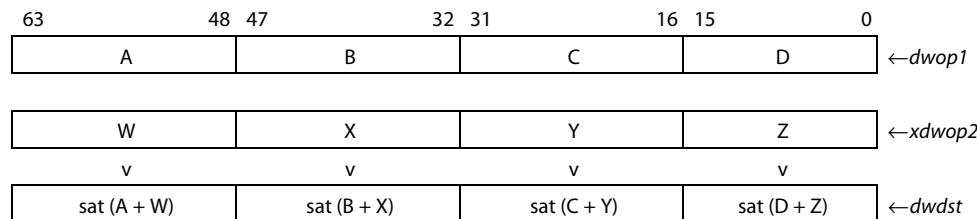
31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	0	0	s	p		

4                                 5                                5                                7                                3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0010100

**Description**

The DSADD2 instruction performs four saturating 16-bit additions of the packed signed 16-bit numbers contained in the two 64-bit source registers. The addition results are returned as four signed 16-bit results packed into *dst*. Results are saturated to within the range  $-2^{15}$  to  $2^{15}-1$ .



**Execution**

```
sat(msb16(src1_e) + msb(src2_e)) -> msb16(dst_e)
sat(lsb16(src1_e) + lsb(src2_e)) -> lsb16(dst_e)
sat(msb16(src1_o) + msb(src2_o)) -> msb16(dst_o)
sat(lsb16(src1_o) + lsb(src2_o)) -> lsb16(dst_o)
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DADD](#), [DSADD](#), [DSADD2](#), [ADD2](#)

**Example**

```
A1 == 0x44444444
A0 == 0x7fff0002
A3 == 0xcccccc444
A2 == 0x7ff00005
DSADD2 .L A1:A0,A3:A2,A15:A14
A15 == 0x11100888
A14 == 0x7fff0007
```

## 4.132 DSHL

2-Way SIMD Shift Left, Packed Signed 32-bit

**Syntax**    **DSHL (.unit) src1, src2, dst**

unit = .S1 or .S2

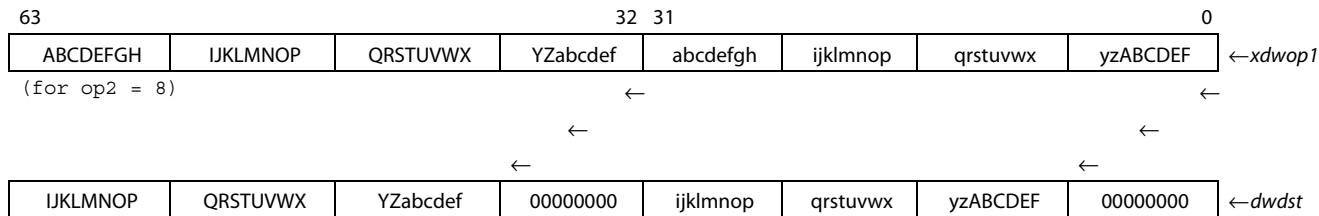
**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	0	0	0	s	p		

4                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src2, src1, dst	xdwop1, op2, dwdst	.S1 or .S2	110011
src2, src1 ,dst	xdwop1, ucst5, dwdst	.S1 or .S2	110010

**Description**    The DSHL instruction shifts the two 32-bit values in *xdwop1* to the right by *op2* bits. Both values are shifted by the same shift count. Bits shifted past bit locations 0 and 32 are lost, and the result is zero extended. Only the lower 6 bits of *op2* are used for the shift count. Bits 6-31 are ignored. A shift count between 32 and 63 will produce the same result as a shift by 31.



**Execution**    *src2\_e << src1\_e -> dst\_e*  
*src2\_o << src1\_o -> dst\_o*

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

---

**See Also** [SHL](#), [DSHRU](#), [DSHR](#)

**Example**

```
A1 == 0x1234abff
A0 == 0xff333415
DSHL .S A1:A0,16,A15:A14
A15 == 0xabff0000
A14 == 0x34150000

A1 == 0x1234abff
A0 == 0xff333415
A2 == 0x00000010
DSHL .S A1:A0,A2,A15:A14
A15 == 0xabff0000
A14 == 0x34150000

A1 == 0x00000009
A0 == 0x419751A5
A2 == 0x00000020
DSHL .S A1:A0,A2,A15:A14
A15 == 0x00000000
A14 == 0x00000000
```

## 4.133 DSHL2

4-Way SIMD Shift Left, Packed Signed 16-bit

**Syntax**    **DSHL2 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	0	0	0	s	p		

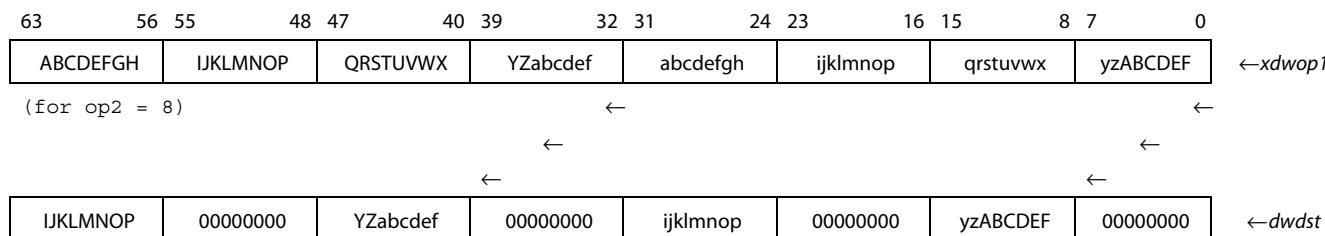
4                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,op2,dwdst	.S1 or .S2	010010
src2,src1,dst	xdwop1,ucst5,dwdst	.S1 or .S2	000110

**Description**    The DSHL2 instruction performs a left shift on packed 16-bit quantities. The values in *xdwop1* are viewed as four packed 16-bit quantities. The lower four bits of *op2* or *ucst5* are treated as a shift amount. The same shift amount is applied to all four input data. The results are placed in a signed packed 16-bit format.

For each unsigned 16-bit quantity in *xdwop1*, the quantity is shifted left by the specified number of bits. Bits shifted out of the most-significant bit of each 16-bit quantity are discarded.

For correct operation bit 4 (the fifth bit) of the constant field (*ucst5*) or register field (*op2*) must be set to 0.



**Execution**    `smsbl6(src2_e) << src1 -> smsbl6(dst_e)`  
`slsb16(src2_e) << src1 -> slsb16(dst_e)`  
`smsbl6(src2_o) << src1 -> smsbl6(dst_o)`  
`slsb16(src2_o) << src1 -> slsb16(dst_o)`

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [DSHL](#), [DSHR2](#)

**Example**

```

A1 == 0xFEDC7A98
A0 == 0x1234fedc
DSHL2 .S A1:A0,4,A3:A2
A3 == 0xEDC0A980
A2 == 0x2340edc0

A1 == 0xFEDC7A98
A0 == 0x1234fedc
A1 == 0x4
DSHL2 .S A1:A0,A1,A3:A2
A3 == 0xEDC0A980
A2 == 0x2340edc0

A1 == 0xFEDC7A98
A0 == 0x1234fedc
A1 == 0x16
ru .S A1:A0,A1,A3:A2
A3 == 0x00000000
A2 == 0x00000000

```

## 4.134 DSHR

2-Way SIMD Shift Right, Packed Signed 32-bit

**Syntax**    **DSHR (.unit) src1, src2, dst**

unit = .S1 or .S2

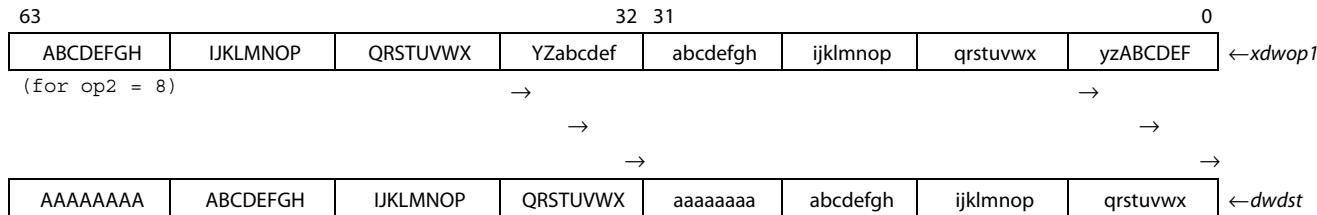
**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	0	0	0	s	p		

4                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,op2,dwdst	.S1 or .S2	110111
src2,src1,dst	xdwop1,ucst5,dwdst	.S1 or .S2	110110

**Description**    The DSHR instruction shifts the two signed 32-bit values in *xdwop1* to the right by *op2* bits. Both values are shifted by the same shift count. Bits shifted past bit locations 0 and 32 are lost, and the result is sign extended. When a register is used, the five LSBs specify the shift amount and valid values are 0-31 for correct operations. When an immediate value is used, valid shift amounts are 0-31 for correct operation.



**Execution**     $\begin{aligned} \text{src2\_e} &\gg s \text{ src1\_e} \rightarrow \text{dst\_e} \\ \text{src2\_o} &\gg s \text{ src1\_o} \rightarrow \text{dst\_o} \end{aligned}$

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

---

**See Also** [SHR](#), [DSHRS](#), [DSHL](#)

**Example**

```
A1 == 0x1234abff
A0 == 0xff333415
DSHR .S A1:A0,16,A15:A14
A15 == 0x00001234
A14 == 0xffffffff33

A1 == 0x1234abff
A0 == 0xff333415
A2 == 0x00000010
DSHR .S A1:A0,A2,A15:A14
A15 == 0x00001234
A14 == 0xffffffff33

A1 == 0x1234abff
A0 == 0xff333415
A2 == 0x00000020
DSHR .S A1:A0,A2,A15:A14
A15 == 0x00000000
A14 == 0xffffffff
```

## 4.135 DSHR2

4-Way SIMD Shift Right, Packed Signed 16-bit

**Syntax**    **DSHR2 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**    Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	1	1	opfield	1	1	0	0	s	p		

4                        5                        5                        2                        4                        4

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,op2,dwdst	.S1 or .S2	0111

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opcode	1	0	0	0	s	p		

4                        5                        5                        6                        4

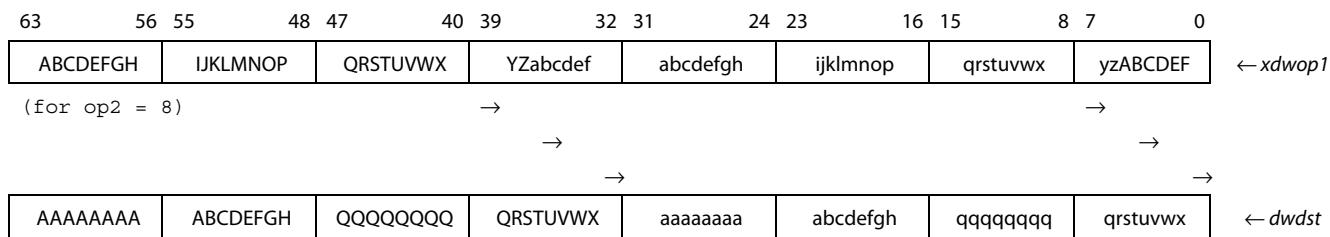
Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,ucst5,dwdst	.S1 or .S2	011000

**Description**

The DSHR2 instruction performs an arithmetic shift right on signed packed 16-bit quantities. The values in *xdwop1* are viewed as four signed packed 16-bit quantities. The lower four bits of *op2* or *ucst5* are treated as a shift amount. The same shift amount is applied to all four input data. The results are placed in a signed packed 16-bit format.

For each signed 16-bit quantity in *xdwop1*, the quantity is shifted right by the specified number of bits. The shifted quantity is sign-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of each signed 16-bit quantity are discarded.

For correct operation bit 4 (the fifth bit) of the constant field (*ucst5*) or register field (*op2*) must be set to 0.



**Execution**

```
smsb16(src2_e) >>s src1 -> smsb16(dst_e)
slsb16(src2_e) >>s src1 -> slsb16(dst_e)
smsb16(src2_o) >>s src1 -> smsb16(dst_o)
slsb16(src2_o) >>s src1 -> slsb16(dst_o)
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [DSHR](#), [DSHRU2](#)

**Example**

```
A1 == 0xFEDC7A98
A0 == 0x1234fedc
DSHR2 .S A1:A0,4,A15:A14
A15 == 0xFFED07A9
A14 == 0x0123ffed

A1 == 0xFEDC7A98
A0 == 0x1234fedc
A2 == 0x00000010
DSHR2 .S A1:A0,A2,A15:A14
A15 == 0xffff0000
A14 == 0x0000ffff

A1 == 0xFEDC7A98
A0 == 0x1234fedc
A2 == 0x00000044
DSHR2 .S A1:A0,A2,A15:A14
A15 == 0xffed07a9
A14 == 0x0123ffed
```

## 4.136 DSHRU

2-Way SIMD Shift Right, Packed Unsigned 32-bit

**Syntax**    **DSHRU (.unit) src1, src2, dst**

unit = .S1 or .S2

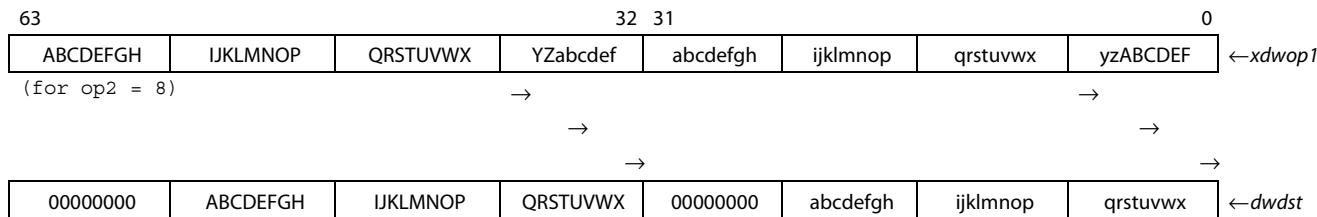
**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	0	0	0	s	p				

4                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,op2,dwdst	.S1 or .S2	100111
src2,src1,dst	xdwop1,ucst5,dwdst	.S1 or .S2	100110

**Description**    The DSHRU instruction shifts the two unsigned 32-bit values in *xdwop1* to the right by op2 bits. Both values are shifted by the same shift count. Bits shifted past bit locations 0 and 32 are lost, and the result is zero extended. When a register is used, the five LSBs specify the shift amount and valid values are 0-31 for correct operations. When an immediate value is used, valid shift amounts are 0-31 for valid operations.



**Execution**     $\text{src2\_e} \gg z \text{ src1\_e} \rightarrow \text{dst\_e}$   
 $\text{src2\_o} \gg z \text{ src1\_o} \rightarrow \text{dst\_o}$

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

---

**See Also** [SHRU](#), [DSHR](#), [DSHL](#)

**Example**

```
A1 == 0x1234abff
A0 == 0xff333415
DSHRU .S A1:A0,16,A15:A14
A15 == 0x00001234
A14 == 0x0000ff33

A1 == 0x1234abff
A0 == 0xff333415
A2 == 0x10
DSHRU .S A1:A0,A2,A15:A14
A15 == 0x00001234
A14 == 0x0000ff33

A1 == 0x1234abff
A0 == 0xff333415
A2 == 0x50
DSHRU .S A1:A0,A2,A15:A14
A15 == 0x00001234
A14 == 0x0000ff33
```

## 4.137 DSHRU2

4-Way SIMD Shift Right, Packed Unsigned 16-bit

**Syntax**    **DSHRU2 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**    Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	1	1	opfield	1	1	0	0	s	p		

4                        5                        5                        2                        4                        4

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,op2,dwdst	.S1 or .S2	1000

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	0	0	0	s	p		

4                        5                        5                        6                        4

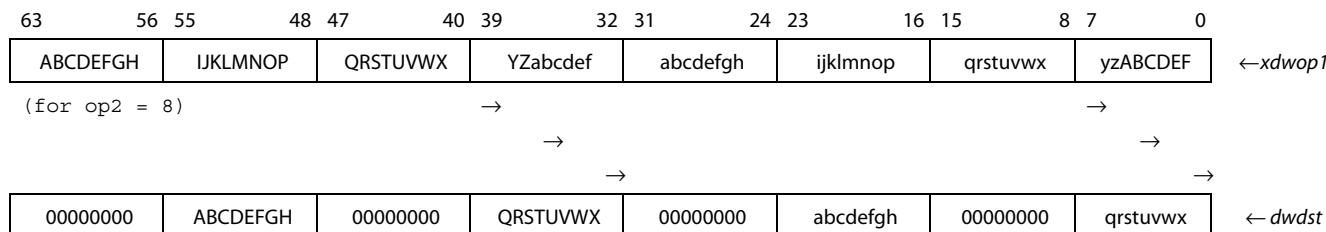
Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xdwop1,ucst5,dwdst	.S1 or .S2	011001

**Description**

The DSHRU2 instruction performs an shift right on unsigned packed 16-bit quantities. The values in *xdwop1* are viewed as four unsigned packed 16-bit quantities. The lower four bits of *op2* or *ucst5* are treated as a shift amount. The same shift amount is applied to all four input data. The results are placed in a signed packed 16-bit format.

For each unsigned 16-bit quantity in *xdwop1*, the quantity is shifted right by the specified number of bits. The shifted quantity is zero-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of each signed 16-bit quantity are discarded.

For correct operation bit 4 (the fifth bit) of the constant field (*ucst5*) or register field (*op2*) must be set to 0.



**Execution**    `smsb16(src2_e) >>z src1 -> smsb16(dst_e)`  
`slsb16(src2_e) >>z src1 -> slsb16(dst_e)`  
`smsb16(src2_o) >>z src1 -> smsb16(dst_o)`  
`slsb16(src2_o) >>z src1 -> slsb16(dst_o)`

**Instruction Type**    Single-cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [DSHR](#), [DSHR2](#)

**Example**

```

A1 == 0xFEDC7A98
A0 == 0x1234fedc
DSHRU2 .S A1:A0,4,A15:A14
A15 == 0x0FED07A9
A14 == 0x01230fed

A1 == 0xFEDC7A98
A0 == 0x1234fedc
A2 == 0x00000004
DSHRU2 .S A1:A0,A2,A15:A14
A15 == 0x0FED07A9
A14 == 0x01230fed

A1 == 0xFEDC7A98
A0 == 0x1234fedc
A2 == 0x00000044
DSHRU2 .S A1:A0,A2,A15:A14
A15 == 0x0FED07A9
A14 == 0x01230fed

```

## 4.138 DSMPY2

4-Way SIMD Multiply Signed by Signed With Left Shift and Saturation, Packed Signed 16-bit

**Syntax**    **DSMPY2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	0	opfield	1	1	0	0	s	p		

4                        5                        5                        5                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,qwdst	.M1 or .M2	00001

**Description**

The DSMPY2 instruction performs 16-bit multiplication between signed packed 16-bit quantities, with an additional left-shift and saturate. The values in *dwop1* and *xdwop2* are treated as signed packed 16-bit quantities. The two 32-bit results are placed in two 64-bit register pairs.

The DSMPY2 instruction produces four 16 x 16 products. Each product is shifted left by one, and if the left-shifted result is equal to 0x80000000, the output value is saturated to 0x7FFFFFFF. If any product saturates, the SAT bit is set in the CSR on the cycle the result is written. If no product saturates, the SAT bit is left unaffected.

**Execution**

```
sat((lsb16(src1_e) x lsb16(src2_e)) << 1) -> dst_0
sat((msb16(src1_e) x msb16(src2_e)) << 1) -> dst_1
sat((lsb16(src1_o) x lsb16(src2_o)) << 1) -> dst_2
sat((msb16(src1_o) x msb16(src2_o)) << 1) -> dst_3
```

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [MPY2](#), [SMPYH](#), [SMPY](#)

**Example**

```
CSR == 0x00000000
A13 == 0xFFFFFFFF
A12 == 0xFFFFFFFF
A23 == 0x00010001
A22 == 0x00010001
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0xFFFFFFFF
A10 == 0xFFFFFFFF
A9 == 0xFFFFFFFF
A8 == 0xFFFFFFFF

A13 == 0xFFFFFFFF
A12 == 0xFFFFFFFF
A23 == 0xFFFFFFFF
A22 == 0xFFFFFFFF
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0x00000002
A10 == 0x00000002
A9 == 0x00000002
```

```

A8 == 0x00000002
A13 == 0xFFFFFFFF
A12 == 0xFFFFFFFF
A23 == 0xAAAAAAA
A22 == 0xAAAAAAA
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0x0000AAC
A10 == 0x0000AAC
A9 == 0x0000AAC
A8 == 0x0000AAC

A13 == 0x7FFF7FFF
A12 == 0x7FFF7FFF
A23 == 0xAAAAAAA
A22 == 0xAAAAAAA
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0xAAAAAAAC
A10 == 0xAAAAAAAC
A9 == 0xAAAAAAAC
A8 == 0xAAAAAAAC

A13 == 0x80018001
A12 == 0x80018001
A23 == 0x80018001
A22 == 0x80018001
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0x7ffe0002
A10 == 0x7ffe0002
A9 == 0x7ffe0002
A8 == 0x7ffe0002

A13 == 0x7fff7fff
A12 == 0x7fff7fff
A23 == 0x7fff7fff
A22 == 0x7fff7fff
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0x7ffe0002
A10 == 0x7ffe0002
A9 == 0x7ffe0002
A8 == 0x7ffe0002

A13 == 0xc333c333
A12 == 0xc333c333
A23 == 0x3ccdc333
A22 == 0x3ccdc333
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0xe31e87ae
A10 == 0x1ce17852
A9 == 0xe31e87ae
A8 == 0x1ce17852

A13 == 0x00000001
A12 == 0x00000001
A23 == 0x80008000
A22 == 0x80008000
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0x00000000
A10 == 0xfffff000
A9 == 0x00000000
A8 == 0xfffff000

CSR= 0x00000000
A13 == 0x80008000
A12 == 0x80008000
A23 == 0x80008000
A22 == 0x80008000
DSMPY2 .M A13:A12,A23:A22,A11:A10:A9:A8
A11 == 0x7FFFFFFF
A10 == 0x7FFFFFFF
A9 == 0x7FFFFFFF
A8 == 0x7FFFFFFF

CSR= 0x00000200
CSR == 0x00000000
B13 == 0xFFFFFFFF
B12 == 0xFFFFFFFF
B23 == 0x00010001
B22 == 0x00010001
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0xFFFFFFF

```

```

B10 == 0xFFFFFFFF
B9 == 0xFFFFFFFF
B8 == 0xFFFFFFFF

B13 == 0xFFFFFFFF
B12 == 0xFFFFFFFF
B23 == 0xFFFFFFFF
B22 == 0xFFFFFFFF
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0x00000002
B10 == 0x00000002
B9 == 0x00000002
B8 == 0x00000002

B13 == 0xFFFFFFFF
B12 == 0xFFFFFFFF
B23 == 0xAAAAAAA
B22 == 0xAAAAAAA
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0x0000AAC
B10 == 0x0000AAC
B9 == 0x0000AAC
B8 == 0x0000AAC

B13 == 0x7FFF7FFF
B12 == 0x7FFF7FFF
B23 == 0xAAAAAAA
B22 == 0xAAAAAAA
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0xAAAAAAAC
B10 == 0xAAAAAAAC
B9 == 0xAAAAAAAC
B8 == 0xAAAAAAAC

B13 == 0x80018001
B12 == 0x80018001
B23 == 0x80018001
B22 == 0x80018001
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0x7ffe0002
B10 == 0x7ffe0002
B9 == 0x7ffe0002
B8 == 0x7ffe0002

B13 == 0x7fff7fff
B12 == 0x7fff7fff
B23 == 0x7fff7fff
B22 == 0x7fff7fff
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0x7ffe0002
B10 == 0x7ffe0002
B9 == 0x7ffe0002
B8 == 0x7ffe0002

B13 == 0xc333c333
B12 == 0xc333c333
B23 == 0x3ccdc333
B22 == 0x3ccdc333
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0xe31e87ae
B10 == 0x1ce17852
B9 == 0xe31e87ae
B8 == 0x1ce17852

B13 == 0x00000001
B12 == 0x00000001
B23 == 0x80008000
B22 == 0x80008000
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0x00000000
B10 == 0xfffff0000
B9 == 0x00000000
B8 == 0xfffff0000

CSR= 0x00000000
B13 == 0x80008000
B12 == 0x80008000
B23 == 0x80008000
B22 == 0x80008000
DSMPY2 .M B13:B12,B23:B22,B11:B10:B9:B8
B11 == 0x7FFFFFFF

```

```
B10 == 0x7FFFFFFF
B9 == 0x7FFFFFFF
B8 == 0x7FFFFFFF

CSR= 0x00000200
```

## 4.139 DSPACKU4

2-Way SIMD Saturate and Pack Into Unsigned Packed 8-bit

**Syntax**    **DSPACKU4 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**    Opcode for .S Unit, 2 src fixed hdr

31	30	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	1	1	opfield	1	1	0	0	s	p		

4                        5                        5                        2                        4                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	0100

**Description**    Precisely equivalent to executing two SPACK4 instructions in parallel.  
E.g., DPACKU4

**Execution**

```

if (msb16(src1_e) >= 0000_00FFh), FFh -> ubyte3(dst_e) or
if (msb16(src1_e) << 0), 0 -> ubyte3(dst_e)
else truncate(msb16(src1_e)) -> ubyte3(dst_e);
if (lsb16(src1_e) >> 0000_00FFh), FFh -> ubyte2(dst_e) or
if (lsb16(src1_e) << 0), 0 -> ubyte2(dst_e)
else truncate(lsb16(src1_e)) -> ubyte2(dst_e);
if (msb16(src2_e) >= 0000_00FFh), FFh -> ubyte1(dst_e) or
if (msb16(src2_e) << 0), 0 -> ubyte1(dst_e)
else truncate(msb16(src2_e)) -> ubyte1(dst_e);
if (lsb16(src2_e) >> 0000_00FFh), FFh -> ubyte0(dst_e) or
if (lsb16(src2_e) << 0), 0 -> ubyte0(dst_e)
else truncate(lsb16(src2_e)) -> ubyte0(dst_e)
if (msb16(src1_o) >= 0000_00FFh), FFh -> ubyte3(dst_o) or
if (msb16(src1_o) << 0), 0 -> ubyte3(dst_o)
else truncate(msb16(src1_o)) -> ubyte3(dst_o);
if (lsb16(src1_o) >> 0000_00FFh), FFh -> ubyte2(dst_o) or
if (lsb16(src1_o) << 0), 0 -> ubyte2(dst_o)
else truncate(lsb16(src1_o)) -> ubyte2(dst_o);
if (msb16(src2_o) >= 0000_00FFh), FFh -> ubyte1(dst_o) or
if (msb16(src2_o) << 0), 0 -> ubyte1(dst_o)
else truncate(msb16(src2_o)) -> ubyte1(dst_o);
if (lsb16(src2_o) >> 0000_00FFh), FFh -> ubyte0(dst_o) or
if (lsb16(src2_o) <= 0), 0 -> ubyte0(dst_o)
else truncate(lsb16(src2_o)) -> ubyte0(dst_o)

```

**Instruction Type**    Single cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also** [SPACK2](#), [PACKL4](#), [PACKH4](#)

**Example**

```
A3 == 0x00120034
A2 == 0x00560078
A1 == 0x0089009a
A0 == 0x00ab00cd
DSPACKU4 .S A3:A2,A1:A0,A7:A6
A7 == 0x899aabcd
A6 == 0x12345678
```

## 4.140 DSPINT

2-Way SIMD Convert Single Precision Floating Point to Signed 32-bit Integer

**Syntax**    **DSPINT (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		scst5		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                        5                        5                        10

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xdwop,dwdst	.L1 or .L2	10100

**Opcode**    Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src		opfield	x	1	1	1	1	0	0	1	0	0	0	s	p	

3                        5                        5                        10

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xdwop,dwdst	.S1 or .S2	10100

**Description**    Converts two single precision values packed in src2 to two integer values packed in dst.

**Execution**

```
if(cond){
  int(src2_e) -> dst_e
  int(src2_o) -> dst_o
}
else nop
```

**Instruction Type**    3-cycle

**Delay Slots**    2

**Functional Unit Latency**    1

---

**See Also** DPINT, DPTRUNC, INTDP, INTDPU, INTSP, INTSPU, SPRUNC

**Example**

```
FADCR== 0x00000000
A7 == 0x4109999a
A6 == 0x7f900000
DSPINT .L A7:A6,A3:A2 ; SNaN
A3 == 0x00000009
A2 == 0x7fffffff

FADCR= 0x00000092

FADCR== 0x00000200
A7 == 0x4109999a
A6 == 0x7fc00000
DSPINT .L A7:A6,A3:A2 ; QNaN
A3 == 0x00000008
A2 == 0x7fffffff

FADCR= 0x00000292

FADCR== 0x00000400
A7 == 0x4109999a
A6 == 0x4effffff
DSPINT .L A7:A6,A3:A2
A3 == 0x00000009
A2 == 0x7fffff80

FADCR= 0x00000480

FADCR== 0x00000600
```

## 4.141 DSPINTH

2-Way SIMD Convert Single Precision Floating Point to Signed 16-bit Integer

**Syntax** **DSPINTH (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, or .S2

**Opcode** Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		scst5		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                    5                    5                    10

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xdwop,dst	.L1 or .L2	10000

**Opcode** Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src		opfield	x	1	1	1	1	0	0	1	0	0	0	s	p	

3                    5                    5                    10

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xdwop,dst	.S1 or .S2	10000

**Description** Converts two single precision floating point value in src2\_e and src2\_o to two signed 16-bit integer values packed in dst.

**Execution**

```
if(cond){
    int(src2_e) -> lsb16(dst)
    int(src2_o) -> msb16(dst)
}
else nop
```

**Instruction Type** 3-cycle

**Delay Slots** 2

**Functional Unit Latency** 1

---

**See Also** [DPINT](#), [DPTRUNC](#), [INTDP](#), [INTDPU](#), [INTSP](#), [INTSPU](#), [SPTRUNC](#)

**Example**

```
FADCR== 0x00000000
A1    == 0x4109999a
A0    == 0x4109999a
DSPINTH .L A1:A0,A2
A2 == 0x00090009

FADCR= 0x00000080

FADCR== 0x00000200
```

## 4.142 DSSUB

2-Way SIMD Saturating Subtract, Packed Signed 32-bit

**Syntax**    **DSSUB (.unit) src1, src2, dst**

unit = .L1 or .L2

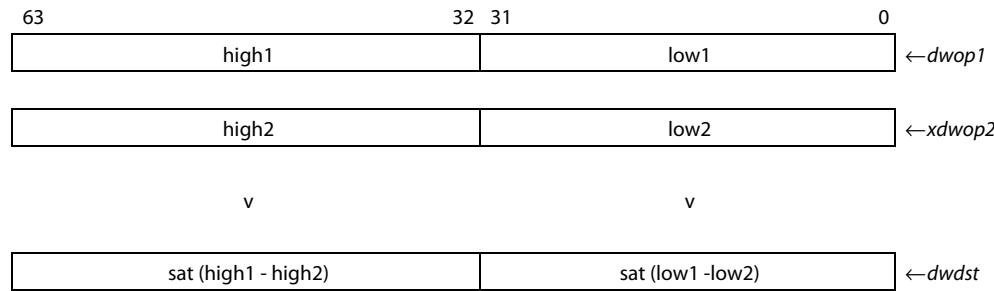
**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield	1	1	0	s	p		

4                                5                                5                                7                                3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1100111

**Description**    The DSSUB instruction performs two saturating 32-bit subtracts of the packed 32-bit numbers contained in the two source register pairs. The subtraction results are saturated to the range  $-2^{31}$  to  $2^{31}-1$  and returned as two 32-bit results packed into *dwdst*.



**Execution**     $\text{sat}(\text{src1\_e} - \text{src2\_e}) \rightarrow \text{dst\_e}$   
 $\text{sat}(\text{src1\_o} - \text{src2\_o}) \rightarrow \text{dst\_o}$

**Instruction Type**    Single cycle

**Delay Slots**    0

**Functional Unit Latency**    1

---

**See Also** [DSUB](#), [SSUB](#), [DSUB2](#)
**Example**

```
A5 == 0x5a2e51a3
A4 == 0x436771f2
A9 == 0x802a3fa2
A8 == 0x5a2e51a3
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0xe939204f
```

```
A5 == 0xffffffff0
A4 == 0x7fffffff4
A9 == 0x7fffffff4
A8 == 0xffffffff0
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x80000000
A0 == 0x7fffffff
```

```
A5 == 0x7fffffff4
A4 == 0x7fffffff
A9 == 0x0fffffff0
A8 == 0xfffffffff
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x70000004
A0 == 0x7fffffff
```

```
A5 == 0x7fffffff0
A4 == 0x00000000
A9 == 0x80000000
A8 == 0xfffffffff
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x00000001
```

```
A5 == 0x00000000
A4 == 0x00000000
A9 == 0x80000000
A8 == 0x7fffffff
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x7fffffff
A0 == 0x80000001
```

```
A5 == 0xfffffffff
A4 == 0xfffffffef
A9 == 0x7fffffff
A8 == 0x7fffffff
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x80000000
A0 == 0x80000000
```

```
A5 == 0xffffffff0
A4 == 0x7fffffff4
A9 == 0x7fffffff4
A8 == 0xffffffff0
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x80000000
A0 == 0x7fffffff
```

```
A5 == 0xfffffffff
A4 == 0xfffffffef
A9 == 0x0fffffff
A8 == 0xffffffffe
DSSUB . A5:A4,A9:A8,A1:A0
A1 == 0x00000000
A0 == 0x00000001
```

## 4.143 DSSUB2

4-Way SIMD Saturating Subtract, Packed Signed 16-bit

**Syntax** **DSSUB2** (.unit) *src1, src2, dst*

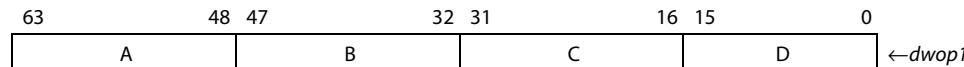
unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst	5	src2	5	src1	x	opfield	7	1	1	0	s	p	4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1100100

**Description** The DSSUB2 instruction performs four saturating 16-bit subtractions of the packed signed 16-bit numbers contained in the two 64-bit source registers. The subtraction results are returned as four signed 16-bit results packed into *dst*. Results are saturated to within the range  $-2^{15}$  to  $2^{15}-1$ .



v v v v



**Execution**

```

sat(msb16(src1_e) - msb16(src2_e)) -> msb16(dst_e);
sat(lsb16(src1_e) - lsb16(src2_e)) -> lsb16(dst_e);
sat(msb16(src1_o) - msb16(src2_o)) -> msb16(dst_o);
sat(lsb16(src1_o) - lsb16(src2_o)) -> lsb16(dst_o);

```

**Instruction Type** Single cycle

**Delay Slots** 0

**Functional Unit Latency** 1

---

**See Also** [DSUB](#), [DSSUB](#), [DSSUB2](#), [SUB2](#)

**Example**

```
A1 == 0x00080007
A0 == 0x00060005
A3 == 0x00010001
A2 == 0x00010001
DSSUB2 .L A1:A0,A3:A2,A15:A14
A15 == 0x00070006
A14 == 0x00050004

A1 == 0xffffffff
A0 == 0xfffffffff
A3 == 0xfffffffff
A2 == 0xfffffffffe
DSSUB2 .L A1:A0,A3:A2,A15:A14
A15 == 0x00000000
A14 == 0x00000001

A1 == 0x80008000
A0 == 0x80008000
A3 == 0x00010001
A2 == 0xfffffffff
DSSUB2 .L A1:A0,A3:A2,A15:A14
A15 == 0x80008000
A14 == 0x80018001
```

## 4.144 DSUB

2-Way SIMD Subtract, Packed Signed 32-bit

**Syntax**    **DSUB (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	0	0	0	s	p				

4                         5                         5                         6                         4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	010111

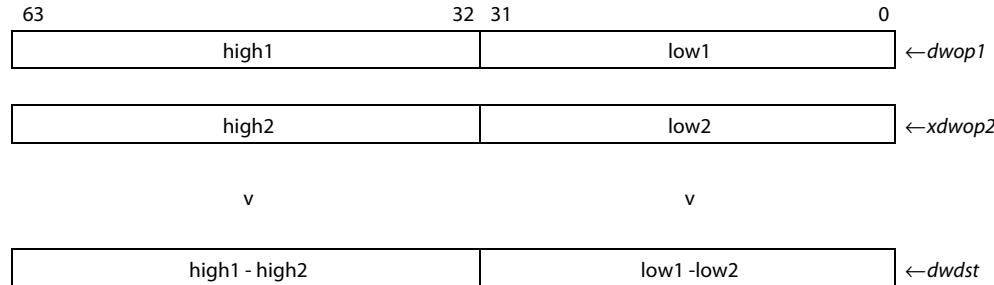
**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	1	0	s	p				

4                         5                         5                         7                         3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0000111

**Description**    The DSUB instruction performs two 32-bit subtractions of the packed 32-bit numbers contained in the two source register pairs. The subtraction results are returned as two 32-bit results packed into *dwdst*.



**Execution**     $\text{src1\_e} - \text{src2\_e} \rightarrow \text{dst\_e}$   
 $\text{src1\_o} - \text{src2\_o} \rightarrow \text{dst\_o}$

**Instruction Type**    Single cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also** [SUB, DSUB2](#)

**Example**

```
A1 == 0x44444444
A0 == 0x80000000
A3 == 0xccccc444
A2 == 0xfffffffff
DSUB .L A1:A0,A3:A2,A15:A14
A15 == 0x77778000
A14 == 0x80000001

A1 == 0x7fffffff
A0 == 0x7fffffffC
A3 == 0x7fffffffE
A2 == 0x7fffffffE
DSUB .L A1:A0,A3:A2,A15:A14
A15 == 0x00000001
A14 == 0x7fffffffE

A1 == 0x80000001
A0 == 0x80000000
A3 == 0xfffffffFc
A2 == 0x00000001
DSUB .L A1:A0,A3:A2,A15:A14
A15 == 0x80000005
A14 == 0x7fffffff

A1 == 0x7fffffff
A0 == 0xfffffffF
A3 == 0xfffffffF
A2 == 0x7fffffff
DSUB .L A1:A0,A3:A2,A15:A14
A15 == 0x80000000
A14 == 0x80000000
```

## 4.145 DSUB2

4-Way SIMD Subtract, Packed Signed 16-bit

**Syntax**    **DSUB2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	0	0	0	s	p				

4                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	010001

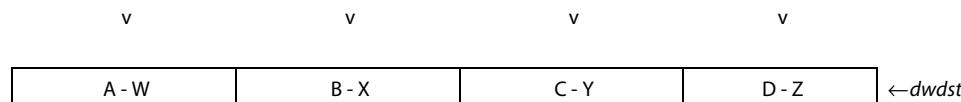
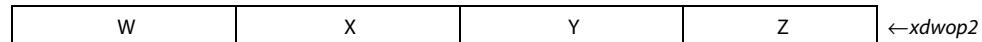
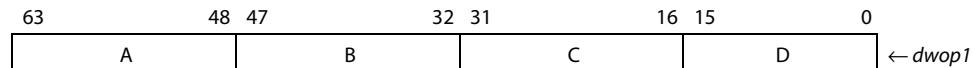
**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst	src2	src1	x	opfield	1	1	0	s	p				

4                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0000100

**Description**    The DSUB2 instruction performs four 16-bit subtractions of the packed 16-bit numbers contained in the two 64-bit wide source registers. The subtraction results are returned as four 16-bit results packed into *dst*.



**Execution**      `msbl6(src1_e) - msbl6(src2_e) -> msbl6(dst_e);  
                   lsbl6(src1_e) - lsbl6(src2_e) -> lsbl6(dst_e);  
                   msbl6(src1_o) - msbl6(src2_o) -> msbl6(dst_o);  
                   lsbl6(src1_o) - lsbl6(src2_o) -> lsbl6(dst_o);`

**Instruction Type**    Single cycle

**Delay Slots**        0

**Functional Unit Latency**    1

**See Also**            [DSUB](#), [DSSUB](#), [DSSUB2](#), [SUB2](#)

**Example**            `A1 == 0x44444444  
                   A0 == 0x7fff7fff  
                   A3 == 0xcccccc444  
                   A2 == 0x00020005  
                   DSUB2 .L A1:A0,A3:A2,A15:A14  
                   A15 == 0x77788000  
                   A14 == 0x7ffd7ffa`

## 4.146 DSUBSP

2-Way SIMD Subtract, Packed Single Precision Floating Point

**Syntax**    **DSUBSP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield		1	1	0	s	p	

4                         5                         5                         5                         7                         3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	0111101

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1	dst		src2		src1	x	opfield		1	0	0	0	s	p	

4                         5                         5                         5                         6                         4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	101101

**Description**    Performs a SIMD single-precision floating point Subtract on the pairs of numbers in dwop1 and xdwop2. The following are equivalent:

DSUBSP A1:A0, A3:A2, A5:A4

and

```
FSUBSP A1, A3, A5
FSUBSP A0, A2, A4
```

**Execution**

```
if(cond) {
    src1_e + src2_e -> dst_e
    src1_o + src2_o -> dst_o
}
else nop
```

**Instruction Type**    3-cycle

**Delay Slots**    2

**Functional Unit Latency**    1

**See Also** ADDDP, SUBDP, SUBSP, FADDDP, FSUBSP, FSUBDP, DADDSP

**Example**

```

FADCR== 0x00000000
A5 == 0x41200000
A4 == 0x40784517
A7 == 0x42300000
A6 == 0x42aeab2d
DSUBSP .L A5:A4,A7:A6,A1:A0 ; 10 - 44 =54 ; 3.8792169094E+00 - 8.7334327698E+01
=-8.3455110788E+01
A1 == 0xc2080000
A0 == 0xc2a6e904

FADCR= 0x00000080

FADCR== 0x02000000
B3 == 0x40784517
B2 == 0x7fc00000
B7 == 0x42aeab2d
B6 == 0x42300000
DSUBSP .L B3:B2,B7:B6,B1:B0 ; low:QNan
B1 == 0xc2a6e904
B0 == 0x7fffffff

FADCR= 0x02810000

FADCR== 0x00000400
A5 == 0x40784517
A4 == 0x41200000
A7 == 0x42aeab2d
A6 == 0x7f900000
DSUBSP .L A5:A4,A7:A6,A1:A0 ; low:SNan
A1 == 0xc2a6e904
A0 == 0x7fffffff

FADCR= 0x00000492

FADCR== 0x06000000
B3 == 0x40784517
B2 == 0x7f800000
B7 == 0x42aeab2d
B6 == 0x7f800000
DSUBSP .L B3:B2,B7:B6,B1:B0 ; low: Inf-Inf=Nan_out
B1 == 0xc2a6e905
B0 == 0x7fffffff

FADCR= 0x06900000

FADCR== 0x06000000
B3 == 0x40784517
B2 == 0x7f800000
B7 == 0x42aeab2d
B6 == 0x42300000
DSUBSP .L B3:B2,B7:B6,B1:B0 ; low: Inf-xxx=Inf
B1 == 0xc2a6e905
B0 == 0x7f800000

FADCR= 0x06a00000

FADCR== 0x00000000
A5 == 0x7f7fffff
A4 == 0xff7fffff
A7 == 0xff780123
A6 == 0x7f780123
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +Inf ; -Inf
A1 == 0x7f800000
A0 == 0xff800000

FADCR= 0x000000e0

FADCR== 0x00000200
A5 == 0x7f7fffff
A4 == 0xff7fffff
A7 == 0xff780123
A6 == 0x7f780123
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +LFPN ; -LFPN
A1 == 0x7f7fffff
A0 == 0xff7fffff

FADCR= 0x000002c0

```

```

FADCR== 0x00000400
A5 == 0x7f7fffff
A4 == 0xffff7fffff
A7 == 0xffff780123
A6 == 0x7f780123
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +Inf ; -LFPN
A1 == 0x7f800000
A0 == 0xffff7fffff

FADCR= 0x000004e0

FADCR== 0x00000600
A5 == 0x7f7fffff
A4 == 0xffff7fffff
A7 == 0xffff780123
A6 == 0x7f780123
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +LFPN ; -Inf
A1 == 0x7f7fffff
A0 == 0xffff800000

FADCR= 0x000006e0

FADCR== 0x00000000
A5 == 0x008d8000
A4 == 0x808d8000
A7 == 0x0008d0000
A6 == 0x808d0000
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +0 ; -0
A1 == 0x00000000
A0 == 0x80000000

FADCR= 0x00000180

FADCR== 0x00000400
A5 == 0x008d8000
A4 == 0x808d8000
A7 == 0x0008d0000
A6 == 0x808d0000
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +SFPN ; -0
A1 == 0x00800000
A0 == 0x80000000

FADCR= 0x00000580

FADCR== 0x00000600
A5 == 0x008d8000
A4 == 0x808d8000
A7 == 0x0008d0000
A6 == 0x808d0000
DSUBSP .L A5:A4,A7:A6,A1:A0 ; +0 ; -SFPN
A1 == 0x00000000
A0 == 0x80800000

FADCR= 0x00000780

```

## 4.147 DXPND2

Expand Bits to Packed 16-bit Masks

**Syntax**    **DXPND2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src		opfield	x	0	0	0	0	1	1	1	1	0	0	s	p	

3                                5                                5                                10

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.M1 or .M2	10001

**Description**

The DXPND2 instruction reads the four least-significant bits of *xdwop* and expands them into a two halfword mask. Bits 1 and 0 are replicated to the upper and lower halfwords of the even result register, and bits 3 and 2 are replicated to the upper and lower halfwords of the odd result register, respectively. Bits 31 through 4 of *xop* are explicitly ignored and may be non-zero.

This instruction is useful when combined with the output of [DCMPGT2](#), for generating a mask which corresponds to the individual halfword positions that were compared. Such a mask may subsequently be used with [ANDN](#), [AND](#) and [OR](#) instructions to perform compositing or other multiplexing operations.

63	48 47	32 31	16 15	4 3 2 1 0	
				$\leftarrow xop$	

v

v

AAAAAAAAAAAAAAA	BBBBBBBBBBBBBBBB	CCCCCCCCCCCCCCC	DDDDDDDDDDDDDDDD	$\leftarrow dwdst$
-----------------	------------------	-----------------	------------------	--------------------

**Execution**

```
if(cond) {
    if(src2 & 1) 0xFFFF -> lsb16(dst_e)
    else 0x0000 -> lsb16(dst_e)

    if(src2 & 2) 0xFFFF -> msb16(dst_e)
    else 0x0000 -> msb16(dst_e)

    if(src2 & 4) 0xFFFF -> lsb16(dst_o)
    else 0x0000 -> lsb16(dst_o)

    if(src2 & 8) 0xFFFF -> msb16(dst_o)
    else 0x0000 -> msb16(dst_o)}
```

```

    }
else nop

```

**Instruction Type** 2-cycle

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [DXPND4](#), [DCMPGT2](#), [XPND2](#)

**Example**

```

A0 == 0x00000000
DXPND2 .M A0,A3:A2
A3 == 0x00000000
A2 == 0x00000000

A0 == 0x00000001
DXPND2 .M A0,A3:A2
A3 == 0x00000000
A2 == 0x0000ffff

A0 == 0x00000002
DXPND2 .M A0,A3:A2
A3 == 0x00000000
A2 == 0xfffff000

A0 == 0x00000003
DXPND2 .M A0,A3:A2
A3 == 0x00000000
A2 == 0xffffffff

A0 == 0x1234567d
DXPND2 .M A0,A3:A2
A3 == 0xffffffff
A2 == 0x0000ffff

A0 == 0x89abcdee
DXPND2 .M A0,A3:A2
A3 == 0xffffffff
A2 == 0xfffff000

```

## 4.148 DXPND4

Expand Bits to Packed 8-bit Masks

**Syntax** **DXPND4 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src		opfield	x	0	0	0	0	1	1	1	1	0	0	s	p	

3                    5                    5                    5                    10

Opcode map field used...	For operand type...	Unit	Opfield
src2, dst	xop,dwdst	.M1 or .M2	10000

**Description**

The DXPND4 instruction reads the 8 least-significant bits of *xop* and expands them into an 8 byte mask. Bits 7 through 0 are replicated to bytes 7 through 0 of the result. Bits 31 through 8 of *xop* are explicitly ignored and may be non-zero.

This instruction is useful when combined with the output of [DCMPGTU4](#), for generating a mask which corresponds to the individual byte positions that were compared. Such a mask may subsequently be used with [ANDN](#), [AND](#) and [OR](#) instructions to perform compositing or other multiplexing operations.

Because DXPND4 only examines the four LSBs of *xop*, it is possible to store a large bit mask in a single 32-bit word, and expand it using multiple [SHR](#) and DXPND4 pairs. This can be useful for expanding a packed 1-bit/pixel bitmap into full 8-bit pixels.

63	56 55	48 47	40 39	32 31	24 23	16 15	8 7	6 5	4 3	2 1	0										← op
				xxxxxxxx	xxxxxxxx	xxxxxxxx	A	B	C	D	E	F	G	H							v

AAAAAAA	BBBBBBB	CCCCCCC	DDDDDDD	EEEEEEE	FFFFFFF	GGGGGGG	HHHHHHH	← dwdst
---------	---------	---------	---------	---------	---------	---------	---------	---------

**Execution**

```

if(cond) {
  if(src2 & 1) 0xFF -> byte0(dst_e)
  else 0x00 -> byte0(dst_e)

  if(src2 & 2) 0xFF -> byte1(dst_e)
  else 0x00 -> byte1(dst_e)

  if(src2 & 4) 0xFF -> byte2(dst_e)
  else 0x00 -> byte2(dst_e)

  if(src2 & 8) 0xFF -> byte3(dst_e)
  else 0x00 -> byte3(dst_e)

  if(src2 & 16) 0xFF -> byte0(dst_o)
  else 0x00 -> byte0(dst_o)

  if(src2 & 32) 0xFF -> byte1(dst_o)
  else 0x00 -> byte1(dst_o)
}

```

```

if(src2 & 64) 0xFF -> byte2(dst_o)
else 0x00 -> byte2(dst_o)

if(src2 & 128) 0xFF -> byte3(dst_o)
else 0x00 -> byte3(dst_o)

}
else nop

```

**Instruction Type** 2-cycle

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [CMPEQ4](#), [CMPGTU4](#), [XPND2](#), [XPND4](#)

**Example**

```

A0 == 0x00000000
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0x00000000

A0 == 0x00000001
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0x000000ff

A0 == 0x00000002
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0x0000ff00

A0 == 0x00000004
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0x00ff0000

A0 == 0x00000008
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0xff000000

A0 == 0x00000005
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0x000ff00ff

A0 == 0x0000000a
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0xff00ff00

A0 == 0x0000000f
DXPND4 .M A0,A1:A0
A1 == 0x00000000
A0 == 0xffffffff

A0 == 0x12345675
DXPND4 .M A0,A1:A0
A1 == 0x00ffffff
A0 == 0x00ff00ff

A0 == 0x89abcdea
DXPND4 .M A0,A1:A0
A1 == 0xfffffff00
A0 == 0xff00ff00

```

## 4.149 EXT

Extract and Sign-Extend a Bit Field

**Syntax**    **EXT (.unit) src2, csta, cstb, dst**

or

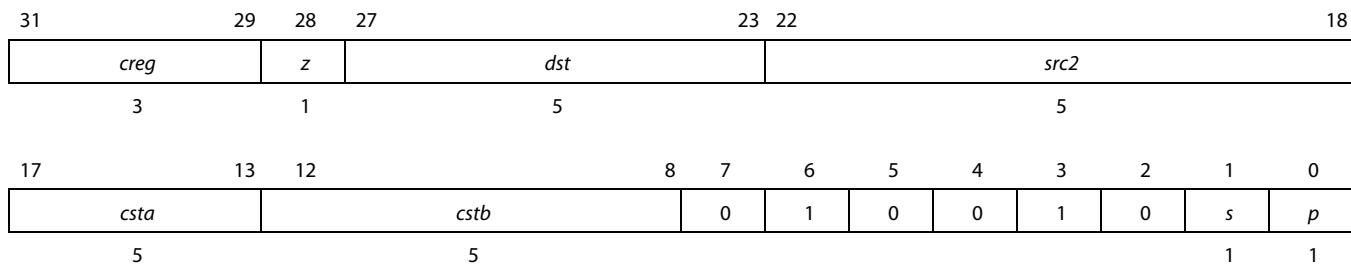
**EXT (.unit) src2, src1, dst**

unit = .S1 or .S2

**Compact Instruction Format**

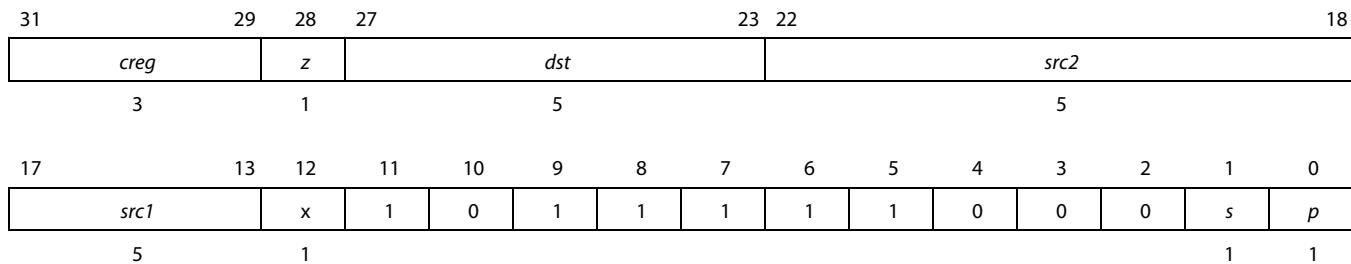
Unit	Opcode Format	Figure
.S	S2ext	<a href="#">Figure F-23</a>

**Opcode**    Constant form



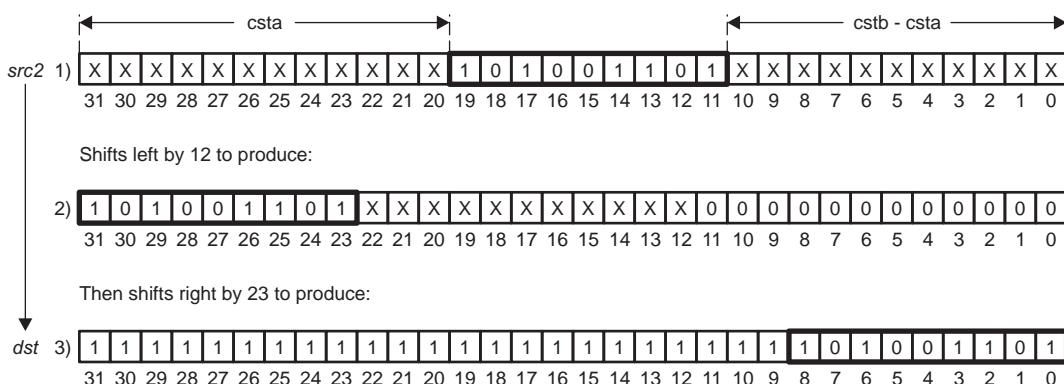
Opcode map field used...	For operand type...	Unit
<i>src2</i>	sint	.S1, .S2
<i>csta</i>	ucst5	
<i>cstb</i>	ucst5	
<i>dst</i>	sint	

**Opcode**    Register form



<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>src2</i>	xsint	.S1, .S2
<i>src1</i>	uint	
<i>dst</i>	sint	

Description	The field in <i>src2</i> , specified by <i>csta</i> and <i>cstb</i> , is extracted and sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right. <i>csta</i> and <i>cstb</i> are the shift left amount and shift right amount, respectively. This can be thought of in terms of the LSB and MSB of the field to be extracted. Then <i>csta</i> = 31 - MSB of the field and <i>cstb</i> = <i>csta</i> + LSB of the field. The shift left and shift right amounts may also be specified as the ten LSBs of the <i>src1</i> register with <i>cstb</i> being bits 0-4 and <i>csta</i> bits 5-9. In the example below, <i>csta</i> is 12 and <i>cstb</i> is $11 + 12 = 23$ . Only the ten LSBs are valid for the register version of the instruction. If any of the 22 MSBs are non-zero, the result is invalid.
-------------	--



**Execution** If the constant form is used:

```
if (cond)    src2 ext csta, cstb → dst  
else non
```

If the register form is used:

```
if (cond)    src2 ext src19..5, src14..0 → dst  
else non
```

## Pipeline

Pipeline Stage	E1
Read	$src1, src2$
Written	$dst$
Unit in use	$.S$

**Instruction Type** Single-cycle

### ***Delay Slots***

**See Also** [EXTU](#)

### *Examples*      Example 1

EXT .S1 A1,10,19,A2

Before instruction		1 cycle after instruction	
A1	07A43F2Ah	A1	07A43F2Ah
A2	xxxxxxxxh	A2	FFFFF21Fh

### Example 2

EXT .S1 A1,A2,A3

Before instruction		1 cycle after instruction	
A1	03B6E7D5h	A1	03B6E7D5h
A2	00000073h	A2	00000073h
A3	xxxxxxxxh	A3	000003B6h

## 4.150 EXTU

Extract and Zero-Extend a Bit Field

**Syntax**    **EXTU (.unit) src2, csta, cstb, dst**

or

**EXTU (.unit) src2, src1, dst**

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Sc5	<a href="#">Figure F-22</a>
	S2ext	<a href="#">Figure F-23</a>

**Opcode**    Constant form:

31	29	28	27		23	22		18
	<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3		1		5			5	
17	13	12		<i>csta</i>				
5			<i>cstb</i>					0
			5	8	7	6	5	4
				0	0	0	0	1
						0	1	0
							1	1

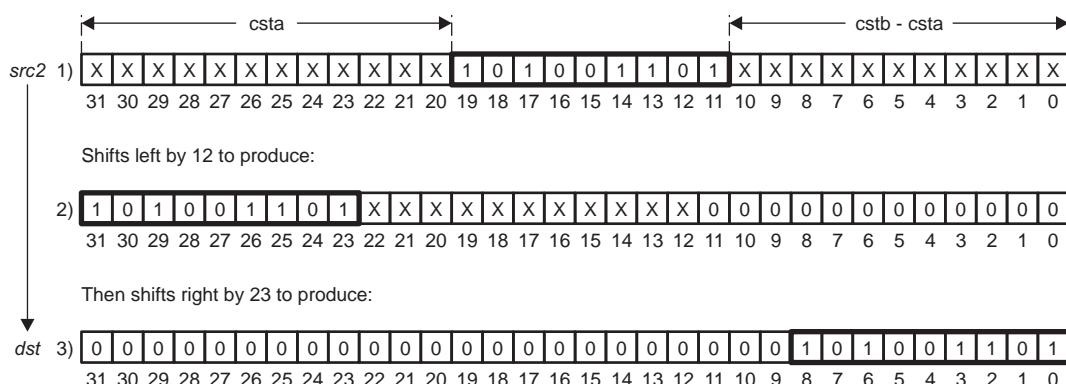
Opcode map field used...	For operand type...	Unit
<i>src2</i>	uint	.S1, .S2
<i>csta</i>	ucst5	
<i>cstb</i>	ucst5	
<i>dst</i>	uint	

**Opcode**    Register form:

31	29	28	27		23	22		18
	<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3		1		5			5	
17	13	12	11	10	9	8	7	6
5	<i>src1</i>	x	1	0	1	0	1	1
							1	0
							0	0
							1	0
								1
								1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.S1, .S2
<i>src1</i>	uint	
<i>dst</i>	uint	

**Description** The field in *src2*, specified by *csta* and *cstb*, is extracted and zero extended to 32 bits. The extract is performed by a shift left followed by an unsigned shift right. *csta* and *cstb* are the amounts to shift left and shift right, respectively. This can be thought of in terms of the LSB and MSB of the field to be extracted. Then  $csta = 31 - \text{MSB}$  of the field and  $cstb = csta + \text{LSB}$  of the field. The shift left and shift right amounts may also be specified as the ten LSBs of the *src1* register with *cstb* being bits 0-4 and *csta* bits 5-9. In the example below, *csta* is 12 and *cstb* is  $11 + 12 = 23$ . Only the ten LSBs are valid for the register version of the instruction. If any of the 22 MSBs are non-zero, the result is invalid.



**Execution** If the constant form is used:

```
if (cond)    src2 extu csta, cstb → dst  
else nop
```

If the register form is used:

```
if (cond)    src2 extu src19..5, src14..0 → dst
else non
```

Pipeline

Pipeline Stage	E1
Read	$src1, src2$
Written	$dst$
Unit in use	$.S$

### **Instruction Type**

### *Delay Slots* 0

**See Also**

### *Examples*      Example 1

EXTII S1 A1 10 19 A2

**Before instruction****1 cycle after instruction**A1      

07A43F2Ah
-----------

A1      

07A43F2Ah
-----------

A2      

xxxxxxxxh
-----------

A2      

0000121Fh
-----------

**Example 2**

EXTU .S1 A1,A2,A3

**Before instruction****1 cycle after instruction**A1      

03B6E7D5h
-----------

A1      

03B6E7D5h
-----------

A2      

00000156h
-----------

A2      

00000156h
-----------

A3      

xxxx xxxxh
------------

A3      

0000036Eh
-----------

## 4.151 FADDDP

Fast Double-Precision Floating Point add

**Syntax**    **FADDDP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	dst		src2	src1	x							opfield			1	1	0	s	p	

3                        5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1011000

**Opcode**    Opcode for .S Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	dst		src2	src1	x							opfield			1	0	0	0	s	p

3                        5                        5                        5                        6                        4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	001100

**Description**    *src2* is added to *src1*. The result is placed in *dst*. This instruction is the fast version of [ADDDP](#), with smaller Delay Slots and Functional Unit Latency.

## Special Cases

INPUTS		OUTPUT	Config. Reg.
src1	src2		
QNaN	QNaN	NaN_out	NAN1,NAN2
QNaN	SNaN	NaN_out	INVAL,NAN1,NAN2
QNaN	other <sup>1</sup>	NaN_out	NAN1
SNaN	QNaN	NaN_out	INVAL,NAN1,NAN2
SNaN	SNaN	NaN_out	INVAL,NAN1,NAN2
SNaN	other <sup>1</sup>	NaN_out	INVAL,NAN1
other <sup>1</sup>	QNaN	NaN_out	NAN2
other <sup>1</sup>	SNaN	NaN_out	INVAL,NAN2
+INF	-INF	NaN_out	INVAL
+INF	other <sup>2</sup>	+INF	INFO
-INF	+INF	NaN_out	INVAL
-INF	other <sup>3</sup>	-INF	INFO
other <sup>2</sup>	+INF	+INF	INFO
other <sup>3</sup>	-INF	+INF	INFO

1. Includes +/-INF

2. Includes +INF

3. Includes -INF

## Overflow Outputs:

1. Set the INEX bit and the OVER bit in configuration register.
2. Set the result as follows:

Overflow Output Rounding Mode				
result sign	nearest ev	zero	+inf	-inf
+	+inf	+LFPN <sup>1</sup>	+inf	+LFPN
-	-inf	-LFPN	-LFPN	-inf

1. LFPN is largest floating point number: man = '1..1' Exp = '1..10'.  
 Infinity outputs are flagged in the configuration register with INFO.

## Underflow Outputs:

Underflow Output Rounding Mode				
result sign	nearest ev	zero	+inf	-inf
+	+0	+0	+SFPN <sup>1</sup>	+0
-	-0	-0	-0	-SFPN

1. SFPN is smallest floating point number: man = '0..0' Exp = '0..01'.

## Zero Outputs:

- a. If an add of two equal numbers opposite in sign is performed, the resulting sign will be (+) unless the rounding is to -inf, in which case it will be (-).

- b. If an add of two zeros, (or denormals), with the same sign is performed, the sign of the output will be the sign of the inputs regardless of rounding mode.

$$(-0 + -0) \rightarrow -0$$

$$(+0 + +0) \rightarrow +0$$

#### **Denormal Inputs:**

Denormals will be flagged in the configuration register accordingly, and will cause the result to be inexact unless the other operand is a NaN or Infinity. The denormal input will be treated zero throughout the operation.

The INEX bit will be set if rounding was performed.

<b>Execution</b>	<pre>if(cond) src1 + src2 -&gt; dst else nop</pre>
<b>Instruction Type</b>	3-cycle
<b>Delay Slots</b>	2
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">ADDSP</a> , <a href="#">SUBDP</a> , <a href="#">SUBSP</a> , <a href="#">FADDSP</a> , <a href="#">FSUBDP</a> , <a href="#">FSUBSP</a>
<b>Example</b>	<pre>A3 == 0x40240000 A2 == 0x00000000 A5 == 0x40460000 A4 == 0x00000000 FADDDP .L A3,A2,A5,A4,A1A0 ; 10 + 44 = 54 A1 == 0x404b0000 A0 == 0x00000000  FADCR== 0x00000000 A3 == 0x402fc106 A2 == 0x24dd2f1b A5 == 0x401070a3 A4 == 0xd70a3d71 FADDDP .L A3,A2,A5,A4,A1A0 A1 == 0x4033fcac A0 == 0x083126ea  FADCR= 0x00000080  FADCR== 0x02000000 B3 == 0x402fc106 B2 == 0x24dd2f1b B5 == 0x401070a3 B4 == 0xd70a3d71 FADDDP .L B3,B2,B5,B4,B1B0 B1 == 0x4033fcac B0 == 0x083126e9  FADCR= 0x02800000  FADCR== 0x00000400 A3 == 0x402fc106 A2 == 0x24dd2f1b A5 == 0x401070a3 A4 == 0xd70a3d71 FADDDP .L A3,A2,A5,A4,A1A0 A1 == 0x4033fcac A0 == 0x083126ea  FADCR= 0x00000480  FADCR== 0x06000000 B3 == 0x402fc106 B2 == 0x24dd2f1b B5 == 0x401070a3 B4 == 0xd70a3d71</pre>

```
FADDDP .L B3,B2,B5,B4,B1B0
B1 == 0x4033fcac
B0 == 0x083126e9

FADCR= 0x06800000
```

## 4.152 FADDSP

Fast Single-Precision Floating Point Add

**Syntax**    **FADDSP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z	dst		src2	src1	x							opfield			1	1	0	s	p	

3                        5                        5                        5                        7                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	0111100

**Opcode**    Opcode for .S Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	5	4	3	2	1	0
creg	z	dst		src2	src1	x	1	1	1	0			opfield		1	1	0	s	p	

3                        5                        5                        5                        4                        3                        3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.S1 or .S2	100

**Description**    *src2* is added to *src1*. The result is placed in *dst*. This instruction is the fast version of ADDSP, with smaller Delay Slots.

## Special Cases

INPUTS		OUTPUT	Config. Reg.
src1	src2		
QNaN	QNaN	NaN_out	NAN1,NAN2
QNaN	SNaN	NaN_out	INVAL,NAN1,NAN2
QNaN	other <sup>1</sup>	NaN_out	NAN1
SNaN	QNaN	NaN_out	INVAL,NAN1,NAN2
SNaN	SNaN	NaN_out	INVAL,NAN1,NAN2
SNaN	other <sup>1</sup>	NaN_out	INVAL,NAN1
other <sup>1</sup>	QNaN	NaN_out	NAN2
other <sup>1</sup>	SNaN	NaN_out	INVAL,NAN2
+INF	-INF	NaN_out	INVAL
+INF	other <sup>2</sup>	+INF	INFO
-INF	+INF	NaN_out	INVAL
-INF	other <sup>3</sup>	-INF	INFO
other <sup>2</sup>	+INF	+INF	INFO
other <sup>3</sup>	-INF	+INF	INFO

1. Includes +/-INF

2. Includes +INF

3. Includes -INF

## Overflow Outputs

- Set the INEX bit and the OVER bit in configuration register.
- Set the result as follows:

Overflow Output Rounding Mode				
result sign	nearest ev	zero	+inf	-inf
+	+inf	+LFPN <sup>1</sup>	+inf	+LFPN
-	-inf	-LFPN	-LFPN	-inf

1. LFPN is largest floating point number: man = '1..1' Exp = '1..10'  
 Infinity outputs are flagged in the configuration register with INFO.

## Underflow Outputs:

Underflow Output Rounding Mode				
result sign	nearest ev	zero	+inf	-inf
+	+0	+0	+SFPN <sup>1</sup>	+0
-	-0	-0	-0	-SFPN

1. SFPN is smallest floating point number: man = '0..0' Exp = '0..01'

## Zero outputs:

- If an add of two equal numbers opposite in sign is performed, the resulting sign will be (+) unless the rounding is to -inf, in which case it will be (-).

- b. If an add of two zeros, (or denormals), with the same sign is performed, the sign of the output will be the sign of the inputs regardless of rounding mode.

$$(-0 + -0) \rightarrow -0$$

$$(+0 + +0) \rightarrow +0$$

#### **Denormal Inputs:**

Denormals will be flagged in the configuration register accordingly, and will cause the result to be inexact unless the other operand is a NaN or Infinity. The denormal input will be treated zero throughout the operation.

The INEX bit will be set if rounding was performed.

<b>Execution</b>	<pre>if(cond) src1 + src2 -&gt; dst else nop</pre>
<b>Instruction Type</b>	3-cycle
<b>Delay Slots</b>	2
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">ADDDP</a> , <a href="#">SUBDP</a> , <a href="#">SUBSP</a> , <a href="#">FADDP</a> , <a href="#">FSUBSP</a> , <a href="#">FSUBDP</a>
<b>Example</b>	<pre>A2 == 0x41200000 A4 == 0x42300000 FADDSP .L A2,A4,A0 ; 10 + 44 = 54 A0 == 0x42580000  FADCR== 0x00000000 A2 == 0x40784517 A4 == 0x42aeab2d FADDSP .L A2,A4,A0 A0 == 0x42b66d56  FADCR= 0x00000080  FADCR== 0x02000000 B2 == 0x40784517 B4 == 0x42aeab2d FADDSP .L B2,B4,B0 B0 == 0x42b66d55  FADCR= 0x02800000  FADCR== 0x00000400 A2 == 0x40784517 A4 == 0x42aeab2d FADDSP .L A2,A4,A0 A0 == 0x42b66d56  FADCR= 0x00000480  FADCR== 0x06000000 B2 == 0x40784517 B4 == 0x42aeab2d FADDSP .L B2,B4,B0 B0 == 0x42b66d55  FADCR= 0x06800000</pre>

## 4.153 FMPYDP

Fast Double-Precision Floating Point Multiply

**Syntax**    FMPYDP (.unit) *src1*, *src2*, *dst*

*unit* = .M1 or .M2

**Opcode**    Opcode for .M Unit, Compound Results

31	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	0	opfield		1	1	0	0	s	p	

3                                5                                5                                5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.M1 or .M2	01111

**Description**    *src1* is multiply by *src2* and the result is placed in *dst*. *src1*, *src2* and *dst* are all double precision floating point numbers stored in two consecutive registers

### Special Cases:

1. If one source is SNaN or QNaN, the result is a signed NaN\_out and the NANn bit is set. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the XOR to the input signs.
2. Signed infinity multiplied by signed infinity or a normalized number (other than signed zero) returns signed infinity. Signed infinity multiplied by signed zero (or denormal) returns a signed NaN\_out and sets the INVAL bit.
3. If one or both source are signed zero, the result is signed zero unless the other source is a NaN or signed infinity, in which case the result is signed NaN\_out.
4. If signed zero is multiplied by signed infinity, the result is signed NaN\_out and the INVAL bit is set.
5. A denormalized source is treated as signed zero and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, OR signed zero. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
6. If rounding is performed, the INEX bit is set.

**Execution**

```
if(cond) src1 x src2 -> dst
```

```
else nop
```

**Instruction Type**

4-cycle

**Delay Slots**

3

**Functional Unit Latency**

1

**See Also**

[MPYDP](#)

**Example**

```
FMCR== 0x00000000
A3 == 0x40213333
A2 == 0x33333333
A5 == 0xc0040000
A4 == 0x00000000
FMPYDP .M1 A3:A2,A5:A4,A1:A0
A1 == 0xc0358000
A0 == 0x00000000
```

```

FMCR= 0x00000080
;RMODE(0) (8.6)*[-2.5]=[-21.5] RMODE(0)+INEX(1)

FMCR== 0x00000200
A3 == 0x40213333
A2 == 0x33333333
A5 == 0xc0040000
A4 == 0x00000000
FMPYDP .M1 A3:A2,A5:A4,A1:A0
A1 == 0xc0357fff
A0 == 0xffffffff
FMCR= 0x00000280
;RMODE(1) (8.6)*[-2.5]=(-21.5) RMODE(1)+INEX(1)

FMCR== 0x04000000
B3 == 0x40213333
B2 == 0x33333333
B5 == 0xc0040000
B4 == 0x00000000
FMPYDP .M2 B3:B2,B5:B4,B1:B0
B1 == 0xc0357fff
B0 == 0xffffffff
FMCR= 0x04800000
;RMODE(2) (8.6)*[-2.5]=(-21.5) RMODE(2)+INEX(1)

FMCR== 0x06000000
B3 == 0x40213333
B2 == 0x33333333
B5 == 0xc0040000
B4 == 0x00000000
FMPYDP .M2 B3:B2,B5:B4,B1:B0
B1 == 0xc0358000
B0 == 0x00000000
FMCR= 0x06800000
;RMODE(3) (8.6)*[-2.5]=(-21.5) RMODE(3)+INEX(1)

```

## **4.154 FSUBDP**

## Fast Double-Precision Floating Point Subtract

**Syntax**      **FSUBDP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, or .S2

**Opcodes**      Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z	dst		src2		src1	x		opfield		1	1	0	s	p	
3		5		5		5			7							

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1011001

**Opcode**      Opcode for .S Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
creg	z	dst		src2		src1		x	1	1	opfield	1	1	0	0	s	p		
3		5		5		5					4								

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	1110

**Description** src2 is subtracted from src1. The result is placed in dst.

**Execution**      if(cond) src1 - src2 -> dst  
                      else nop

**Instruction Type** 3-cycle

*Delay Slots* 2

### ***Functional Unit Latency***

**See Also**    [ADDDP](#), [ADDSP](#), [SUBSP](#), [FADDDP](#), [FADDSP](#), [FSUBSP](#)

```

Example FADCR== 0x00000000
          A3 == 0x40213333
          A2 == 0x33333333
          A5 == 0xc0040000
          A4 == 0x00000000
          FSUBDP . A3,A2,A5,A4,A1A0
          A1 == 0x40263333
          A0 == 0x33333333
          8.6 - [-2.5] = (11.1) RMODE (0)+INEX (0)
          FADCR== 0x00000000

          FADCR== 0x00000000
          A3 == 0x41766489
          A2 == 0x78903832
          A5 == 0x40130877
          A4 == 0x983fffff
          FSUBDP . A3,A2,A5,A4,A1A0
          A1 == 0x41766489
          A0 == 0x2c6e59d1
          23480471.535... - (4.75826871) = 234
          FADCR= 0x00000080

```

## 4.155 FSUBSP

Fast Single-Precision Floating Point Subtract

**Syntax** **FSUBSP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode** Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opfield	1	1	0	s	p	

3                                       5                                   5                                   7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	0111101

**Opcode** Opcode for .S Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	5	4	3	2	1	0
creg	z		dst		src2		src1	x	1	1	1	0	opfield	1	1	0	s	p		

3                                       5                                   5                                   3

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.S1 or .S2	110

**Description** src2 is subtracted from src1. The result is placed in dst.

**Execution** if(cond) src1 - src2 → dst

else nop

**Instruction Type** 3-cycle

**Delay Slots** 2

**Functional Unit Latency** 1

**See Also** ADDDP, ADDSP, SUBDP, FADDP, FADDSP, FSUBSP

**Example**

```
FADCR== 0x00000000
A1 == 0x4109999a
A2 == 0xc0200000
FSUBSP . A1,A2,A0
A0 == 0x4131999a

8.6 - (-2.5) = 11.1
FADCR= 0x00000000
A2 == 0x41200000
A4 == 0xc2300000
FSUBSP . A2,A4,A0 ;
A0 == 0x42580000
2.5 - (-11) = 13.5

FADCR== 0x00000000
A2 == 0x40784517
A4 == 0xc2aeab2d
FSUBSP . A2,A4,A0
```

```

A0 == 0x42b66d56
3.8792169094 - (-87.33432769) = 91.2135467529
FADCR= 0x00000080

FADCR== 0x02000000
B2 == 0x40784517
B4 == 0xc2aab2d
FSUBSP . B2,B4,B0
B0 == 0x42b66d55
3.8792169094 - (-87.33432769) = 91.2135391235

FADCR= 0x02800000

FADCR== 0x00000400
A2 == 0x40784517
A4 == 0xc2aab2d
FSUBSP . A2,A4,A0
A0 == 0x42b66d56
3.8792169094 - (-87.33432769) = 91.2135467529
FADCR= 0x00000480

FADCR== 0x06000000
B2 == 0x40784517
B4 == 0xc2aab2d
FSUBSP . B2,B4,B0
B0 == 0x42b66d55
3.8792169094 - (-87.33432769) = 91.2135391235
FADCR= 0x06800000

```

## 4.156 GMPY

Galois Field Multiply

**Syntax**    **GMPY (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	30	29	28	27			23	22											18
0	0	0	1		dst				src2										
					5					5									
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
		src1	x	0	1	1	1	1	1	1	1	0	0	s	p				
		5		1										1	1				

Opcode map field used...	For operand type...	Unit
src1	uint	.M1, .M2
src2	uint	
dst	uint	

**Description**    Performs a Galois field multiply, where *src1* is 32 bits and *src2* is limited to 9 bits. This utilizes the existing hardware and produces a 32-bit result. This multiply connects all levels of the gmpy4 together and only extends out by 8 bits, the resulting data is XORed down by the 32-bit polynomial.

The polynomial used comes from either the GPLYA or GPLYB control register depending on which side (A or B) the instruction executes. If the A-side M1 unit is used, the polynomial comes from GPLYA; if the B-side M2 unit, the polynomial comes from GPLYB.

This instruction executes unconditionally.

```
uword gmpy(uword src1,uword src2,uword polynomial)
{
    // the multiply is always between GF(2^9) and GF(2^32)
    // so no size information is needed

    uint pp;
    uint mask, tpp;
    uint I;

    pp = 0;
    mask = 0x00000100; // multiply by computing
                        // partial products.
    for ( I=0; i<8; I++ ){
        if ( src2 & mask ) pp ^= src1;
        mask >>= 1;
        tpp = pp << 1;
        if (pp & 0x80000000) pp = polynomial ^ tpp;
        else pp = tpp;
    }
    if ( src2 & 0x1 ) pp ^= src1;
```

```
        return (pp) ; // leave it asserted left.
    }
```

**Execution**

```
if (unit = M1)
    GMPY_poly = GPLYA
    lsb9(src2) gmpy src1 → dst

else if (unit = M2)
    GMPY_poly = GPLYB
    lsb9(src2) gmpy src1 → dst
```

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [GMPY4](#), [XORMPY](#), [XOR](#)

**Example** GMPY .M1 A0,A1,A2 GPLYA = 87654321

**Before instruction**

A0	12345678h
----	-----------

A1	00000126h
----	-----------

**4 cycles after instruction**

A2	C721A0EFh
----	-----------

4.157 GMPY4

## Galois Field Multiply, Packed 8-Bit

**Syntax**    **GMPY4** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

## *Opcode*

Opcode map field used...	For operand type...	Unit
<i>src1</i>	u4	.M1, .M2
<i>src2</i>	xu4	
<i>dst</i>	u4	

**Description** Performs the Galois field multiply on four values in *src1* with four parallel values in *src2*. The four products are packed into *dst*. The values in both *src1* and *src2* are treated as unsigned, 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned, 8-bit value from *src1* is Galois field multiplied (gmpy) with the unsigned, 8-bit value from *src2*. The product of *src1* byte 0 and *src2* byte 0 is written to byte0 of *dst*. The product of *src1* byte 1 and *src2* byte 1 is written to byte1 of *dst*. The product of *src1* byte 2 and *src2* byte 2 is written to byte2 of *dst*. The product of *src1* byte 3 and *src2* byte 3 is written to the most-significant byte in *dst*.

31	24 23	16 15	8 7	0
ua_3	ua_2	ua_1	ua_0	$\leftarrow srcl$

GMPY4

ub\_3      ub\_2      ub\_1      ub\_0       $\leftarrow src2$

|| || || ||

31	0		
ua_3 gmpy ub_3	ua_2 gmpy ub_2	ua_1 gmpy ub_1	ua_0 gmpy ub_0

The size and polynomial are controlled by the Galois field polynomial generator function register (GFPGFR). All registers in the control register file can be written using the **MVC** instruction (see [MVC](#)).

The default field generator polynomial is 1Dh, and the default size is 7. This setting is used for many communications standards.

Note that the **GMPY4** instruction is commutative, so:

```
GMPY4 .M1 A10,A12,A13
```

is equivalent to:

```
GMPY4 .M1 A12,A10,A13
```

**Execution**

```
if (cond) {
    (ubyte0(src1) gmpy ubyte0(src2)) → ubyte0(dst);
    (ubyte1(src1) gmpy ubyte1(src2)) → ubyte1(dst);
    (ubyte2(src1) gmpy ubyte2(src2)) → ubyte2(dst);
    (ubyte3(src1) gmpy ubyte3(src2)) → ubyte3(dst)
}
else nop
```

#### Pipeline

Pipeline Stage	E1	E2	E3	E4
Read		src1,src2		
Written				dst
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [GMPY](#), [MVC](#), [XOR](#)

**Examples** [Example 1](#)

```
GMPY4 .M1 A5,A6,A7; polynomial = 0x11d
```

Before instruction			4 cycles after instruction		
A5	45 23 00 01h	69 35 0 1 unsigned	A5	45 23 00 01h	
A6	57 34 00 01h	87 52 0 1 unsigned	A6	57 34 00 01h	
A7	xxxx xxxxh		A7	72 92 00 01h	114 146 0 1 unsigned

**Example 2**

```
GMPY4 .M1 A5,A6,A7; field size is 256
```

Before instruction			4 cycles after instruction		
A5	FF FE 02 1Fh	255 254 2 31 unsigned	A5	FF FE 02 1Fh	

A6 

FF FE 02 01h
--------------

 255 254 2 1  
unsigned

A6 

FF FE 02 01h
--------------

  
A7 

xxxx xxxxh
------------

 226 227 4 31  
unsigned

## 4.158 IDLE

Multicycle NOP With No Termination Until Interrupt

**Syntax**    **IDLE**

unit = none

**Opcode**

31	18	17	16												
Reserved															
14															
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1
1															1

**Description**    Performs an infinite multicycle **NOP** that terminates upon servicing an interrupt, or a branch occurs due to an **IDLE** instruction being in the delay slots of a branch.

The **IDLE** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **ADDKPC**, **BNOP**, and the multicycle **NOP**.

**Instruction Type**    NOP

**Delay Slots**    0

## 4.159 INTDP

Convert Signed Integer to Double-Precision Floating-Point Value

**Syntax**    **INTDP (.unit) *src2*, *dst***

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22						18	17	16	
				<i>dst</i>								0	0
3				5						5			
15	14	13	12	11	10	9	8	7	6	5	4	3	2
0	0	0	x	0	1	1	1	0	0	1	1	1	0
				1								1	1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xsint	.L1, .L2
<i>dst</i>	dp	

**Description**    The signed integer value in *src2* is converted to a double-precision value and placed in *dst*.

You cannot set configuration bits with this instruction.

**Execution**

```
if (cond)    dp(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read		<i>src2</i>			
Written				<i>dst_l</i>	<i>dst_h</i>
Unit in use		.L			

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type**    INTDP

**Delay Slots**    4

**Functional Unit Latency**    1

**See Also**    [DPINT](#), [INTDPU](#), [INTSP](#), [INTSPU](#)

**Example**    INTDP .L1X B4,A1:A0

**Before instruction**

B4	1965 1127h	426,053,927
----	------------	-------------

**4 cycles after instruction**

B4	1965 1127h
----	------------

A1:A0	xxxx xxxxh	xxxx xxxxh	A1:A0	41B9 6511h	2700 0000h
4.2605393 E08					

## 4.160 INTDPU

Convert Unsigned Integer to Double-Precision Floating-Point Value

**Syntax**    **INTDPU (.unit) src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22						18	17	16			
				<i>dst</i>						<i>src2</i>		0	0		
3				5						5					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	x	0	1	1	1	0	1	1	1	1	0	s	p
				1									1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.L1, .L2
<i>dst</i>	dp	

**Description**    The unsigned integer value in *src2* is converted to a double-precision value and placed in *dst*.

You cannot set configuration bits with this instruction.

**Execution**

```
if (cond)    dp(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read		src2			
Written				dst_l	dst_h
Unit in use		.L			

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type**    INTDP

**Delay Slots**    4

**Functional Unit Latency**    1

**See Also**    [DPINT](#), [INTDP](#), [INTSP](#), [INTSPU](#)

**Example**    INTDPU .L1 A4,A1:A0

**Before instruction**

A4	FFFF FFDEh	4,294,967,262
----	------------	---------------

**4 cycles after instruction**

A4	FFFF FFDEh
----	------------

A1:A0	xxxx xxxxh	xxxx xxxxh	A1:A0	41EF FFFFh	FBC0 0000h
4.2949673 E09					

4.161 INTSP

## Convert Signed Integer to Single-Precision Floating Point

**Syntax**    **INTSP** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**      Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z	dst		src2		src1	x		opfield		1	1	0	s	p	
3		5		5		5			7							

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src2,dst	xop, dst	.L1 or .L2	1001010

**Opcode**      Opcode for .S Unit, 1 src

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src2,dst	xop, dst	.S1 or .S2	00101

**Description** Converts the signed integer in src2 to a single precision floating point value. The INEX bit is set if the mantissa was rounded.

**Execution**      if(cond) sp(src2) -> dst  
                  else nop

**Instruction Type** 4-cycle

*Delay Slots* 3

## ***Functional Unit Latency***

---

**See Also** DPINT, DPTRUNC, INTDP, INTDPU, INTSP, SPINT, SPTRUNC

**Example**

```

FADCR== 0x00000000
A2    == 0x19651127
INTSP .L A2,A0
A0    == 0x4dcb2889

FADCR= 0x00000080

FADCR== 0x02000000
B2    == 0x19651127
INTSP .L B2,B0
B0    == 0x4dcb2889

FADCR= 0x02800000

FADCR== 0x00000400
A2    == 0x19651127
INTSP .L A2,A0
A0    == 0x4dcb288a

FADCR= 0x00000480

FADCR== 0x06000000
B2    == 0x19651127
INTSP .L B2,B0
B0    == 0x4dcb2889

FADCR= 0x06800000

FADCR== 0x00000000
A2    == 0xfffffffde
INTSP .L A2,A0
A0    == 0xc2080000

FADCR== 0x00000000

FADCR== 0x00000000
A2    == 0xfffffffff
INTSP .L A2,A0
A0    == 0xbff80000

FADCR== 0x00000080

```

## 4.162 INTSPU

Convert Unsigned Integer to Single-Precision Floating Point

**Syntax**    **INTSPU** (.unit) *src1, src2, dst*

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opfield		1	1	0	s	p

3                         5                         5                         5                         7

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dst	.L1 or .L2	1001001

**Opcode**    Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	1	1	1	1	0	0	1	0	0	0	0	s	p

3                         5                         5                         5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dst	.S1 or .S2	00001

**Description**    Converts the unsigned integer in src2 to a single precision floating point value. The INEX bit is set if the mantissa was rounded.

**Execution**

```
if(cond) sp(uint(src2)) -> dst
else nop
```

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [DPINT](#), [DPTRUNC](#), [INTDP](#), [INTDPU](#), [INTSP](#), [SPINT](#), [SPTRUNC](#)

**Example**

```

FADCR== 0x00000000
A2    == 0x19651127
INTSPU .L A2,A0
A0    == 0x4dcb2889

FADCR= 0x00000080

FADCR== 0x02000000
B2    == 0x19651127
INTSPU .L B2,B0
B0    == 0x4dcb2889

FADCR= 0x02800000

FADCR== 0x00000400
A2    == 0x19651127
INTSPU .L A2,A0
A0    == 0x4dcb288a

FADCR= 0x00000480

FADCR== 0x06000000
B2    == 0x19651127
INTSPU .L B2,B0
B0    == 0x4dcb2889

FADCR= 0x06800000

FADCR== 0x00000000
A2    == 0xfffffffde
INTSPU .L A2,A0
A0    == 0x4f800000

FADCR= 0x00000080

FADCR== 0x00000000
A2    == 0xfffffffff
INTSPU .L A2,A0
A0    == 0x4f800000

FADCR== 0x00000080

```

## 4.163 LAND

Logical AND

**Syntax** **LAND (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opfield	1	1	0	s	p	

3                    5                    5                    7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1111000

**Description** The LAND instruction performs logical AND between two source registers. If the two operands are not zero, the result is one. Otherwise the result is zero. The result is stored in the destination register

**Execution**

```
If (cond) {
    if ( src1 != 0 AND src2 != 0 ){
        dst = 1
    }
    else dst = 0
}
```

**Instruction Type** Single cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [AND, LOR](#)

**Example**

```
A0 == 0x12340000
A0 == 0x00005678
LAND .L A0,A0,A15
A15 == 0x00000001

A0 == 0x00000000
A0 == 0x00005678
LAND .L A0,A0,A15
A15 == 0x00000000
```

## 4.164 LANDN

Logical AND, One Operand Negated

**Syntax** **LANDN (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode** Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opfield	1	1	0	s	p	

3                    5                    5                    7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1111001

**Description** The LANDN instruction performs logical AND between a source register and the NOT of a second register.

```
Execution      If (cond) {
                  if ( src1 != 0 AND (NOT src2) !=0 ){
                      dst = 1
                  }
                  else dst = 0
              }
```

**Instruction Type** Single cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [AND](#), [LOR](#)

**Example**

```
A0 == 0x12340000
A0 == 0x00005678
LANDN .L A0,A0,A15
A15 == 0x00000000

A0 == 0x12340000
A0 == 0x00000000
LANDN .L A0,A0,A15
A15 == 0x00000001
```

## 4.165 LDB(U)

Load Byte From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax	Register Offset	Unsigned Constant Offset
<code>LDB (.unit) *+baseR[offsetR], dst</code>		<code>LDB (.unit) *+baseR[ucst5], dst</code>
<code>LDBU (.unit) *+baseR[offsetR], dst</code>		<code>LDBU (.unit) *+baseR[ucst5], dst</code>
unit = .D1 or .D2		

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>

Opcode											
31	29	28	27	23 22		18					
	<i>creg</i>	<i>z</i>		<i>dst</i>		<i>baseR</i>					
3		1		5			5				
17	13	12		9	8	7	6	4	3	2	1 0
	<i>offsetR/ucst5</i>		<i>mode</i>	0	<i>y</i>		<i>op</i>	0	1	<i>s</i>	<i>p</i>
5			4		1		3		1	1	

### Description

Loads a byte from memory to a general-purpose register (*dst*). [Table 4-6](#) summarizes the data types supported by loads. [Table 3-12](#) on page 3-30 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

**Table 4-6 Data Types Supported by LDB(U) Instruction**

Mnemonic	<i>op</i> Field	Load Data Type	Size	Left Shift of Offset
LDB	0 1 0	Load byte	8	0 bits
LDBU	0 0 1	Load byte unsigned	8	0 bits

*offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

*offsetR/ucst5* is scaled by a left-shift of 0 bits. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed in memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

For **LDB(U)**, the values are loaded into the 8 LSBs of *dst*. For **LDB**, the upper 24 bits of *dst* values are sign-extended; for **LDBU**, the upper 24 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit should be cleared to 0.

Increments and decrements default to 1 and offsets default to 0 when no bracketed register or constant is specified. Loads that do no modification to the *baseR* can use the syntax \**R*. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

**Execution**

```
if (cond)    mem → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR, offsetR</i>				
Written	<i>baseR</i>				<i>dst</i>
Unit in use	.D				

**Instruction Type** Load

**Delay Slots** 4 for loaded value

0 for address modification from pre/post increment/decrement

For more information on delay slots for a load, see Chapter 5 “[Pipeline](#)” on page 5-1.

**See Also** [LDH\(U\), LDW](#)

**Examples** [Example 1](#)

LDB .D1 \*-A5[4],A7

	<b>Before instruction</b>	<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>	
A5	00000204h	A5	00000204h	A5	00000204h
A7	19511970h	A7	19511970h	A7	FFFFFE1h
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 200h	E1h	mem 200h	E1h	mem 200h	E1h

**Example 2**

LDB .D1 \*++A4[5],A8

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	00000400h	A4	0000 4005h	A4	0000 4005h
A8	0000 0000h	A8	0000 0000h	A8	0000 0067h
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 4000h	0112 2334h	mem 4000h	0112 2334h	mem 4000h	0112 2334h
mem 4004h	4556 6778h	mem 4004h	4556 6778h	mem 4004h	4556 6778h

### Example 3

LDB .D1 \*A4++ [5] ,A8

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	00000400h	A4	0000 4005h	A4	0000 4005h
A8	0000 0000h	A8	0000 0000h	A8	0000 0034h
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 4000h	0112 2334h	mem 4000h	0112 2334h	mem 4000h	0112 2334h
mem 4004h	4556 6778h	mem 4004h	4556 6778h	mem 4004h	4556 6778h

### Example 4

LDB .D1 \*++A4 [A12] ,A8

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	00000400h	A4	0000 4006h	A4	0000 4006h
A8	0000 0000h	A8	0000 0000h	A8	0000 0056h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 4000h	0112 2334h	mem 4000h	0112 2334h	mem 4000h	0112 2334h
mem 4004h	4556 6778h	mem 4004h	4556 6778h	mem 4004h	4556 6778h

## 4.166 LDB(U)

Load Byte From Memory With a 15-Bit Unsigned Constant Offset

**Syntax**    **LDB** (.unit) \*+B14/B15[*ucst15*], *dst*

or

**LDBU** (.unit) \*+B14/B15[*ucst15*], *dst*

unit = .D2

**Opcode**

31	29	28	27	23
<i>creg</i>	<i>z</i>			<i>dst</i>
3	1			5
22			8    7    6	4    3    2    1    0
	<i>ucst15</i>		<i>y</i>	<i>op</i>
	15	1	3	1    1

**Description**    Loads a byte from memory to a general-purpose register (*dst*). [Table 4-7](#) summarizes the data types supported by loads. The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction operates only on the .D2 unit.

The offset, *ucst15*, is scaled by a left shift of 0 bits. After scaling, *ucst15* is added to *baseR*. Subtraction is not supported. The result of the calculation is the address sent to memory. The addressing arithmetic is always performed in linear mode.

For **LDB(U)**, the values are loaded into the 8 LSBs of *dst*. For **LDB**, the upper 24 bits of *dst* values are sign-extended; for **LDBU**, the upper 24 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

**Table 4-7      Data Types Supported by LDB(U) Instruction (15-Bit Offset)**

Mnemonic	op Field			Load Data Type	Size	Left Shift of Offset
LDB	0	1	0	Load byte	8	0 bits
LDBU	0	0	1	Load byte unsigned	8	0 bits

**Execution**    if (cond) mem → *dst*

```
else nop
```



**Note**—This instruction executes only on the B side (.D2).

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	B14/B15				
Written					dst
Unit in use	.D2				

**Instruction Type** Load

**Delay Slots** 4

**See Also** [LDH\(U\), LDW](#)

**Example** LDB .D2 \*+B14 [36] ,B1

**Before instruction**                            **1 cycle after instruction**

B1	xxxxxxxx	B1	xxxxxxxx
B14	00000100h	B14	00000100h
mem 124-127h	4E7AFF12h	mem 124-127h	4E7AFF12h
mem 124h	12h	mem 124h	12h

**5 cycles after instruction**

B1	0000 0012h
B14	00000100h
mem 124-127h	4E7AFF12h
mem 124h	12h

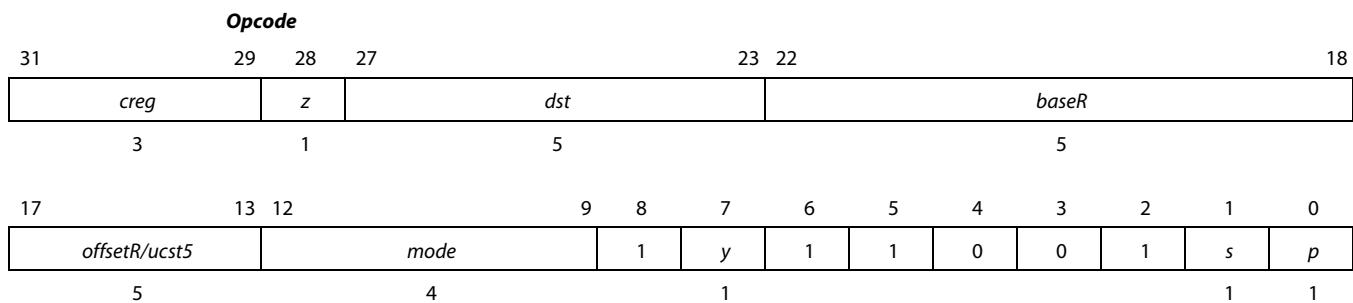
## 4.167 LDDW

Load Doubleword From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

<b>Syntax</b>	<b>Register Offset</b>	<b>Unsigned Constant Offset</b>
	<b>LDDW (.unit) *+baseR[offsetR], dst</b>	<b>LDDW (.unit) *+baseR[ucst5], dst</b>
	unit = .D1 or .D2	

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Doff4DW	<a href="#">Figure C-9</a>
	DindDW	<a href="#">Figure C-11</a>
	DincDW	<a href="#">Figure C-13</a>
	DdecDW	<a href="#">Figure C-15</a>
	Dpp	<a href="#">Figure C-21</a>



**Description**

Loads a 64-bit quantity from memory into a register pair *dst\_o:dst\_e*. [Table 3-12](#) on page 3-30 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

Both *offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and the register file used: *y* = 0 selects the .D1 unit and the *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file. The *s* bit determines the register file into which the *dst* is loaded: *s* = 0 indicates that *dst* is in the A register file, and *s* = 1 indicates that *dst* is in the B register file. The *r* bit has a value of 1 for the **LDDW** instruction. The *dst* field must always be an even value because the **LDDW** instruction loads register pairs. Therefore, bit 23 is always zero.

The *offsetR/ucst5* is scaled by a left-shift of 3 to correctly represent doublewords. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the shifted value of *baseR* before the addition or subtraction is the address to be accessed in memory.

Increments and decrements default to 1 and offsets default to 0 when no bracketed register, bracketed constant, or constant enclosed in parentheses is specified. Square brackets, [ ], indicate that *ucst5* is left shifted by 3. Parentheses, ( ), indicate that *ucst5* is not left shifted. In other words, parentheses indicate a byte offset rather than a doubleword offset. You must type either brackets or parenthesis around the specified offset if you use the optional offset parameter.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

The destination register pair must consist of a consecutive even and odd register pair from the same register file. The instruction can be used to load a double-precision floating-point value (64 bits), a pair of single-precision floating-point words (32 bits), or a pair of 32-bit integers. The 32 least-significant bits are loaded into the even-numbered register and the 32 most-significant bits (containing the sign bit and exponent) are loaded into the next register (which is always odd-numbered register). The register pair syntax places the odd register first, followed by a colon, then the even register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

All 64 bits of the double-precision floating point value are stored in big- or little-endian byte order, depending on the mode selected. When the LDDW instruction is used to load two 32-bit single-precision floating-point values or two 32-bit integer values, the order is dependent on the endian mode used. In little-endian mode, the first 32-bit word in memory is loaded into the even register. In big-endian mode, the first 32-bit word in memory is loaded into the odd register. Regardless of the endian mode, the doubleword address must be on a doubleword boundary (the three LSBs are zero).

**Execution**

```
if (cond)    mem → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR, offsetR</i>				
Written	<i>baseR</i>				<i>dst</i>
Unit in use	.D				

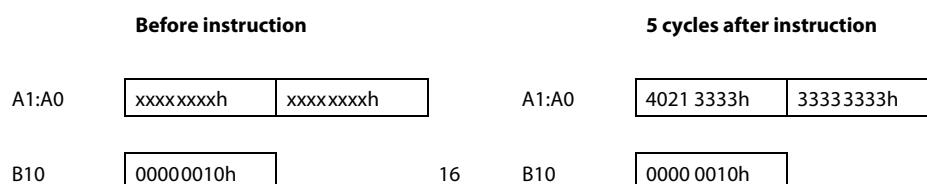
**Instruction Type** Load

**Delay Slots** 4

**Functional Unit Latency** 1

**Examples** [Example 1](#)

LDDW .D2 \*+B10[1],A1:A0



mem 18h	33333333h	40213333h	8.6	mem 18h	33333333h	40213333h
Little-endian mode						

### Example 2

LDDW .D1 \*++A10[1],A1:A0

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A1:A0	xxxxxxxxh	xxxxxxxxh	A1:A0	xxxxxxxxh	xxxxxxxxh
A10	00000010h		16	A10	0000 0018h
mem 18h	40213333h	33333333h	8.6	mem 18h	40213333h

### 5 cycles after instruction

A1:A0	4021 3333h	33333333h
A10	0000 0018h	24
mem 18h	40213333h	33333333h

Big-endian mode

### Example 3

LDDW .D1 \*A4++[5],A9:A8

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A9:A8	xxxxxxxxh	xxxxxxxxh	A9:A8	xxxxxxxxh	xxxxxxxxh
A4	0000 40B0h		A4	0000 40B0h	
mem 40B0h	0112 2334h	4556 6778h	mem 40B0h	0112 2334h	4556 6778h

### 5 cycles after instruction

A9:A8	4556 6778h	0112 2334h
A4	0000 40B0h	

mem 40B0h	0112 2334h	4556 6778h
Little-endian mode		

### Example 4

LDDW .D1 \*++A4 [A12], A9:A8

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A9:A8	xxxxxxxxh	xxxxxxxxh	
A4	0000 40B0h		0000 40E0h
A12	0000 0006h		0000 0006h
mem 40E0h	0112 2334h	4556 6778h	8
		mem 40E0h	0112 2334h    4556 6778h

### 5 cycles after instruction

A9:A8	4556 6778h	0112 2334h
A4	0000 40E0h	
A12	0000 0006h	
mem 40E0h	0112 2334h	4556 6778h

Little-endian mode

### Example 5

LDDW .D1 \*++A4 (16), A9:A8

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A9:A8	xxxxxxxxh	xxxxxxxxh	
A4	0000 40B0h		0000 40C0h
mem 40C0h	4556 6778h	899A ABBCh	mem 40C0h    4556 6778h    899A ABBCh

### 5 cycles after instruction

A9:A8	899A ABBCh	4556 6778h
-------	------------	------------

A4

0000 40C0h

mem 40C0h	4556 6778h	899A ABBCh
Little-endian mode		

## 4.168 LDH(U)

Load Halfword From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax	Register Offset	Unsigned Constant Offset
	<code>LDH (.unit) *+baseR[offsetR], dst</code>	<code>LDH (.unit) *+baseR[ucst5], dst</code>
	<code>LDHU (.unit) *+baseR[offsetR], dst</code>	<code>LDHU (.unit) *+baseR[ucst5], dst</code>
unit = .D1 or .D2		

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>

<b>Opcode</b>											
31	29	28	27	23 22				18			
creg	z	dst				baseR					
3	1	5				5					
17	13	12	9	8	7	6	4	3	2	1	0
offsetR/ucst5	mode			0	y	op	0	1	1	s	p
5	4			1		3			1		1

**Description**

Loads a halfword from memory to a general-purpose register (*dst*). [Table 4-8](#) summarizes the data types supported by halfword loads. [Table 3-12](#) on page 3-30 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

**Table 4-8 Data Types Supported by LDH(U) Instruction**

Mnemonic	op Field	Load Data Type	Size	Left Shift of Offset
LDH	1 0 0	Load halfword	16	1 bit
LDHU	0 0 0	Load halfword unsigned	16	1 bit

*offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

*offsetR/ucst5* is scaled by a left-shift of 1 bit. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed in memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

For **LDH(U)**, the values are loaded into the 16 LSBs of *dst*. For **LDH**, the upper 16 bits of *dst* are sign-extended; for **LDHU**, the upper 16 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit should be cleared to 0.

Increments and decrements default to 1 and offsets default to 0 when no bracketed register or constant is specified. Loads that do no modification to the *baseR* can use the syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

**Execution**

```
if (cond) mem → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR, offsetR</i>				
Written	<i>baseR</i>				<i>dst</i>
Unit in use	.D				

**Instruction Type** Load

**Delay Slots** 4 for loaded value

0 for address modification from pre/post increment/decrement

For more information on delay slots for a load, see Chapter 5 “[Pipeline](#)” on page 5-1.

**See Also** [LDB\(U\), LDW](#)

**Example** LDH .D1 \*++A4[A1],A8

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A1	00000002h	A1	00000002h	A1	00000002h
A4	00000020h	A4	00000024h	A4	00000024h
A8	110351FFh	A8	110351FFh	A8	FFFFA21Fh
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 24h	A21Fh	mem 24h	A21Fh	mem 24h	A21Fh

## 4.169 LDH(U)

Load Halfword From Memory With a 15-Bit Unsigned Constant Offset

**Syntax**    **LDH (.unit) \*+B14/B15[ucst15], dst**

or

**LDHU (.unit) \*+B14/B15[ucst15], dst**

unit = .D2

**Opcode**

31	29	28	27	23
<i>creg</i>	<i>z</i>			<i>dst</i>
3	1			5
22			8      7      6	4      3      2      1      0
	<i>ucst15</i>		<i>y</i>	<i>op</i>
	15		1	3

**Description**    Loads a halfword from memory to a general-purpose register (*dst*). [Table 4-9](#) summarizes the data types supported by loads. The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction operates only on the .D2 unit.

The offset, *ucst15*, is scaled by a left shift of 1 bit. After scaling, *ucst15* is added to *baseR*. Subtraction is not supported. The result of the calculation is the address sent to memory. The addressing arithmetic is always performed in linear mode.

For **LDH(U)**, the values are loaded into the 16 LSBs of *dst*. For **LDH**, the upper 16 bits of *dst* are sign-extended; for **LDHU**, the upper 16 bits of *dst* are zero-filled. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

**Table 4-9      Data Types Supported by LDH(U) Instruction (15-Bit Offset)**

Mnemonic	op Field			Load Data Type	Size	Left Shift of Offset
LDH	1	0	0	Load halfword	16	1 bit
LDHU	0	0	0	Load halfword unsigned	16	1 bit

**Execution**    if (cond) mem → *dst*

```
else nop
```

 **Note**—This instruction executes only on the B side (.D2).

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	B14/B15				
Written					dst
Unit in use	.D2				

**Instruction Type** Load

**Delay Slots** 4

**See Also** [LDB\(U\), LDW](#)

## 4.170 LDNDW

Load Nonaligned Doubleword From Memory With Constant or Register Offset

<b>Syntax</b>	<b>Register Offset</b>	<b>Unsigned Constant Offset</b>
	<b>LDNDW (.unit) *+baseR[offsetR], dst</b>	<b>LDNDW (.unit) *+baseR[ucst5], dst</b>
	unit = .D1 or .D2	

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Doff4DW	<a href="#">Figure C-9</a>
	DindDW	<a href="#">Figure C-11</a>
	DincDW	<a href="#">Figure C-13</a>
	DdecDW	<a href="#">Figure C-16</a>

**Opcode**

31	29	28	27	24	23	22	18
creg	z		dst	sc			baseR
3	1		4	1			5
17	13	12		9	8	7	6
offsetR/ucst5		mode		1	y	0	1
5		4		1		0	1
						1	1
						s	p

Opcode map field used...	For operand type...	Unit
baseR offsetR dst	uint uint ullong	.D1, .D2
baseR offsetR dst	uint ucst5 ullong	.D1, .D2

**Description**

Loads a 64-bit quantity from memory into a register pair, *dst\_o:dst\_e*. [Table 3-12](#) on page 3-30 describes the addressing generator options. The LDNDW instruction may read a 64-bit value from any byte boundary. Thus alignment to a 64-bit boundary is not required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

Both *offsetR* and *baseR* must be in the same register file, and on the same side, as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

The LDNDW instruction supports both scaled offsets and nonscaled offsets. The *sc* field is used to indicate whether the *offsetR/ucst5* is scaled or not. If *sc* is 1 (scaled), the *offsetR/ucst5* is shifted left 3 bits before adding or subtracting from the *baseR*. If *sc* is 0 (nonscaled), the *offsetR/ucst5* is not shifted before adding or subtracting from the *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

The *dst* field of the instruction selects a register pair, a consecutive even-numbered and odd-numbered register pair from the same register file. The instruction can be used to load a pair of 32-bit integers. The 32 least-significant bits are loaded into the even-numbered register and the 32 most-significant bits are loaded into the next register (that is always an odd-numbered register).

The *dst* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit has a value of 1 for the LDNDW instruction.



**Note**—No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel as long as it is not performing a memory access.

#### Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword loads.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled offset.

For example, **LDNDW (.unit) \*+baseR (14)**, *dst* represents an offset of 14 bytes, and the assembler writes out the instruction with *offsetC* = 14 and *sc* = 0.

**LDNDW (.unit) \*+baseR [16]**, *dst* represents an offset of 16 doublewords, or 128 bytes, and the assembler writes out the instruction with *offsetC* = 16 and *sc* = 1.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

#### Execution

```
if (cond)    mem → dst
else nop
```

#### Pipeline

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR,offsetR</i>				
Written	<i>baseR</i>				<i>dst</i>
Unit in use	.D				

#### Instruction Type

Load

#### Delay Slots

4 for loaded value

0 for address modification from pre/post increment/decrement

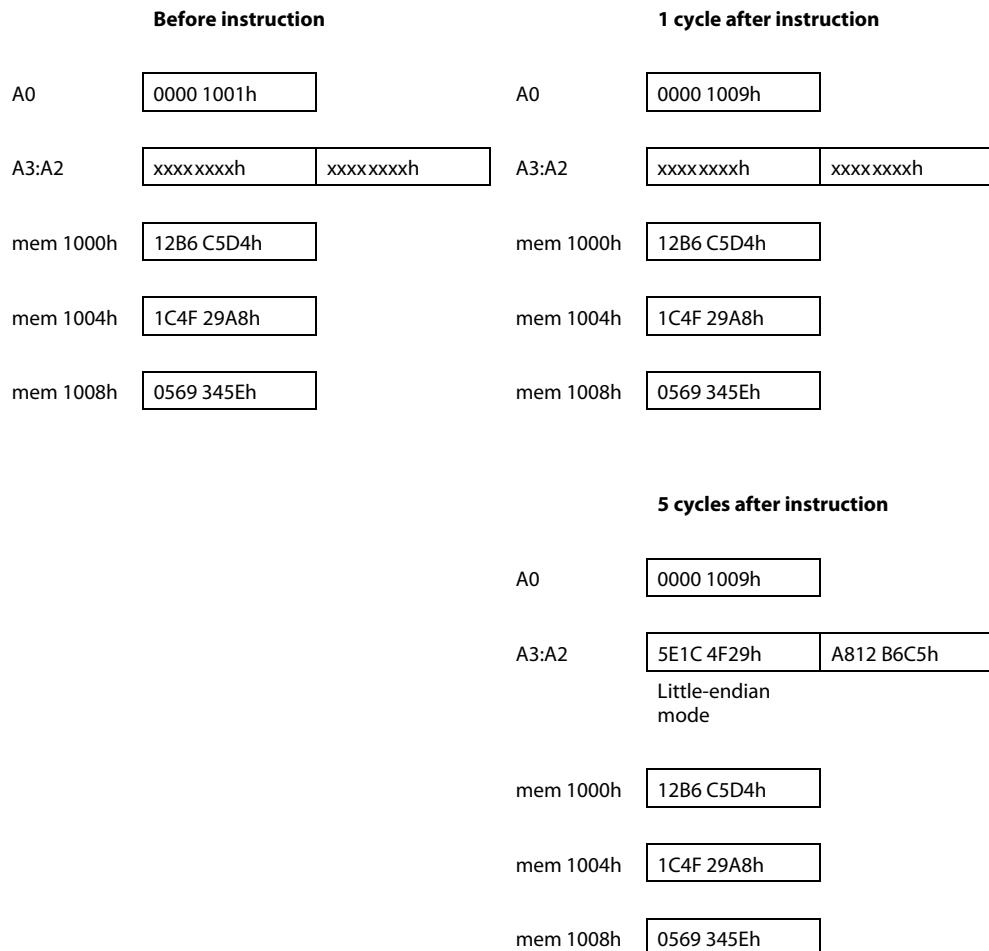
#### See Also

[LDNW, STNDW, STNW](#)

#### Examples

[Example 1](#)

LDNDW .D1 \*A0++, A3:A2



<b>Byte Memory Address</b>	<b>100C</b>	<b>100B</b>	<b>100A</b>	<b>1009</b>	<b>1008</b>	<b>1007</b>	<b>1006</b>	<b>1005</b>	<b>1004</b>	<b>1003</b>	<b>1002</b>	<b>1001</b>	<b>1000</b>
Data Value	11	05	69	34	5E	1C	4F	29	A8	12	B6	C5	D4

### Example 2

LDNDW .D1 \*A0++, A3:A2

**Before instruction****1 cycle after instruction**

A0	0000 1003h		A0	0000 100Bh	
A3:A2	xxxxxxxxh		A3:A2	xxxxxxxxh	
mem 1000h	12B6 C5D4h		mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h		mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh		mem 1008h	0569 345Eh	

**5 cycles after instruction**

A0	0000 100Bh	
A3:A2	6934 5E1Ch	4F29 A812h
mem 1000h	12B6 C5D4h	
mem 1004h	1C4F 29A8h	
mem 1008h	0569 345Eh	

Little-endian mode

Byte Memory Address	100C	100B	100A	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value	11	05	69	34	5E	1C	4F	29	A8	12	B6	C5	D4

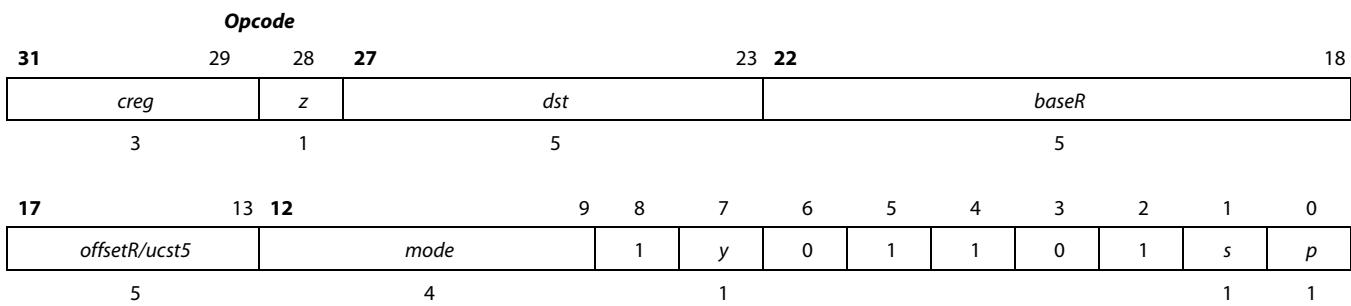
## 4.171 LDNW

Load Nonaligned Word From Memory With Constant or Register Offset

<b>Syntax</b>	<b>Register Offset</b>	<b>Unsigned Constant Offset</b>
	<b>LDNW (.unit) *+baseR[offsetR], dst</b>	<b>LDNW (.unit) *+baseR[ucst5], dst</b>
	unit = .D1 or .D2	

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>



Opcode map field used...	For operand type...	Unit
baseR	uint	.D1, .D2
offset	uint	
dst	int	
baseR	uint	.D1, .D2
offset	ucst5	
dst	int	

**Description** Loads a 32-bit quantity from memory into a 32-bit register, *dst*. [Table 3-12](#) on page 3-30 describes the addressing generator options. The LDNW instruction may read a 32-bit value from any byte boundary. Thus alignment to a 32-bit boundary is not required. The memory address is formed from a base address register (*baseR*), and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

Both *offsetR* and *baseR* must be in the same register file, and on the same side, as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

The *offsetR/ucst5* is scaled by a left shift of 2 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

The *dst* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit has a value of 1 for the **LDNW** instruction.



**Note**—No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel, as long as it is not doing a memory access.

#### Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 2 for word loads.

Parentheses, ( ), can be used to tell the assembler that the offset is a nonscaled, constant offset. The assembler right shifts the constant by 2 bits for word loads before using it for the *ucst5* field. After scaling by the **LDNW** instruction, this results in the same constant offset as the assembler source if the least-significant two bits are zeros.

For example, **LDNW** (.unit) \*+*baseR* (12), *dst* represents an offset of 12 bytes (3 words), and the assembler writes out the instruction with *ucst5* = 3.

**LDNW** (.unit) \*+*baseR* [12], *dst* represents an offset of 12 words, or 48 bytes, and the assembler writes out the instruction with *ucst5* = 12.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

**Execution**

```
if (cond)    mem → dst
else nop
```

#### Pipeline

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR,offsetR</i>				
Written	<i>baseR</i>				<i>dst</i>
Unit in use	.D				

#### Instruction Type

Load

#### Delay Slots

4 for loaded value

0 for address modification from pre/post increment/decrement

#### See Also

[LDNDW](#), [STNDW](#), [STNW](#)

#### Examples

**Example 1**

```
LDNW .D1 *A0++, A2
```

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A0	0000 1001h	A0	0000 1005h	A0	0000 1005h
A2	xxxx xxxxh	A2	xxxx xxxxh	A2	A812 B6C5h
mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h
mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h

<b>Byte Memory Address</b>	<b>1007</b>	<b>1006</b>	<b>1005</b>	<b>1004</b>	<b>1003</b>	<b>1002</b>	<b>1001</b>	<b>1000</b>
Data Value	1C	4F	29	A8	12	B6	C5	D4

### Example 2

LDNW .D1 \*A0++, A2

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A0	0000 1003h	A0	0000 1007h	A0	0000 1007h
A2	xxxx xxxxh	A2	xxxx xxxxh	A2	4F29 A812h
mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h	mem 1000h	12B6 C5D4h
mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h	mem 1004h	1C4F 29A8h

<b>Byte Memory Address</b>	<b>1007</b>	<b>1006</b>	<b>1005</b>	<b>1004</b>	<b>1003</b>	<b>1002</b>	<b>1001</b>	<b>1000</b>
Data Value	1C	4F	29	A8	12	B6	C5	D4

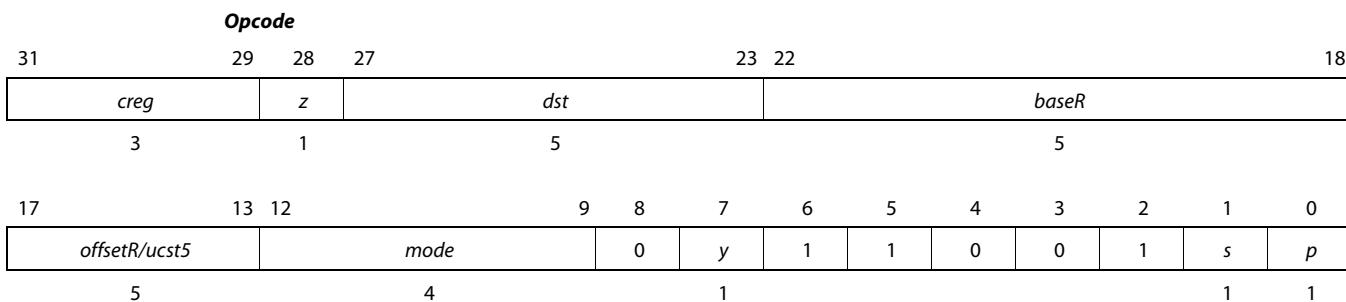
## 4.172 LDW

Load Word From Memory With a 5-Bit Unsigned Constant Offset or Register Offset

Syntax	<b>Register Offset</b>	<b>Unsigned Constant Offset</b>
	<b>LDW (.unit) *+baseR[offsetR], dst</b>	<b>LDW (.unit) *+baseR[ucst5], dst</b>
	unit = .D1 or .D2	

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>



**Description**

Loads a word from memory to a general-purpose register (*dst*). [Table 3-12](#) on page 3-30 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

*offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

*offsetR/ucst5* is scaled by a left-shift of 2 bits. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed in memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see [“Addressing Mode Register \(AMR\)”](#) on page 2-12).

For LDW, the entire 32 bits fills *dst*. *dst* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file. The *r* bit should be cleared to 0.

Increments and decrements default to 1 and offsets default to 0 when no bracketed register or constant is specified. Loads that do no modification to the *baseR* can use the syntax \**R*. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 2. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **LDW (.unit) \*+baseR (12)**, *dst* represents an offset of 12 bytes; whereas, **LDW (.unit) \*+baseR [12]**, *dst* represents an offset of 12 words, or 48 bytes. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Word addresses must be aligned on word (two LSBs are 0) boundaries.

**Execution**

```
if (cond)    mem → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR, offsetR</i>				
Written	<i>baseR</i>				<i>dst</i>
Unit in use	.D				

**Instruction Type** Load

**Delay Slots** 4 for loaded value

0 for address modification from pre/post increment/decrement

For more information on delay slots for a load, see Chapter 5 “[Pipeline](#)” on page 5-1.

**See Also** [LDB\(U\), LDH\(U\)](#)

**Examples** [Example 1](#)

LDW .D1 \*A10,B1

	Before instruction	1 cycle after instruction	5 cycles after instruction
B1	0000 0000h	B1	0000 0000h
A10	00000100h	A10	00000100h
mem 100h	21F3 1996h	mem 100h	21F3 1996h

**Example 2**

LDW .D1 \*A4++ [1],A6

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	00000100h	A4	00000104h	A4	00000104h
A6	12344321h	A6	12344321h	A6	0798F25Ah
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 100h	0798 F25Ah	mem 100h	0798 F25Ah	mem 100h	0798 F25Ah
mem 104h	1970 19F3h	mem 104h	1970 19F3h	mem 104h	1970 19F3h

### Example 3

LDW .D1 \*++A4 [1], A6

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	00000100h	A4	00000104h	A4	00000104h
A6	1234 5678h	A6	12345678h	A6	02176991h
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 104h	0217 6991h	mem 104h	0217 6991h	mem 104h	0217 6991h

### Example 4

LDW .D1 \*++A4 [A12], A8

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	0000 40B0h	A4	0000 40C8h	A4	0000 40C8h
A8	0000 0000h	A8	0000 0000h	A8	DCCB BAA8h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 40C8h	DCCB BAA8h	mem 40C8h	DCCB BAA8h	mem 40C8h	DCCB BAA8h

---

**Example 5**

LDW .D1 \*++A4 (8) ,A8

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>5 cycles after instruction</b>
A4	0000 40B0h	A4	0000 40B8h	A4	0000 40B8h
A8	0000 0000h	A8	0000 0000h	A8	9AAB BCCDh
AMR	00000000h	AMR	00000000h	AMR	00000000h
mem 40B8h	9AAB BCCDh	mem 40B8h	9AAB BCCDh	mem 40B8h	9AAB BCCDh

## 4.173 LDW

Load Word From Memory With a 15-Bit Unsigned Constant Offset

**Syntax** **LDW (.unit) \*+B14/B15[*ucst15*], *dst***

*unit* = .D2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Dstk	<a href="#">Figure C-16</a>
	Dpp	<a href="#">Figure C-21</a>

<b>Opcodē</b>									
31	29	28	27	23					
<i>creg</i>	<i>z</i>	<i>dst</i>							
3	1			5					
22				8      7      6      5      4      3      2      1      0					
		<i>ucst15</i>		<i>y</i>	1	1	0	1	1
		15		1					1      1

**Description** Load a word from memory to a general-purpose register (*dst*). The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction operates only on the .D2 unit.

The offset, *ucst15*, is scaled by a left shift of 2 bits. After scaling, *ucst15* is added to *baseR*. Subtraction is not supported. The result of the calculation is the address sent to memory. The addressing arithmetic is always performed in linear mode.

For LDW, the entire 32 bits fills *dst*. *dst* can be in either register file. The *s* bit determines which file *dst* will be loaded into: *s* = 0 indicates *dst* will be loaded in the A register file and *s* = 1 indicates *dst* will be loaded in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 2. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **LDW (.unit) \*+B14/B15(60)**, *dst* represents an offset of 60 bytes; whereas, **LDW (.unit) \*+B14/B15[60]**, *dst* represents an offset of 60 words, or 240 bytes. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Word addresses must be aligned on word (two LSBs are 0) boundaries.

**Execution** if (cond) mem → *dst*

---

else nop



---

**Note**—This instruction executes only on the B side (.D2).

---

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
Read	B14/B15				
Written					dst
Unit in use	.D2				

**Instruction Type** Load

**Delay Slots** 4

**See Also** [LDB\(U\)](#), [LDH\(U\)](#)

## 4.174 LMBD

Leftmost Bit Detection

**Syntax**    **LMBD** (.unit) *src1, src2, dst*

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22		18
<i>creg</i>	<i>z</i>			<i>dst</i>		<i>src2</i>
3	1			5		5
17	13	12	11			
<i>src1/cst5</i>	<i>x</i>			<i>op</i>	1	1
5	1			7		1
						1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	uint	.L1, .L2	1101011
<i>src2</i>	xuint		
<i>dst</i>	uint		
<i>src1</i>	cst5	.L1, .L2	1101010
<i>src2</i>	xuint		
<i>dst</i>	uint		

**Description**

The LSB of the *src1* operand determines whether to search for a leftmost 1 or 0 in *src2*. The number of bits to the left of the first 1 or 0 when searching for a 1 or 0, respectively, is placed in *dst*.

The following diagram illustrates the operation of **LMBD** for several cases.

When searching for 0 in *src2*, **LMBD** returns 0:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

When searching for 1 in *src2*, **LMBD** returns 4:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

When searching for 0 in *src2*, LMBD returns 32:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

```
Execution    if (cond)    {
                  if (src10 == 0), lmb0(src2) → dst if (src10 == 1), lmb1(src2) → dst
                  }
                else nop
```

#### Pipeline

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**Example** LMBD .L1 A1,A2,A3

#### Before instruction

A1	00000001h
A2	009E3A81h
A3	xxxxxxxxh

#### 1 cycle after instruction

A1	00000001h
A2	009E3A81h
A3	00000008h

## 4.175 LOR

Logical OR

**Syntax**    **LOR (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opfield	1	1	0	s	p	

3                                        5                                        5                                        7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1111101

**Description**    The LOR instruction performs logical OR between two source registers. If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0.

```
Execution    If (cond) {
    if ( src1 != 0 OR src2 !=0 ){
        dst = 1
    }
    else dst = 0
}
```

**Instruction Type**    Single cycle

**Delay Slots**    0

**Functional Unit Latency**    1

**See Also**    [OR](#), [LAND](#)

**Example**

```
A0 == 0x12340000
A0 == 0x000005678
LOR .L A0,A0,A15
A15 == 0x00000001

A0 == 0x00000000
A0 == 0x00000000
LOR .L A0,A0,A15
A15 == 0x00000000
```

## 4.176 MAX2

Maximum, Signed, Packed 16-Bit

**Syntax**    **MAX2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

31	29	28	27							23	22							18	
<i>creg</i>	<i>z</i>	<i>dst</i>						<i>src2</i>											
3		1		5						5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
<i>src1</i>	x	1	0	0	0	0	1	0	1	1	0	0	s	p	1	1			
5		1													1	1			

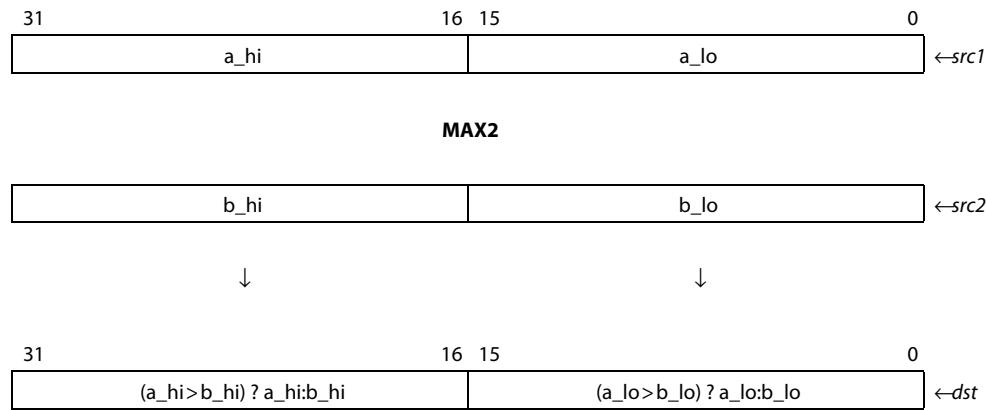
Opcode map field used...	For operand type...	Unit
<i>src1</i>	s2	.L1, .L2
<i>src2</i>	xs2	
<i>dst</i>	s2	

**Opcode**    .S unit

31	29	28	27							23	22									18
<i>creg</i>	<i>z</i>	<i>dst</i>						<i>src2</i>												18
3		1		5						5										
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
<i>src1</i>	x	1	1	1	1	0	1	1	1	0	0	0	s	p	1	1	1	1		
5		1													1	1	1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	s2	.S1, .S2
<i>src2</i>	xs2	
<i>dst</i>	s2	

**Description**    Performs a maximum operation on signed, packed 16-bit values. For each pair of signed 16-bit values in *src1* and *src2*, **MAX2** places the larger value in the corresponding position in *dst*.



**Execution**

```

if (cond) {
    if (lsb16(src1) >= lsb16(src2)), lsb16(src1) → lsb16(dst)
    else lsb16(src2) → lsb16(dst);
    if (msb16(src1) >= msb16(src2)), msb16(src1) → msb16(dst)
    else msb16(src2) → msb16(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [MAXU4](#), [MIN2](#), [MINU4](#)

**Examples** [Example 1](#)

MAX2 .L1 A2, A8, A9

**Before instruction**

A2	3789 F23Ah
A8	04B8 4975h
A9	xxxx xxxxh

**1 cycle after instruction**

A2	3789 F23Ah	14217 -3526
A8	04B8 4975h	1208 18805
A9	3789 4975h	14217 18805

**Example 2**

MAX2 .L2X A2, B8, B12

**Before instruction****1 cycle after instruction**A2 0124 2451hA2 0124 2451h 292 9297B8 01A6 A051hB8 01A6 A051h 422 -24495B12 xxxx xxxxhB12 01A6 2451h 422 9297**Example 3**

MAX2 .S1 A2, A8, A9

**Before instruction****1 cycle after instruction**A2 3789 F23AhA2 3789 F23Ah 14217 -3526A8 04B8 4975hA8 04B8 4975h 1208 18805A9 xxxx xxxxhA9 3789 4975h 14217 18805**Example 4**

MAX2 .S2X A2, B8, B12

**Before instruction****1 cycle after instruction**A2 0124 2451hA2 0124 2451h 292 9297B8 01A6 A051hB8 01A6 A051h 422 -24495B12 xxxx xxxxhB12 01A6 2451h 422 9297

## 4.177 MAXU4

Maximum, Unsigned, Packed 8-Bit

**Syntax**    **MAXU4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22								18					
creg	z	dst								src2							
3	1	5								5							
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
src1	x	1	0	0	0	0	0	1	1	1	1	0	s	p			
5	1												1	1			

Opcode map field used...	For operand type...	Unit
src1	u4	.L1, .L2
src2	xu4	
dst	u4	

**Description**

Performs a maximum operation on unsigned, packed 8-bit values. For each pair of unsigned 8-bit values in *src1* and *src2*, **MAXU4** places the larger value in the corresponding position in *dst*.

31	24 23	16 15	8 7	0
ua_3	ua_2	ua_1	ua_0	←src1

**MAXU4**

ub_3	ub_2	ub_1	ub_0	←src2
------	------	------	------	-------

↓                          ↓                          ↓                          ↓

31	24 23	16 15	8 7	0
ua_3 > ub_3 ? ua_3:ub_3	ua_2 > ub_2 ? ua_2:ub_2	ua_1 > ub_1 ? ua_1:ub_1	ua_0 > ub_0 ? ua_0:ub_0	←dst

**Execution**    if (cond) {

```

if (ubyte0(src1) >= ubyte0(src2)), ubyte0(src1) → ubyte0(dst)
else ubyte0(src2) → ubyte0(dst);
if (ubyte1(src1) >= ubyte1(src2)), ubyte1(src1) → ubyte1(dst)
else ubyte1(src2) → ubyte1(dst);
if (ubyte2(src1) >= ubyte2(src2)), ubyte2(src1) → ubyte2(dst)
else ubyte2(src2) → ubyte2(dst);
if (ubyte3(src1) >= ubyte3(src2)), ubyte3(src1) → ubyte3(dst)
else ubyte3(src2) → ubyte3(dst)
```

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [MAX2, MIN2, MINU4](#)

**Examples** [Example 1](#)

```
MAXU4 .L1 A2, A8, A9
```

**Before instruction**

A2	<span style="border: 1px solid black; padding: 2px;">37 89 F2 3Ah</span>
----	--

**1 cycle after instruction**

A2	<span style="border: 1px solid black; padding: 2px;">37 89 F2 3Ah</span>	55 137 242 58 unsigned
----	--	---------------------------

A8	<span style="border: 1px solid black; padding: 2px;">04 B8 49 75h</span>
----	--

A8	<span style="border: 1px solid black; padding: 2px;">04 B8 49 75h</span>	4 184 73 117 unsigned
----	--	--------------------------

A9	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>
----	--

A9	<span style="border: 1px solid black; padding: 2px;">37 B8 F2 75h</span>	55 184 242 117 unsigned
----	--	----------------------------

**Example 2**

```
MAXU4 .L2X A2, B8, B12
```

**Before instruction**

A2	<span style="border: 1px solid black; padding: 2px;">01 24 24 B9h</span>
----	--

**1 cycle after instruction**

A2	<span style="border: 1px solid black; padding: 2px;">01 24 24 B9h</span>	1 36 36 185 unsigned
----	--	-------------------------

B8	<span style="border: 1px solid black; padding: 2px;">01 A6 A0 51h</span>
----	--

B8	<span style="border: 1px solid black; padding: 2px;">01 A6 A0 51h</span>	1 166 160 81 unsigned
----	--	--------------------------

B12	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>
-----	--

B12	<span style="border: 1px solid black; padding: 2px;">01 A6 A0 B9h</span>	1 166 160 185 unsigned
-----	--	---------------------------

## 4.178 MFENCE

Memory Fence

**Syntax**    **MFENCE**

**Opcode**    Opcode for CTLEMU

31	30	29	28	27	18	17	16	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1		parameter	0		opcode	0	0	0	0	0	0	0	0	0	0	0	s	p

10                          4

Opcode map field used...	For operand type...	Unit	Opfield
		1 or 2	0100

**Description**    The MFENCE instruction stalls the instruction fetch pipeline until the memory system busy flag goes low.

The instruction will ALWAYS wait at least 5 clock cycles before checking the busy flag in order to account for pipeline delays on where we are sampling the mem\_to\_cpu\_busy signal.

E.g., this code will wait until the STW data has completed

STW A0, \*A1

MFENCE ; This will wait until the STW write above has landed in it's final destination

During the course of executing an MFENCE operation, any enabled interrupts will still be serviced. IRP or NRP will be set to the PC of the execute packet containing the MFENCE instruction (NOTE: This is different than how IDLEs and Multi-cycle NOPs are handled)

The MFENCE operation has the following restrictions:

1. MFENCE should not be executed in parallel with any other instructions. It should be the only instruction in the execute packet. This restriction is not enforced by hardware, and correct operation is not guaranteed if this is done.
2. MFENCE cannot be included in the prolog or body of an SPLOOP. This is not enforced by hardware.
3. MFENCE cannot be included in the epilog of a reloadable SPLOOP.
4. MFENCE should not be included in the epilog of an SPLOOP earlier than 5 cycles before the SPLOOP goes completely IDLE (this is because MFENCE only guarantees 5 cycles of "NOP")

**Execution**

```
for (i=0; i<5; i++)
{
    nop ;
}
while ( mem_to_cpu_busy() )
```

```
{  
    nop ;  
}
```

**Instruction Type** Flow Control

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also**

**Example**

## 4.179 MIN2

Minimum, Signed, Packed 16-Bit

**Syntax**    **MIN2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3		1			5						5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

	<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>src1</i>		s2	.L1, .L2
<i>src2</i>		xs2	
<i>dst</i>		s2	

**Opcode**    .S unit

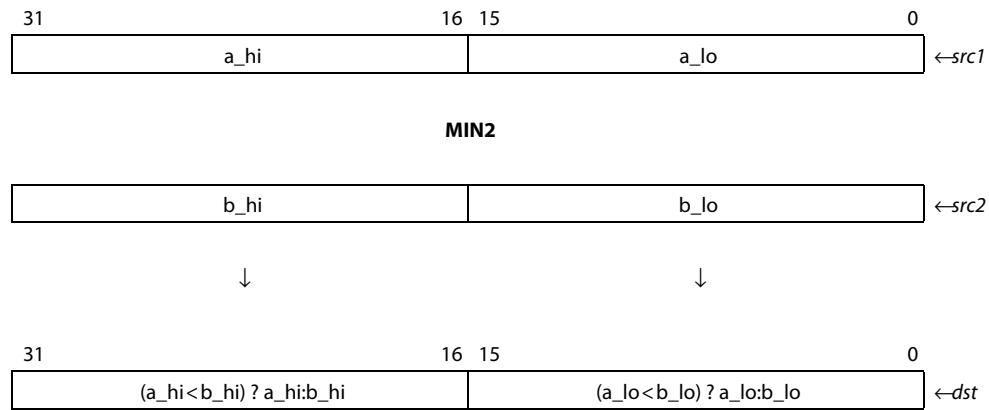
31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3		1			5						5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

	<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>src1</i>		s2	.S1, .S2
<i>src2</i>		xs2	
<i>dst</i>		s2	

**Description**

Performs a minimum operation on signed, packed 16-bit values. For each pair of signed 16-bit values in *src1* and *src2*, MIN2 instruction places the smaller value in the corresponding position in *dst*.



**Execution**

```

if (cond) {
    if (lsb16(src1) <= lsb16(src2)), lsb16(src1) → lsb16(dst)
    else lsb16(src2) → lsb16(dst);
    if (msb16(src1) <= msb16(src2)), msb16(src1) → msb16(dst)
    else msb16(src2) → msb16(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [MAX2](#), [MAXU4](#), [MINU4](#)

**Examples** [Example 1](#)

MIN2 .L1 A2, A8, A9

Before instruction		1 cycle after instruction	
A2	3789 F23Ah	A2	3789 F23Ah 14217 -3526
A8	04B8 4975h	A8	04B8 4975h 1208 18805
A9	xxxx xxxxh	A9	04B8 F23Ah 1208 -3526

**Example 2**

MIN2 .L2X A2, B8, B12

**Before instruction**A2 0124 8003hB8 0A37 8001hB12 xxxx xxxxh**1 cycle after instruction**A2 0124 8003h 292 -32765B8 0A37 8001h 2615 -32767B12 0124 8001h 292 -32767**Example 3**

MIN2 .S1 A2, A8, A9

**Before instruction**A2 3789 F23AhA8 04B8 4975hA9 xxxx xxxxh**1 cycle after instruction**A2 3789 F23Ah 14217 -3526A8 04B8 4975h 1208 18805A9 04B8 F23Ah 1208 -3526**Example 4**

MIN2 .S2X A2, B8, B12

**Before instruction**A2 0124 8003hB8 0A37 8001hB12 xxxx xxxxh**1 cycle after instruction**A2 0124 8003h 292 -32765B8 0A37 8001h 2615 -32767B12 0124 8001h 292 -32767

## 4.180 MINU4

Minimum, Unsigned, Packed 8-Bit

**Syntax**    **MINU4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22								src2					
creg	z	dst								src2							
3	1	5								5							
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
src1	x	1	0	0	1	0	0	0	0	1	1	0	s	p			
5	1												1	1			

Opcode map field used...	For operand type...	Unit
src1	u4	.L1, .L2
src2	xu4	
dst	u4	

**Description**

Performs a minimum operation on unsigned, packed 8-bit values. For each pair of unsigned 8-bit values in *src1* and *src2*, **MINU4** places the smaller value in the corresponding position in *dst*.

31	24 23	16 15	8 7	0
ua_3	ua_2	ua_1	ua_0	←src1

**MINU4**

ub_3	ub_2	ub_1	ub_0	←src2
------	------	------	------	-------

↓                          ↓                          ↓                          ↓

31	24 23	16 15	8 7	0
ua_3 < ub_3 ? ua_3:ub_3	ua_2 < ub_2 ? ua_2:ub_2	ua_1 < ub_1 ? ua_1:ub_1	ua_0 < ub_0 ? ua_0:ub_0	←dst

```
Execution    if (cond)    {
    if (ubyte0(src1) <= ubyte0(src2)), ubyte0(src1) → ubyte0(dst)
    else ubyte0(src2) → ubyte0(dst);
    if (ubyte1(src1) <= ubyte1(src2)), ubyte1(src1) → ubyte1(dst)
    else ubyte1(src2) → ubyte1(dst);
    if (ubyte2(src1) <= ubyte2(src2)), ubyte2(src1) → ubyte2(dst)
    else ubyte2(src2) → ubyte2(dst);
    if (ubyte3(src1) <= ubyte3(src2)), ubyte3(src1) → ubyte3(dst)
    else ubyte3(src2) → ubyte3(dst)
```

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [MAX2](#), [MAXU4](#), [MIN2](#)

**Examples** [Example 1](#)

MINU4 .L1 A2, A8, A9

**Before instruction**

A2	37 89 F2 3Ah
----	--------------

**1 cycle after instruction**

A2	37 89 F2 3Ah	55 137 242 58 unsigned
----	--------------	---------------------------

A8	04 B8 49 75h	4 184 73 117 unsigned
----	--------------	--------------------------

A9	xxxx xxxxh
----	------------

A9	04 89 49 3Ah	4 137 73 58 unsigned
----	--------------	-------------------------

**Example 2**

MINU4 .L2 B2, B8, B12

**Before instruction**

B2	01 24 24 B9h
----	--------------

**1 cycle after instruction**

B2	01 24 24 B9h	1 36 36 185 unsigned
----	--------------	-------------------------

B8	01 A6 A0 51h	1 166 160 81 unsigned
----	--------------	--------------------------

B12	xxxx xxxxh
-----	------------

B12	01 24 24 51h	1 36 36 81 unsigned
-----	--------------	------------------------

## 4.181 MPY

Multiply Signed 16 LSB × Signed 16 LSB

**Syntax** **MPY (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

Opcode									
31	29	28	27	23	22	18			
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>			
3	1		5			5			
17	13	12	11	7	6	5	4	3	2
<i>src1</i>	<i>x</i>		<i>op</i>	0	0	0	0	0	<i>s</i>
5	1		5						1
									1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	slsb16	.M1, .M2	11001
<i>src2</i>	xslsb16		
<i>dst</i>	sint		
<i>src1</i>	scst5	.M1, .M2	11000
<i>src2</i>	xslsb16		
<i>dst</i>	sint		

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

**Execution**

```
if (cond) lsb16(src1) × lsb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYU](#), [MPYSU](#), [MPYUS](#), [SMPY](#)

**Examples** [Example 1](#)

MPY .M1 A1,A2,A3

**Before instruction****2 cycles after instruction**

A1	<span style="border: 1px solid black; padding: 2px;">00000123h</span>	291 <sup>1</sup>	A1	<span style="border: 1px solid black; padding: 2px;">00000123h</span>
A2	<span style="border: 1px solid black; padding: 2px;">01E0FA81h</span>	-1407 <sup>1</sup>	A2	<span style="border: 1px solid black; padding: 2px;">01E0FA81h</span>
A3	<span style="border: 1px solid black; padding: 2px;">xxxxxxxxh</span>		A3	<span style="border: 1px solid black; padding: 2px;">FFF9C0A3h</span>

1. Signed 16-LSB integer

**Example 2**

MPY .M1 13,A1,A2

**Before instruction****2 cycles after instruction**

A1	<span style="border: 1px solid black; padding: 2px;">3497FFF3h</span>	-13 <sup>1</sup>	A1	<span style="border: 1px solid black; padding: 2px;">3497FFF3h</span>
A2	<span style="border: 1px solid black; padding: 2px;">xxxxxxxxh</span>		A2	<span style="border: 1px solid black; padding: 2px;">FFFFFF57h</span>

1. Signed 16-LSB integer

## 4.182 MPY2

Multiply Signed by Signed, 16 LSB  $\times$  16 LSB and 16 MSB  $\times$  16 MSB

**Syntax** **MPY2 (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Opcode**

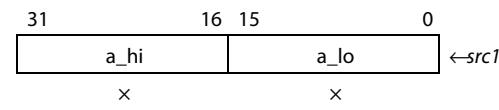
31	29	28	27						23	22						18
creg	z	dst					src2									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	0	0	0	0	0	0	1	1	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
src1	s2	.M1, .M2
src2	xs2	
dst	ullong	

**Description** Performs two 16-bit by 16-bit multiplications between two pairs of signed, packed 16-bit values. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The two 32-bit results are written into a 64-bit register pair.

The product of the lower halfwords of *src1* and *src2* is written to the even destination register, *dst\_e*. The product of the upper halfwords of *src1* and *src2* is written to the odd destination register, *dst\_o*.

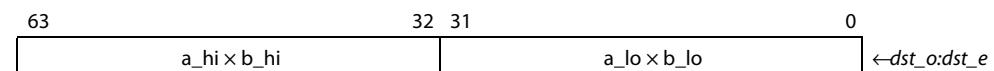
This instruction helps reduce the number of instructions required to perform two 16-bit by 16-bit multiplies on both the lower and upper halves of two registers.



**MPY2**



=



The following code:

```
MPY .M1 A0, A1, A2
MPYH .M1 A0, A1, A3
```

may be replaced by:

```
MPY2 .M1 A0, A1, A3:A2
```

**Execution**

```
if (cond) {
    lsb16(src1) × lsb16(src2) → dst_e;
    msb16(src1) × msb16(src2) → dst_o
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		<i>.M</i>		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYSU4](#), [MPY2IR](#), [SMPY](#)

**Examples** **Example 1**

```
MPY2 .M1 A5,A6, A9:A8
```

Before instruction			4 cycles after instruction		
A5	6A32 1193h	27186 4499	A5	6A32 1193h	
A6	B174 6CA4h	-20108 27812	A6	B174 6CA4h	
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	DF6A B0A8h	0775 462Ch
				-546,656,088	125,126,188

**Example 2**

```
MPY2 .M2 B2, B5, B9:B8
```

Before instruction			4 cycles after instruction		
B2	1234 3497h	4660 13463	B2	1234 3497h	
B5	21FF 50A7h	8703 20647	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	026A D5CCh	1091 7E81h
				40,555,980	277,970,561

## 4.183 MPY2IR

Multiply Two 16-Bit  $\times$  32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result

**Syntax** **MPY2IR** (.unit) *src1*, *src2*, *dst\_o:dst\_e*

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	30	29	28	27			23	22											18
0	0	0	1		<i>dst</i>				<i>src2</i>										
					5					5									
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
	<i>src1</i>		x	0	0	1	1	1	1	1	1	0	0	s	p		1	1	
			5	1															

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	dint	

**Description** Performs two 16-bit by 32-bit multiplies. The upper and lower halves of *src1* are treated as 16-bit signed inputs. The value in *src2* is treated as a 32-bit signed value. The products are then rounded to a 32-bit result by adding the value  $2^{14}$  and then these sums are right shifted by 15. The lower 32 bits of the two results are written into *dst\_o:dst\_e*.

If either result saturates, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst\_o:dst\_e*.

This instruction executes unconditionally and cannot be predicated.



**Note**—In the overflow case, where the 16-bit input to the MPYIR operation is 8000h and the 32-bit input is 8000 0000h, the saturation value 7FFF FFFFh is written into the corresponding 32-bit *dst* register.

**Execution**

```
if (msb16(src1) = 8000h && src2 = 80000000h), 7FFFFFFFh → dst_o
else lsb32((msb16(src1) × (src2)) + 4000h) >> 15) → dst_o;
if (lsb16(src1) = 8000h && src2 = 80000000h), 7FFFFFFFh → dst_e
else lsb32((lsb16(src1) × (src2)) + 4000h) >> 15) → dst_e
```

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYLIR](#), [MPYHIR](#)

**Examples** [Example 1](#)

MPY2IR .M2 B2,B5,B9:B8

	<b>Before instruction</b>		<b>4 cycles after instruction</b>
B2	80008001h		B8      7FFF0000h
B5	80000000h		B9      7FFFFFFFh
CSR	00010100h		CSR <sup>1</sup> 00010300h
SSR	00000000h		SSR <sup>1</sup> 00000020h

1. CSR.SAT and SSR.M2 set to 1, 5 cycles after instruction

### Example 2

```
MPY2IR .M1X A2,B5,A9:A8
```

	<b>Before instruction</b>		<b>4 cycles after instruction</b>
A2	87654321h		A8      098C16C1h
B5	12345678h		A9      EED8E38Fh
CSR	00010100h		CSR <sup>1</sup> 00010100h
SSR	00000000h		SSR <sup>1</sup> 00000000h

1. CSR.SAT and SSR.M1 unchanged by operation

## 4.184 MPY32

Multiply Signed 32-Bit  $\times$  Signed 32-Bit Into 32-Bit Result

**Syntax** **MPY32 (.unit) src1, src2, dst**

unit = .M1 or .M2

### Compatibility

#### Opcode

31	29	28	27			23	22			18				
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>						
3		1		5				5						
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	1	0	0	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>	
5		1										1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	int	

**Description** Performs a 32-bit by 32-bit multiply. *src1* and *src2* are signed 32-bit values. Only the lower 32 bits of the 64-bit result are written to *dst*.

**Execution**  
if (cond)    *src1*  $\times$  *src2*  $\rightarrow$  *dst*  
else nop

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPY32](#), [MPY32SU](#), [MPY32US](#), [MPY32U](#), [SMPY32](#)

## 4.185 MPY32

Multiply Signed 32-Bit  $\times$  Signed 32-Bit Into Signed 64-Bit Result

**Syntax** **MPY32 (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	29	28	27			23	22	18									
creg	z	dst		src2													
3	1	5		5													
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
src1	x	1	0	1	0	0	0	0	0	0	0	0	s	p			
5	1												1	1			

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	dint	

**Description** Performs a 32-bit by 32-bit multiply. *src1* and *src2* are signed 32-bit values. The signed 64-bit result is written to the register pair specified by *dst*.

**Execution** if (cond) *src1*  $\times$  *src2*  $\rightarrow$  *dst\_o:dst\_e*  
else nop

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPY32](#), [MPY32SU](#), [MPY32US](#), [MPY32U](#), [SMPY32](#)

## 4.186 MPY32SU

Multiply Signed 32-Bit  $\times$  Unsigned 32-Bit Into Signed 64-Bit Result

**Syntax** **MPY32SU** (.unit) *src1*, *src2*, *dst\_o:dst\_e*

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	29	28	27			23	22											18
<i>creg</i>	<i>z</i>			<i>dst</i>		<i>src2</i>												
3	1			5	5													
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
<i>src1</i>	<i>x</i>	1	0	1	1	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>	1	1		
5	1																	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xuint	
<i>dst</i>	dint	

**Description** Performs a 32-bit by 32-bit multiply. *src1* is a signed 32-bit value and *src2* is an unsigned 32-bit value. The signed 64-bit result is written to the register pair specified by *dst*.

**Execution**

```
if (cond)    src1 × src2 → dst_o:dst_e
else nop
```

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPY32](#), [MPY32U](#), [MPY32US](#), [SMPY32](#)

## 4.187 MPY32U

Multiply Unsigned 32-Bit  $\times$  Unsigned 32-Bit Into Unsigned 64-Bit Result

**Syntax** **MPY32U (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	29	28	27			23	22			18				
creg	z	dst				src2								
3	1	5				5								
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
src1	x	0	1	1	0	0	0	1	1	0	0	s	p	
5	1											1	1	

Opcode map field used...	For operand type...	Unit
src1	uint	.M1, .M2
src2	xuint	
dst	duint	

**Description** Performs a 32-bit by 32-bit multiply. *src1* and *src2* are unsigned 32-bit values. The unsigned 64-bit result is written to the register pair specified by *dst*.

**Execution** if (cond) *src1*  $\times$  *src2*  $\rightarrow$  *dst\_o:dst\_e*  
else nop

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPY32](#), [MPY32SU](#), [MPY32US](#), [SMPY32](#)

## 4.188 MPY32US

Multiply Unsigned 32-Bit  $\times$  Signed 32-Bit Into Signed 64-Bit Result

**Syntax** **MPY32US** (.unit) *src1*, *src2*, *dst\_o:dst\_e*

unit = .M1 or .M2

**Compatibility**

**Opcode**

31	29	28	27			23	22											18
<i>creg</i>	<i>z</i>			<i>dst</i>												<i>src2</i>		
3	1			5											5			
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
<i>src1</i>	<i>x</i>	0	1	1	0	0	1	1	1	0	0	<i>s</i>	<i>p</i>		1	1		
5	1																	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	uint	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	dint	

**Description** Performs a 32-bit by 32-bit multiply. *src1* is an unsigned 32-bit value and *src2* is a signed 32-bit value. The signed 64-bit result is written to the register pair specified by *dst*.

**Execution**

```
if (cond)    src1 × src2 → dst_o:dst_e
else nop
```

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPY32](#), [MPY32SU](#), [MPY32U](#), [SMPY32](#)

## 4.189 MPYDP

Multiply Two Double-Precision Floating-Point Values

**Syntax**    **MPYDP (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst						src2						18
creg	z															
3	1			5											5	
src1	x	0	1	1	1	0	0	0	0	0	0	0	0	0	s	p
5	1														1	1

Opcode map field used...	For operand type...	Unit
src1	dp	.M1, .M2
src2	dp	
dst	dp	

**Description**    The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*.



**Note—**

- 1) If one source is SNaN or QNaN, the result is a signed NaN\_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the exclusive-OR of the input signs.
- 2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN\_out and sets the INVAL bit.
- 3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN\_out.
- 4) A denormalized source is treated as signed 0 and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, or signed 0. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
- 5) If rounding is performed, the INEX bit is set.

**Execution**

```
if (cond)    src1 × src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
<b>Read</b>	<i>src1_l</i>	<i>src1_l</i>	<i>src1_h</i>	<i>src1_h</i>						
		<i>src2_l</i>	<i>src2_h</i>	<i>src2_l</i>	<i>src2_h</i>					
<b>Written</b>									<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M	.M	.M	.M						

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYSP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

<b>Instruction Type</b>	MPYDP
<b>Delay Slots</b>	9
<b>Functional Unit Latency</b>	4
<b>See Also</b>	<a href="#">MPY</a> , <a href="#">MPYSP</a>
<b>Example</b>	MPYDP .M1 A1:A0,A3:A2,A5:A4

<b>Before instruction</b>				<b>10 cycles after instruction</b>			
A1:A0	4021 3333h	3333 3333h	8.6	A1:A0	4021 3333h	4021 3333h	
A3:A2	C004 0000h	0000 0000h	-2.5	A3:A2	C004 0000h	0000 0000h	
A5:A4	xxxx xxxxh	xxxx xxxxh		A5:A4	C035 8000h	0000 0000h	-21.5

## 4.190 MPYH

Multiply Signed 16 MSB × Signed 16 MSB

**Syntax** **MPYH (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

Opcode															
31	29	28	27			23	22	18							
creg	z	dst					src2								
3	1	5					5								
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
src1	x	0	0	0	0	1	0	0	0	0	0	s	p		
5	1														

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xsmbs16	
dst	sint	

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

**Execution**

```
if (cond) msb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src1, src2	
Written		dst
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYHU](#), [MPYHSU](#), [MPYHUS](#), [SMPYH](#)

**Example** MPYH .M1 A1,A2,A3

Before instruction	2 cycles after instruction
A1 00230000h 35 <sup>1</sup>	A1 00230000h

A2	FFA71234h	-89 <sup>1</sup>	A2	FFA71234h	
A3	xxxxxxxxh		A3	FFFFF3D5h	-3115

1. Signed 16-MSB integer

## 4.191 MPYHI

Multiply 16 MSB × 32-Bit Into 64-Bit Result

**Syntax** **MPYHI** (.unit) *src1*, *src2*, *dst\_o:dst\_e*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	1	0	1	0	0	0	1	1	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	slong	

**Description** Performs a 16-bit by 32-bit multiply. The upper half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst\_o:dst\_e*, and sign extended to 64 bits.

**Execution**

```
if (cond) msb16(src1) × src2 → dst_o:dst_e
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
<b>Read</b>	<i>src1</i> , <i>src2</i>			
<b>Written</b>				<i>dst</i>
<b>Unit in use</b>	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYLI](#)

**Examples** [Example 1](#)

MPYHI .M1 A5,A6,A9:A8

	Before instruction	4 cycles after instruction
A5	6A32 1193h 27,186	A5 6A32 1193h
A6	B174 6CA4h -1,317,770,076	A6 B174 6CA4h

A9:A8	xxxxxxxxh	xxxx xxxxh	A9:A8	FFFF DF6Ah	DDB9 2008h
-35,824,897,286,136					

### Example 2

MPYHI .M2 B2,B5,B9:B8

Before instruction			4 cycles after instruction		
B2	1234 3497h	4660	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9:B8	xxxxxxxxh	xxxx xxxxh	B9:B8	0000 026Ah	DB88 1FECh
				2,657,972,920,300	

## 4.192 MPYHIR

Multiply 16 MSB  $\times$  32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result

**Syntax** **MPYHIR (.unit) src1, src2, dst**

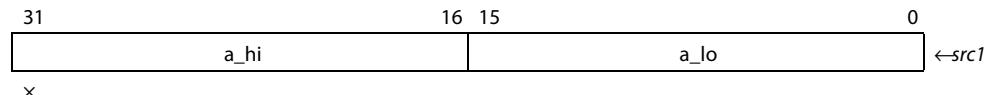
unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
creg	z	dst					src2									
3		1		5				5				5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	1	0	0	0	0	0	1	1	0	0	s	p		
5		1											1	1		

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	int	

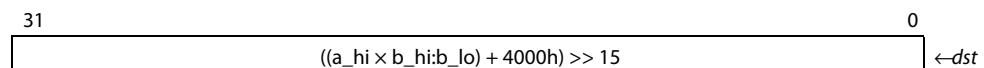
**Description** Performs a 16-bit by 32-bit multiply. The upper half of *src1* is treated as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded to a 32-bit result by adding the value  $2^{14}$  and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*.



**MPYHIR**



=



**Execution** if (cond) lsb32(((msb16(src1)  $\times$  (src2)) + 4000h) >> 15)  $\rightarrow$  dst

---

else nop

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYLIR](#)
**Example** MPYHIR .M2 B2,B5,B9

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B2	1234 3497h	4660	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9	xxxx xxxxh		B9	04D5 B710h	81,114,896

## 4.193 MPYHL

Multiply Signed 16 MSB × Signed 16 LSB

**Syntax** **MPYHL (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

Opcode																
31	29	28	27			23	22	18								
creg	z	dst				src2										
3	1					5								5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	1	0	0	1	0	0	0	0	0	0	s	p		
5	1															

Opcode map field used...	For operand type...	Unit
src1	smsb16	.M1, .M2
src2	xlsb16	
dst	sint	

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

**Execution**

```
if (cond)    msb16(src1) × lsb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src1, src2	
Written		dst
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYHLU](#), [MPYHSLU](#), [MPYHULS](#), [SMPYHL](#)

**Example** MPYHL .M1 A1,A2,A3

Before instruction	2 cycles after instruction	
A1 008A003Eh	138 <sup>1</sup>	A1 008A003Eh

A2	<input type="text" value="21FF00A7h"/>	167 <sup>2</sup>	A2	<input type="text" value="21FF00A7h"/>
A3	<input type="text" value="xxxxxxxxh"/>		A3	<input type="text" value="00005A06h"/> 23,046

1. Signed 16-MSB integer
2. Signed 16-LSB integer

## 4.194 MPYHLU

Multiply Unsigned 16 MSB × Unsigned 16 LSB

**Syntax**    **MPYHLU** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	1	1	1	1	1	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	umsb16	.M1, .M2
<i>src2</i>	xulsb16	
<i>dst</i>	uint	

**Description**    The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

**Execution**    if (cond)    msb16(*src1*) × lsb16(*src2*) → *dst*  
else nop

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type**    Multiply (16 × 16)

**Delay Slots**    1

**See Also**    [MPYHL](#), [MPYHSLU](#), [MPYHULS](#)

## 4.195 MPYHSLU

Multiply Signed 16 MSB × Unsigned 16 LSB

**Syntax** **MPYHSLU** (.unit) *src1, src2, dst*

unit = .M1 or .M2

### Opcode

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					
3		1		5							5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	1	0	1	1	1	0	0	0	0	0	s	p	1	1
5		1														

Opcode map field used...	For operand type...	Unit
<i>src1</i>	smsb16	.M1, .M2
<i>src2</i>	xulsb16	
<i>dst</i>	sint	

**Description** The signed operand *src1* is multiplied by the unsigned operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond)    msb16(src1) × lsb16(src2) → dst
else nop
```

### Pipeline

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use		.M

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYHL](#), [MPYHLU](#), [MPYHULS](#)

## 4.196 MPYHSU

Multiply Signed 16 MSB × Unsigned 16 MSB

**Syntax** **MPYHSU** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	0	0	1	1	0	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	smsb16	.M1, .M2
<i>src2</i>	xumsb16	
<i>dst</i>	sint	

**Description** The signed operand *src1* is multiplied by the unsigned operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond) msb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYH](#), [MPYHU](#), [MPYHUS](#)

**Example** MPYHSU .M1 A1,A2,A3

Before instruction			2 cycles after instruction		
A1	00230000h	35 <sup>1</sup>	A1	00230000h	
A2	FFA7FFFFh	65,447 <sup>2</sup>	A2	FFA7FFFFh	

A3      

xxxxxxxxh
-----------

A3      

0022F3D5h
-----------

 2,290,645

1. Signed 16-MSB integer
2. Unsigned 16-MSB integer

## 4.197 MPYHU

Multiply Unsigned 16 MSB × Unsigned 16 MSB

**Syntax** **MPYHU** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	0	1	1	1	1	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	umsb16	.M1, .M2
<i>src2</i>	xumsb16	
<i>dst</i>	uint	

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

**Execution**

```
if (cond) msb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYH](#), [MPYHSU](#), [MPYHUS](#)

**Example** MPYHU .M1 A1,A2,A3

Before instruction				2 cycles after instruction			
A1	00230000h	35 <sup>1</sup>		A1	00230000h		
A2	FFA71234h	65,447 <sup>1</sup>		A2	FFA71234h		
A3	xxxxxxxxh			A3	0022F3D5h	2,290,645 <sup>2</sup>	

1. Unsigned 16-MSB integer
2. Unsigned 32-bit integer

## 4.198 MPYHULS

Multiply Unsigned 16 MSB × Signed 16 LSB

**Syntax** **MPYHULS** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	1	1	0	1	0	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	umsb16	.M1, .M2
<i>src2</i>	xlsb16	
<i>dst</i>	sint	

**Description** The unsigned operand *src1* is multiplied by the signed operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond) msb16(src1) × lsb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use		.M

**Instruction Type** Multiply (16 × 16)

**Delay Slots** 1

**See Also** [MPYHL](#), [MPYHLU](#), [MPYHSLU](#)

## 4.199 MPYHUS

Multiply Unsigned 16 MSB × Signed 16 MSB

**Syntax** **MPYHUS** (.unit) *src1, src2, dst*

unit = .M1 or .M2

### Opcode

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					
3		1		5							5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	0	1	0	1	0	0	0	0	0	0	s	p	1	1
5		1														

Opcode map field used...	For operand type...	Unit
<i>src1</i>	umsb16	.M1, .M2
<i>src2</i>	xsmbsb16	
<i>dst</i>	sint	

**Description** The unsigned operand *src1* is multiplied by the signed operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond) msb16(src1) × msb16(src2) → dst
else nop
```

### Pipeline

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use		.M

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYH](#), [MPYHU](#), [MPYHSU](#)

## 4.200 MPYI

Multiply 32-Bit  $\times$  32-Bit Into 32-Bit Result

**Syntax** **MPYI (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	23	22	18
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>
3	1		5			5
17	13	12	11	7	6	5
<i>src1</i>	<i>x</i>		<i>op</i>	0	0	0
5	1		5			
				0	0	0
				1	1	0

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.M1, .M2	00100
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	cst5	.M1, .M2	00110
<i>src2</i>	xsint		
<i>dst</i>	sint		

**Description** The *src1* operand is multiplied by the *src2* operand. The lower 32 bits of the result are placed in *dst*.

**Execution**

```
if (cond)    lsb32(src1  $\times$  src2)  $\rightarrow$  dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9
<b>Read</b>	<i>src1,</i> <i>src2</i>	<i>src1,</i> <i>src2</i>	<i>src1,</i> <i>src2</i>	<i>src1,</i> <i>src2</i>					
<b>Written</b>									<i>dst</i>
<b>Unit in use</b>	.M	.M	.M	.M					

**Instruction Type** MPYI

**Delay Slots** 8

**Functional Unit Latency** 4

**See Also** [MPYID](#)

**Example** MPYI .M1X A1,B2,A3

**Before instruction**

A1

0034 5678h

3,430,008

**9 cycles after instruction**

A1

0034 5678h

B2	<input type="text" value="0011 2765h"/>	1,124,197	B2	<input type="text" value="0011 2765h"/>	
A3	<input type="text" value="xxxxxxxxh"/>		A3	<input type="text" value="CBCA 6558h"/>	-875,928,232

## 4.201 MPYID

Multiply 32-Bit  $\times$  32-Bit Into 64-Bit Result

**Syntax**    **MPYID** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22		18
<i>creg</i>	<i>z</i>			<i>dst</i>		<i>src2</i>
3	1			5		5
17	13	12	11	7	6	5
<i>src1</i>	<i>x</i>			<i>op</i>	0	0
5	1			5	0	0
					<i>s</i>	<i>p</i>
					1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.M1, .M2	01000
<i>src2</i>	xsint		
<i>dst</i>	sdint		
<i>src1</i>	cst5	.M1, .M2	01100
<i>src2</i>	xsint		
<i>dst</i>	sdint		

**Description**    The *src1* operand is multiplied by the *src2* operand. The 64-bit result is placed in the *dst* register pair.

**Execution**

```
if (cond)    lsb32(src1  $\times$  src2)  $\rightarrow$  dst_l
              msb32(src1  $\times$  src2)  $\rightarrow$  dst_h
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
<b>Read</b>	<i>src1</i> , <i>src2</i>	<i>src1</i> , <i>src2</i>	<i>src1</i> , <i>src2</i>	<i>src1</i> , <i>src2</i>						
<b>Written</b>									<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M	.M	.M	.M						

**Instruction Type**    MPYID

**Delay Slots**    9 (8 if *dst\_l* is *src* of next instruction)

**Functional Unit Latency**    4

**See Also**    [MPYI](#)

**Example**    MPYID .M1 A1,A2,A5:A4

		<b>Before instruction</b>		<b>10 cycles after instruction</b>	
A1	0034 5678h	3,430,008	A1	0034 5678h	
B2	0011 2765h	1,124,197	B2	0011 2765h	
A5:A4	xxxxxxxxh	xxxxxxxxh	A5:A4	0000 0381h	CBCA 6558h 3,856,004,703,576

## 4.202 MPYIH

Multiply 32-Bit  $\times$  16-MSB Into 64-Bit Result

**Syntax** **MPYIH** (.unit) *src2*, *src1*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					
3		1		5							5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	<i>x</i>	0	1	0	1	0	0	0	1	1	0	0	<i>s</i>	<i>p</i>	1	1
5		1														

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	slong	

**Description**

The **MPYIH** pseudo-operation performs a 16-bit by 32-bit multiply. The upper half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst\_o:dst\_e*, and sign extended to 64 bits. The assembler uses the **MPYHI** (.unit) *src1*, *src2*, *dst* instruction to perform this operation.

**Execution**

```
if (cond)    src2  $\times$  msb16(src1)  $\rightarrow$  dst_o:dst_e
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1</i> , <i>src2</i>			
Written				<i>dst</i>
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYHI](#), [MPYIL](#)

## 4.203 MPYIHR

Multiply 32-Bit  $\times$  16 MSB, Shifted by 15 to Produce a Rounded 32-Bit Result

**Syntax** **MPYIHR (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31		29	28	27			23	22	18					
creg		z	dst				src2							
3		1	5				5							
17		13	12	11	10	9	8	7	6	5				
src1		x	0	1	0	0	0	0	1	1				
5		1								0				
										s p				
										1 1				

Opcode map field used...	For operand type...	Unit
src1	int	.M1, .M2
src2	xint	
dst	int	

**Description**

The **MPYIHR** pseudo-operation performs a 16-bit by 32-bit multiply. The upper half of *src1* is treated as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded to a 32-bit result by adding the value  $2^{14}$  and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*. The assembler uses the **MPYHIR (.unit) src1, src2, dst** instruction to perform this operation.

**Execution**

```
if (cond) lsb32(((src2)  $\times$  msb16(src1)) + 4000h) >> 15)  $\rightarrow$  dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read		src1, src2		
Written				dst
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYHIR](#), [MPYILR](#)

## 4.204 MPYIL

Multiply 32-Bit  $\times$  16 LSB Into 64-Bit Result

**Syntax** **MPYIL (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27			23	22			18				
<i>creg</i>	<i>z</i>			<i>dst</i>				<i>src2</i>						
3		1		5				5						
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	1	0	1	0	1	1	1	0	0	<i>s</i>	<i>p</i>	
5		1								1		1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	slong	

**Description**

The **MPYIL** pseudo-operation performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst\_o:dst\_e*, and sign extended to 64 bits. The assembler uses the **MPYLI** (.unit) *src1, src2, dst* instruction to perform this operation.

**Execution**

```
if (cond)    src2  $\times$  lsb16(src1)  $\rightarrow$  dst_o:dst_e
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYIH](#), [MPYLI](#)

## 4.205 MPYILR

Multiply 32-Bit  $\times$  16 LSB, Shifted by 15 to Produce a Rounded 32-Bit Result

**Syntax** **MPYILR (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst					src2					18
3	1			5					5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	<i>src1</i>	x	0	0	1	1	1	0	1	1	0	0	s	p
	5		1										1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	int	

**Description**

The **MPYILR** pseudo-operation performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded to a 32-bit result by adding the value  $2^{14}$  and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*. The assembler uses the **MPYLIR (.unit) src1, src2, dst** instruction to perform this operation.

**Execution**

```
if (cond) lsb32(((src2) × lsb16(src1)) + 4000h) >> 15) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYIHR](#), [MPYLIR](#)

## 4.206 MPY LH

Multiply Signed 16 LSB × Signed 16 MSB

**Syntax** **MPY LH (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

Opcode															
31	29	28	27			23	22	18							
creg	z	dst					src2								
3	1	5					5								
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
src1	x	1	0	0	0	1	0	0	0	0	0	0	s	p	
5	1														

Opcode map field used...	For operand type...	Unit
src1	slsb16	.M1, .M2
src2	xsmbs16	
dst	sint	

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are signed by default.

**Execution**

```
if (cond) lsb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src1, src2	
Written		dst
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPY LHU](#), [MPY LSHU](#), [MPY LUHS](#), [SMPY LH](#)

**Example** MPY LH .M1 A1,A2,A3

Before instruction	2 cycles after instruction
A1 0900000Eh 14 <sup>1</sup>	A1 0900000Eh

A2	<table border="1"><tr><td>002900A7h</td></tr></table>	002900A7h	41 <sup>2</sup>	A2	<table border="1"><tr><td>002900A7h</td></tr></table>	002900A7h	
002900A7h							
002900A7h							
A3	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A3	<table border="1"><tr><td>00000023Eh</td></tr></table>	00000023Eh	574
xxxxxxxxh							
00000023Eh							

1. Signed 16-LSB integer
2. Signed 16-MSB integer

## 4.207 MPYLHU

Multiply Unsigned 16 LSB  $\times$  Unsigned 16 MSB

**Syntax** **MPYLHU** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	1	0	1	1	1	1	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	ulsb16	.M1, .M2
<i>src2</i>	xumsb16	
<i>dst</i>	uint	

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

**Execution**

```
if (cond)    lsb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYLNH](#), [MPYLSHU](#), [MPYLUHS](#)

## 4.208 MPYLI

Multiply 16 LSB × 32-Bit Into 64-Bit Result

**Syntax** **MPYLI** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	1	0	1	0	1	1	1	1	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	slong	

**Description** Performs a 16-bit by 32-bit multiply. The lower half of *src1* is used as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The result is written into the lower 48 bits of a 64-bit register pair, *dst\_o*:*dst\_e*, and sign extended to 64 bits.

**Execution**

```
if (cond) lsb16(src1) × src2 → dst_o:dst_e
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1</i> , <i>src2</i>			
Written				<i>dst</i>
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYHI](#)

**Examples** [Example 1](#)

MPYLI .M1 A5,A6,A9:A8

Before instruction			4 cycles after instruction		
A5	6A32 1193h	4499	A5	6A32 1193h	
A6	B174 6CA4h	-1,317,770,076	A6	B174 6CA4h	

A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	FFFF FA9Bh	A111 462Ch
-5,928,647,571,924					

### Example 2

MPYLI .M2 B2,B5,B9:B8

Before instruction			4 cycles after instruction		
B2	1234 3497h	13,463	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	0000 06FBh	E9FA 7E81h 7,679,032,065,665

## 4.209 MPYLIR

Multiply 16 LSB × 32-Bit, Shifted by 15 to Produce a Rounded 32-Bit Result

**Syntax**    **MPYLIR** (.unit) *src1*, *src2*, *dst*

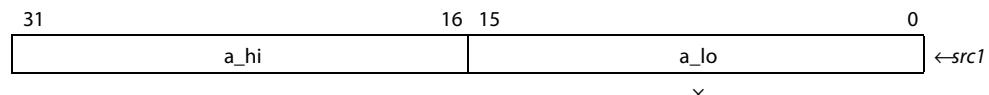
unit = .M1 or .M2

**Opcode**

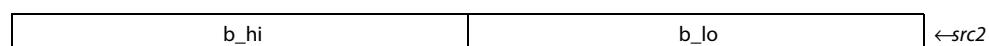
31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	0	0	1	1	1	1	0	1	1	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	int	

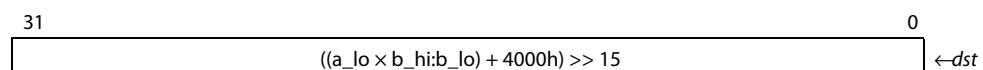
**Description**    Performs a 16-bit by 32-bit multiply. The lower half of *src1* is treated as a signed 16-bit input. The value in *src2* is treated as a signed 32-bit value. The product is then rounded into a 32-bit result by adding the value  $2^{14}$  and then this sum is right shifted by 15. The lower 32 bits of the result are written into *dst*.



**MPYLIR**



=



**Execution**    if (cond) lsb32(((lsb16(*src1*) × (*src2*) + 4000h) >> 15) → *dst*

```
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src1, src2</i>		
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYHIR](#)

**Example** MPYLIR .M2 B2,B5,B9

<b>Before instruction</b>			<b>4 cycles after instruction</b>		
B2	1234 3497h	13,463	B2	1234 3497h	
B5	21FF 50A7h	570,380,455	B5	21FF 50A7h	
B9	xxxx xxxxh		B9	0DF7 D3F5h	234,345,461

## 4.210 MPYLSHU

Multiply Signed 16 LSB × Unsigned 16 MSB

**Syntax** **MPYLSHU** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22	18					
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	1	0	0	1	1	0	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	slsb16	.M1, .M2
<i>src2</i>	xumsb16	
<i>dst</i>	sint	

**Description** The signed operand *src1* is multiplied by the unsigned operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond)    lsb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use		.M

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPYLH](#), [MPYLUH](#), [MPYLUHS](#)

## 4.211 MPYLUHS

Multiply Unsigned 16 LSB × Signed 16 MSB

**Syntax** **MPYLUHS** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	1	0	1	0	1	0	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	ulsb16	.M1, .M2
<i>src2</i>	xsmbs16	
<i>dst</i>	sint	

**Description** The unsigned operand *src1* is multiplied by the signed operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond)    lsb16(src1) × msb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use		.M

**Instruction Type** Multiply (16 × 16)

**Delay Slots** 1

**See Also** [MPYLH](#), [MPYLHU](#), [MPYLSHU](#)

## 4.212 MPYSP

Multiply Two Single-Precision Floating-Point Values

**Syntax**    **MPYSP** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22						18				
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>				
3	1			5						5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	1	1	1	0	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>
5	1												1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sp	.M1, .M2
<i>src2</i>	xsp	
<i>dst</i>	sp	

**Description**    The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*.



**Note—**

- 1) If one source is SNaN or QNaN, the result is a signed NaN\_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the exclusive-OR of the input signs.
- 2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN\_out and sets the INVAL bit.
- 3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN\_out.
- 4) A denormalized source is treated as signed 0 and the DENN bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, or signed 0. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
- 5) If rounding is performed, the INEX bit is set.

**Execution**

```
if (cond)    src1 × src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		<i>.M</i>		

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

<b>Instruction Type</b>	Four-cycle
<b>Delay Slots</b>	3
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">MPY</a> , <a href="#">MPYDP</a> , <a href="#">MPYSP2DP</a>
<b>Example</b>	MPYSP .M1X A1,B2,A3

<b>Before instruction</b>			<b>4 cycles after instruction</b>				
A1	<table border="1"><tr><td>C020 0000h</td></tr></table>	C020 0000h	-2.5	A1	<table border="1"><tr><td>C020 0000h</td></tr></table>	C020 0000h	
C020 0000h							
C020 0000h							
B2	<table border="1"><tr><td>4109 999Ah</td></tr></table>	4109 999Ah	8.6	B2	<table border="1"><tr><td>4109 999Ah</td></tr></table>	4109 999Ah	
4109 999Ah							
4109 999Ah							
A3	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh		A3	<table border="1"><tr><td>C1AC 0000h</td></tr></table>	C1AC 0000h	-21.5
xxxx xxxxh							
C1AC 0000h							

## 4.213 MPYSPDP

Multiply Single-Precision Floating-Point Value  $\times$  Double-Precision Floating-Point Value

**Syntax**    **MPYSPDP** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22						18				
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>				
3	1			5						5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	1	0	1	1	1	0	1	1	0	0	<i>s</i>	<i>p</i>
5	1											1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sp	.M1, .M2
<i>src2</i>	xsp	
<i>dst</i>	sp	

**Description**

The single-precision *src1* operand is multiplied by the double-precision *src2* operand to produce a double-precision result. The result is placed in *dst*.



**Note—**

- 1) If one source is SNaN or QNaN, the result is a signed NaN\_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the exclusive-OR of the input signs.
- 2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN\_out and sets the INVAL bit.
- 3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN\_out.
- 4) A denormalized source is treated as signed 0 and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, or signed 0. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
- 5) If rounding is performed, the INEX bit is set.

**Execution**

```
if (cond)    src1 × src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
<b>Read</b>	<i>src1</i> , <i>src2_l</i>	<i>src1</i> ,					
		<i>src2_h</i>					
<b>Written</b>						<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M		.M				

The low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, **MPYSPDP**, **MPYSP2DP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type** MPYSPDP

**Delay Slots** 6

**Functional Unit Latency** 3

**See Also** [MPY](#), [MPYDP](#), [MPYSP](#), [MPYSP2DP](#)

## 4.214 MPYSP2DP

Multiply Two Single-Precision Floating-Point Values for Double-Precision Result

**Syntax** **MPYSP2DP** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22						18				
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>				
3	1			5						5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	1	0	1	1	1	1	1	1	0	0	<i>s</i>	<i>p</i>
5	1											1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sp	.M1, .M2
<i>src2</i>	xsp	
<i>dst</i>	sp	

**Description** The *src1* operand is multiplied by the *src2* operand to produce a double-precision result. The result is placed in *dst*.



**Note—**

- 1) If one source is SNaN or QNaN, the result is a signed NaN\_out. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the exclusive-OR of the input signs.
- 2) Signed infinity multiplied by signed infinity or a normalized number (other than signed 0) returns signed infinity. Signed infinity multiplied by signed 0 returns a signed NaN\_out and sets the INVAL bit.
- 3) If one or both sources are signed 0, the result is signed 0 unless the other source is NaN or signed infinity, in which case the result is signed NaN\_out.
- 4) A denormalized source is treated as signed 0 and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, or signed 0. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
- 5) If rounding is performed, the INEX bit is set.

**Execution**

```
if (cond)    src1 × src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5
<b>Read</b>			<i>src1, src2</i>		
<b>Written</b>					<i>dst_l</i>
<b>Unit in use</b>			<i>.M</i>		<i>dst_h</i>

The low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, **MPYSPDP**, **MPYSP2DP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type** Five-cycle

**Delay Slots** 4

**Functional Unit Latency** 2

**See Also** [MPY](#), [MPYDP](#), [MPYSP](#), [MPYSP2DP](#)

## 4.215 MPYSU

Multiply Signed 16 LSB × Unsigned 16 LSB

**Syntax** **MPYSU** (.unit) *src1, src2, dst*

unit = .M1 or .M2

### Opcode

31	29	28	27	23	22	18
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>
3	1		5			5
17	13	12	11	7	6	5
<i>src1</i>	<i>x</i>		<i>op</i>	0	0	0
5	1		5			0
				1	1	<i>s</i>
						<i>p</i>

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	slsb16	.M1, .M2	11011
<i>src2</i>	xulsb16		
<i>dst</i>	sint		
<i>src1</i>	scst5	.M1, .M2	11110
<i>src2</i>	xulsb16		
<i>dst</i>	sint		

**Description** The signed operand *src1* is multiplied by the unsigned operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond) lsb16(src1) × lsb16(src2) → dst
else nop
```

### Pipeline

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Multiply ( $16 \times 16$ )

**Delay Slots** 1

**See Also** [MPY](#), [MPYU](#), [MPYUS](#)

**Example** MPYSU .M1 13,A1,A2

### Before instruction

A1	3497FFF3h	65,523 <sup>1</sup>
----	-----------	---------------------

### 2 cycles after instruction

A1	3497FFF3h
----	-----------

A2      

xxxxxxxxh
-----------

A2      

000CFF57h
-----------

 851,779

1. Unsigned 16-LSB integer

## 4.216 MPYSU4

Multiply Signed  $\times$  Unsigned, Four 8-Bit Pairs for Four 8-Bit Results

**Syntax** **MPYSU4 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
creg	z	dst					src2									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	0	0	1	0	1	1	1	1	0	0	s	p		
5	1												1	1		

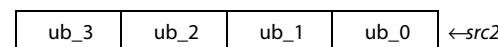
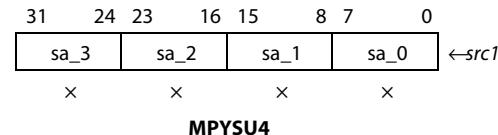
Opcode map field used...	For operand type...	Unit
src1	s4	.M1, .M2
src2	xu4	
dst	dws4	

**Description**

Returns the product between four sets of packed 8-bit values producing four signed 16-bit results. The four signed 16-bit results are packed into a 64-bit register pair, *dst\_o:dst\_e*. The values in *src1* are treated as signed 8-bit packed quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*:

- The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst\_e*.
- The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst\_e*.
- The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst\_o*.
- The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst\_o*.



=

63	48	47	32	31	16	15	0
sa_3 $\times$ ub_3	sa_2 $\times$ ub_2	sa_1 $\times$ ub_1	sa_0 $\times$ ub_0				

$\leftarrow \text{dst}_o:\text{dst}_e$

**Execution** if (cond) {

```

        (sbyte0(src1) < ubyte0(src2)) → lsb16(dst_e);
        (sbyte1(src1) < ubyte1(src2)) → msb16(dst_e);
        (sbyte2(src1) < ubyte2(src2)) → lsb16(dst_o);
        (sbyte3(src1) < ubyte3(src2)) → msb16(dst_o)

    }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		src1, src2		
Written				dst
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYU4](#)

**Examples** [Example 1](#)

MPYSU4 .M1 A5,A6,A9:A8

<b>Before instruction</b>				<b>4 cycles after instruction</b>			
A5	6A 32 11 93h	106 50 17 -109 signed	A5	6A 32 11 93h			
A6	B1 74 6C A4h	177 116 108 164 unsigned	A6	B1 74 6C A4h			
A9:A8	xxxx xxxxh	xxxx xxxxh	A9:A8	494A 16A8h	072C BA2Ch	18762 5800	1386 -17876 signed

**Example 2**

MPYSU4 .M2 B5,B6,B9:B8

<b>Before instruction</b>				<b>4 cycles after instruction</b>			
B5	3F F6 50 10h	63 -10 80 16 signed	B5	3F F6 50 10h			
B6	C3 56 02 44h	195 86 2 68 unsigned	B6	C3 56 02 44h			
B9:B8	xxxx xxxxh	xxxx xxxxh	B9:B8	2FFD FCA4h	00A0 0440h	12285 -680	160 1088 signed

## 4.217 MPYU

Multiply Unsigned 16 LSB × Unsigned 16 LSB

**Syntax** **MPYU (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
creg	z	dst					src2									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	1	1	1	1	1	1	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
src1	ulsb16	.M1, .M2
src2	xulsb16	
dst	uint	

**Description** The *src1* operand is multiplied by the *src2* operand. The result is placed in *dst*. The source operands are unsigned by default.

**Execution**

```
if (cond)    lsb16(src1) × lsb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src1, src2	
Written		dst
Unit in use	.M	

**Instruction Type** Multiply (16 × 16)

**Delay Slots** 1

**See Also** [MPY](#), [MPYSU](#), [MPYUS](#)

**Example** MPYU .M1 A1,A2,A3

Before instruction				2 cycles after instruction			
A1	00000123h	291 <sup>1</sup>		A1	00000123h		
A2	0F12FA81h	64,129 <sup>1</sup>		A2	0F12FA81h		
A3	xxxxxxxxh			A3	011C C0A3h	18,661,539 <sup>2</sup>	

- 
- 1. Unsigned 16-LSB integer
  - 2. Unsigned 32-bit integer

## 4.218 MPYU2

Multiply Unsigned by Unsigned, Packed 16-bit

**Syntax** **MPYU2 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results

31	29	28	27	23 22	18 17	13	12	11	10	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	0	opfield	1	1	0	0	s	p

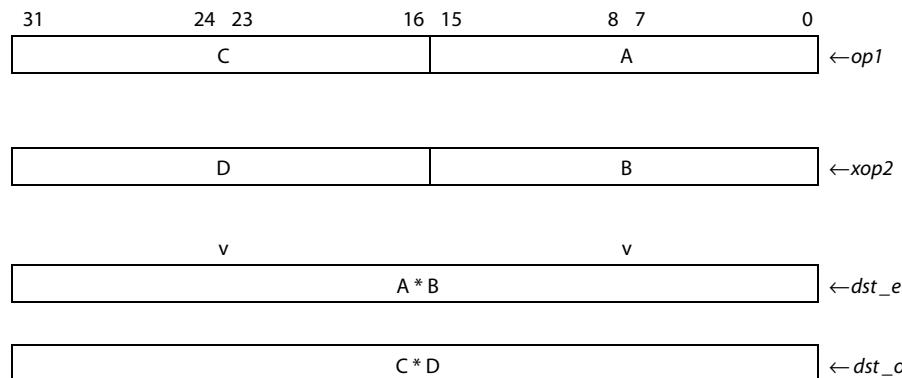
3                    5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dwdst	.M1 or .M2	01000

**Description** The MPY2 instruction performs 16-bit multiplication between unsigned packed 16-bit quantities. The values in *op1* and *xop2* are treated as unsigned packed 16-bit quantities. The 32-bit results are placed in a 64-bit register pair.

The product of the lower 16-bit quantities in *op1* and *xop2* is written to the even register of the destination, *dst\_e*. The product of the upper 16-bit quantities in *op1* and *xop2* is written to the odd register of the destination, *dst\_o*.

Effectively, **MPY2 op1, xop2, dst\_o:dst\_e** performs the same operation as **MPYU op1, xop2, dst\_e** and **MPYUH op1, xop2, dst\_o** together.



Where (unsigned int) dst\_e = (unsigned short)((unsigned short) a \* (unsigned short) b);  
and (unsigned int) dst\_o = (unsigned short)((unsigned short) c \* (unsigned short) d);

---

<b>Execution</b>	<pre> if(cond) {     ulsb16(src1) x ulsb(src2) -&gt; dst_e     umsb16(src1) x umsb(src2) -&gt; dst_o } else nop </pre>
<b>Instruction Type</b>	
<b>Delay Slots</b>	3
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">MPYSU4</a>
<b>Example</b>	<pre> A12 == 0x80018001 A23 == 0x80018001 MPYU2 .M A12,A23,A11:A10 A11 == 0x40010001 A10 == 0x40010001  A12 == 0x7fff7fff A23 == 0x7fff7fff MPYU2 .M A12,A23,A11:A10 A11 == 0x3ffff0001 A10 == 0x3ffff0001  A12 == 0x80017fff A23 == 0x7fff8001 MPYU2 .M A12,A23,A11:A10 A11 == 0x3fffffff A10 == 0x3fffffff  A12 == 0xc333c333 A23 == 0x3ccdc333 MPYU2 .M A12,A23,A11:A10 A11 == 0x2e5c43d7 A10 == 0x94d6bc29  A12 == 0x80008000 A23 == 0x80008000 MPYU2 .M A12,A23,A11:A10 A11 == 0x40000000 A10 == 0x40000000  A12 == 0x321089ab A23 == 0x87654321 MPYU2 .M A12,A23,A11:A10 A11 == 0x1a7a3050 A10 == 0x2419800b </pre>

## 4.219 MPYU4

Multiply Unsigned  $\times$  Unsigned, Four 8-Bit Pairs for Four 8-Bit Results

**Syntax** **MPYU4 (.unit) src1, src2, dst\_o:dst\_e**

unit = .M1 or .M2

**Opcode**

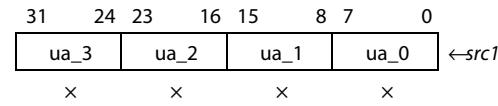
31	29	28	27	dst					src2					18
3	1			5					5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
src1	x	0	0	0	1	0	0	1	1	1	0	0	s	p

Opcode map field used...	For operand type...	Unit
src1	u4	.M1, .M2
src2	xu4	
dst	dwu4	

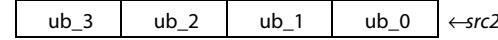
**Description** Returns the product between four sets of packed 8-bit values producing four unsigned 16-bit results that are packed into a 64-bit register pair, *dst\_o:dst\_e*. The values in both *src1* and *src2* are treated as unsigned 8-bit packed data.

For each pair of 8-bit quantities in *src1* and *src2*, the unsigned 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*:

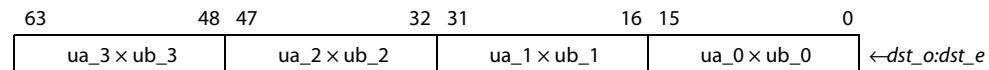
- The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst\_e*.
- The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst\_e*.
- The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst\_o*.
- The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst\_o*.



**MPYU4**



=



**Execution** if (cond) {

```

(ubyte0(src1) < ubyte0(src2)) → lsb16(dst_e);
(ubyte1(src1) < ubyte1(src2)) → msb16(dst_e);
(ubyte2(src1) < ubyte2(src2)) → lsb16(dst_o);
(ubyte3(src1) < ubyte3(src2)) → msb16(dst_o)

}
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		src1, src2		
Written				dst
Unit in use	.M			

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYSU4](#)

**Examples** [Example 1](#)

MPYU4 .M1 A5,A6,A9:A8

<b>Before instruction</b>				<b>4 cycles after instruction</b>			
A5	68 32 C1 93h	104 50 193 147	unsigned	A5	68 32 C1 93h		
A6	B1 74 2C ABh	177 116 44 171	unsigned	A6	B1 74 2C ABh		
A9:A8	xxxx xxxxh	xxxx xxxxh		A9:A8	47E8 16A8h	212C 6231h	
					18408 5800	8492 25137	unsigned

**Example 2**

MPYU4 .M2 B2,B5,B9:B8

<b>Before instruction</b>				<b>4 cycles after instruction</b>			
B2	3D E6 50 7Fh	61 230 80 127	unsigned	B2	3D E6 50 7Fh		
B5	C3 56 02 44h	195 86 2 68	unsigned	B5	C3 56 02 44h		
B9:B8	xxxx xxxxh	xxxx xxxxh		B9:B8	2E77 4D44h	00A0 21BCh	
					11895 19780	160 8636	unsigned

## 4.220 MPYUS

Multiply Unsigned 16 LSB × Signed 16 LSB

**Syntax** **MPYUS** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>src1</i>	x	1	1	1	0	1	0	0	0	0	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	ulsb16	.M1, .M2
<i>src2</i>	xlsb16	
<i>dst</i>	sint	

**Description** The unsigned operand *src1* is multiplied by the signed operand *src2*. The result is placed in *dst*. The S is needed in the mnemonic to specify a signed operand when both signed and unsigned operands are used.

**Execution**

```
if (cond) lsb16(src1) × lsb16(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Multiply (16 × 16)

**Delay Slots** 1

**See Also** [MPY](#), [MPYU](#), [MPYSU](#)

**Example** MPYUS .M1 A1,A2,A3

Before instruction			2 cycles after instruction		
A1	1234FFA1h	65,441 <sup>1</sup>	A1	1234FFA1h	
A2	1234FFA1h	-95 <sup>2</sup>	A2	1234FFA1h	

A3      

xxxxxxxxh
-----------

A3      

FFA12341h
-----------

 -6,216,895

1. Unsigned 16-LSB integer
2. Signed 16-LSB integer

## 4.221 MPYUS4

Multiply Unsigned  $\times$  Signed, Four 8-Bit Pairs for Four 8-Bit Results

**Syntax** **MPYUS4 (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27						23	22						18
creg	z	dst					src2									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	0	0	1	0	1	1	1	1	0	0	s	p		
5	1												1	1		

Opcode map field used...	For operand type...	Unit
src1	s4	.M1, .M2
src2	xu4	
dst	dws4	

**Description**

The **MPYUS4** pseudo-operation returns the product between four sets of packed 8-bit values, producing four signed 16-bit results. The four signed 16-bit results are packed into a 64-bit register pair, *dst\_o:dst\_e*. The values in *src1* are treated as signed 8-bit packed quantities; whereas, the values in *src2* are treated as unsigned 8-bit packed data. The assembler uses the **MPYSU4 (.unit)src1, src2, dst** instruction to perform this operation.

For each pair of 8-bit quantities in *src1* and *src2*, the signed 8-bit value from *src1* is multiplied with the unsigned 8-bit value from *src2*:

- The product of *src1* byte 0 and *src2* byte 0 is written to the lower half of *dst\_e*;
- The product of *src1* byte 1 and *src2* byte 1 is written to the upper half of *dst\_e*;
- The product of *src1* byte 2 and *src2* byte 2 is written to the lower half of *dst\_o*;
- The product of *src1* byte 3 and *src2* byte 3 is written to the upper half of *dst\_o*.

**Execution**

```
if (cond) {
    (ubyte0(src2) x sbyte0(src1)) → lsb16(dst_e);
    (ubyte1(src2) x sbyte1(src1)) → msb16(dst_e);
    (ubyte2(src2) x sbyte2(src1)) → lsb16(dst_o);
    (ubyte3(src2) x sbyte3(src1)) → msb16(dst_o)
```

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
Read		<i>src1, src2</i>		
Written				<i>dst</i>
Unit in use		.M		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPYSU4](#), [MPYU4](#)

## 4.222 MV

Move From Register to Register

**Syntax**    **MV (.unit) src2, dst**

unit = .L1, .L2, .S1, .S2, .D1, .D2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L, .S, .D	LSDmvto	<a href="#">Figure G-1</a>
	LSDmvfr	<a href="#">Figure G-2</a>

**Opcode**    .L unit

31	29	28	27		23	22		18	17	16
creg	z		dst		src2			0	0	
3	1		5		5					
15	14	13	12	11	5	4	3	2	1	0
0	0	0	0	op	1	1	0	s	p	
				7				1	1	

Opcode map field used...	For operand type...	Unit	Opfield
src2	xsint	.L1, .L2	0000010
dst	sint		
src2	slong	.L1, .L2	0100000
dst	slong		

31	29	28	27		23	22		18
creg	z		dst		src2			
3	1		5		5			
17	13	12	11	10	9	8	7	6
src1	0	1	1	1	1	1	1	0
								1
								1

Opcode map field used...	For operand type...	Unit
src1	scst5	.L1, .L2
src2	xuint	
dst	uint	

**Opcode**    .S unit

31 29 28 27			23 22			18 17 16		
creg	z	dst	src2			0	0	
3	1	5				5		
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 0 0 x 0 0 0 1 1 0 1 0 0 0 0 s p							
	1						1	1

Opcode map field used...	For operand type...	Unit
src2	xsint	.S1, .S2
dst	sint	

**Opcode** .D unit

31 29 28 27			23 22			18 17 16		
creg	z	dst	src2			0	0	
3	1	5				5		
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 s p							
	1						1	1

Opcode map field used...	For operand type...	Unit
src2	sint	.D1, .D2
dst	sint	

**Description** The **MV** pseudo-operation moves a value from one register to another. The assembler will either use the **ADD** (.unit) 0, *src2*, *dst* or the **OR** (.unit) 0, *src2*, *dst* operation to perform this task.

**Execution** if (cond) 0 + *src2* → *dst*  
else nop

**Instruction Type** Single-cycle

**Delay Slots** 0

## 4.223 MVC

Move Between Control File and Register File

**Syntax** **MVC (.unit) src2, dst**

unit = .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Sx1	<a href="#">Figure F-26</a>

**Opcode**

**Operands when moving from the control file to the register file:**

31	29	28	27	23 22		18
<i>creg</i>	<i>z</i>			<i>dst</i>		<i>crlo</i>
3	1			5		5
17	13	12	11	10	9	8
<i>crhi</i>	<i>x</i>	0	0	1	1	1
5	1					1
1						1

Opcode map field used...	For operand type...	Unit
<i>crlo</i>	ucst5	
<i>dst</i>	uint	.S2
<i>crhi</i>	ucst5	

**Description**

The contents of the control file specified by the *crhi* and *crlo* fields is moved to the register file specified by the *dst* field. Valid assembler values for *crlo* and *crhi* are shown in [Table 4-10](#).

**Operands when moving from the register file to the control file:**

31	29	28	27	23 22		18
<i>creg</i>	<i>z</i>			<i>crlo</i>		<i>src2</i>
3	1			5		5
17	13	12	11	10	9	8
<i>crhi</i>	<i>x</i>	0	0	1	1	1
5	1					1
1						1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.S2
<i>crlo</i>	ucst5	
<i>crhi</i>	ucst5	

**Description** The contents of the register file specified by the *src2* field is moved to the control file specified by the *crhi* and *crlo* fields. Valid assembler values for *crlo* and *crhi* are shown in [Table 4-10](#).

**Execution**

```
if (cond)    src2 → dst
else nop
```



**Note**—The **MVC** instruction executes only on the B side (.S2). Refer to the individual control register descriptions for specific behaviors and restrictions in accesses via the **MVC** instruction.

**Pipeline**

Pipeline Stage	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.S2

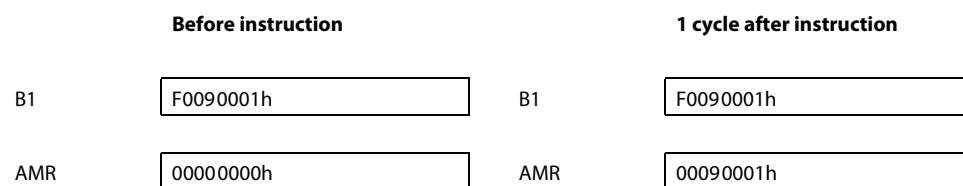
**Instruction Type** Single-cycle

Any write to the ISR or ICR (by the **MVC** instruction) effectively has one delay slot because the results cannot be read (by the **MVC** instruction) in the IFR until two cycles after the write to the ISR or ICR.

**Delay Slots**

0

**Example** `MVC .S2 B1,AMR`



**Note**—The six MSBs of the AMR are reserved and therefore are not written to.

Table 4-10 contains the register addresses required to access the control registers.

**Table 4-10 Register Addresses for Accessing the Control Registers**

<b>Acronym</b>	<b>Register Name</b>	<b>Address</b>		<b>Supervisor</b>	<b>User</b>
		<i>crhi</i>	<i>crlo</i>	<b>Read/Write<sup>1</sup></b>	<b>Read/Write<sup>1</sup></b>
AMR	Addressing mode register	00000	00000	R, W	R, W
		0xxxx	00000		
CSR	Control status register	00000	00001	R, W*	R, W*
		00001	00001		
		0xxxx	00001		
DNUM	DSP core number register	00000	10001	R	R
ECR	Exception clear register	00000	11101	W	X
EFR	Exception flag register	00000	11101	R	X
FADCR	Floating-point adder configuration register	00000	10010	R, W	R, W
FAUCR	Floating-point auxiliary configuration register	00000	10011	R, W	R, W
FMCR	Floating-point multiplier configuration register	00000	10100	R, W	R, W
GFPGFR	Galois field multiply control register		11000	R, W	R, W
GPLYA	GMPY A-side polynomial register	00000	10110	R, W	R, W
GPLYB	GMPY B-side polynomial register	00000	10111	R, W	R, W
ICR	Interrupt clear register	00000	00011	W	X
		0xxxx	00011		
IER	Interrupt enable register	00000	00100	R, W	X
		0xxxx	00100		
IERR	Internal exception report register	00000	11111	R,W	X
IFR	Interrupt flag register	00000	00010	R	X
		00010	00010		
ILC	Inner loop count register	00000	01101	R, W	R, W
IRP	Interrupt return pointer register	00000	00110	R, W	R, W
		0xxxx	00110		
ISR	Interrupt set register	00000	00010	W	X
		0xxxx	00010		
ISTP	Interrupt service table pointer register	00000	00101	R, W	X
		0xxxx	00101		
ITSR	Interrupt task state register	00000	11011	R, W	X
NRP	Nonmaskable interrupt or exception return pointer register	00000	00111	R, W	R, W
		0xxxx	00111		
NTSR	NMI/Exception task state register	00000	11100	R, W	X
PCE1	Program counter, E1 phase	00000	10000	R	R
		10000	10000		
REP	Restricted entry point address register	00000	01111	R, W	X
RILC	Reload inner loop count register	00000	01110	R, W	R, W
SSR	Saturation status register	00000	10101	R, W	R, W
TSCH	Time-stamp counter (high 32 bits) register	00000	01011	R	R
TSCL	Time-stamp counter (low 32 bits) register	00000	01010	R	R
TSR	Task state register	00000	11010	R, W*	R,W*

- 
1. **Legend:** R = Readable by the **MVC** instruction; W = Writable by the **MVC** instruction; W\* = Partially writable by the **MVC** instruction; X = Access causes exception

## 4.224 MVD

Move From Register to Register, Delayed

**Syntax**    **MVD** (.unit) *src2*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27	23 22					18	17	16	
	<i>creg</i>	<i>z</i>		<i>dst</i>					<i>src2</i>			
3		1		5				5				
15	14	13	12	11	10	9	8	7	6	5	4	3
0	1	0	x	0	0	0	0	1	1	1	1	0
			1									1
												1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xint	.M1, .M2
<i>dst</i>	int	

**Description**    Moves data from the *src2* register to the *dst* register over 4 cycles. This is done using the multiplier path.

```

MVD      .M2x    A0,    B0      ;
NOP
NOP
NOP      ; B0 = A0
;
```

**Execution**    if (cond)    *src2* → *dst*  
else nop

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src2</i>			
Written				<i>dst</i>
Unit in use	.M			

**Instruction Type**    Four-cycle

**Delay Slots**    3

**Example**    MVD .M2X A5,B8

**Before instruction**

A5	6A32 1193h
B8	xxxx xxxxh

**4 cycles after instruction**

A5	6A32 1193h
B8	6A32 1193h

4.225 MVK

## Move Signed Constant Into Register and Sign Extend

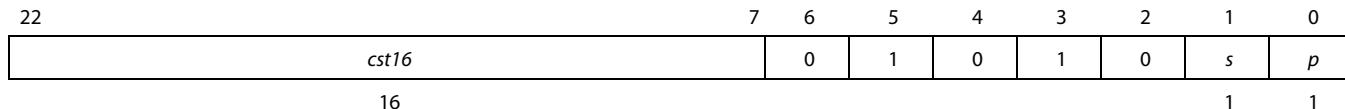
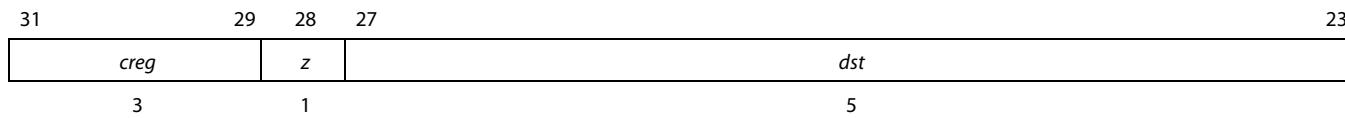
**Syntax** MVK (.unit) *cst, dst*

unit = .L1, .L2, .S1, .S2, .D1, .D2

## ***Compact Instruction Format***

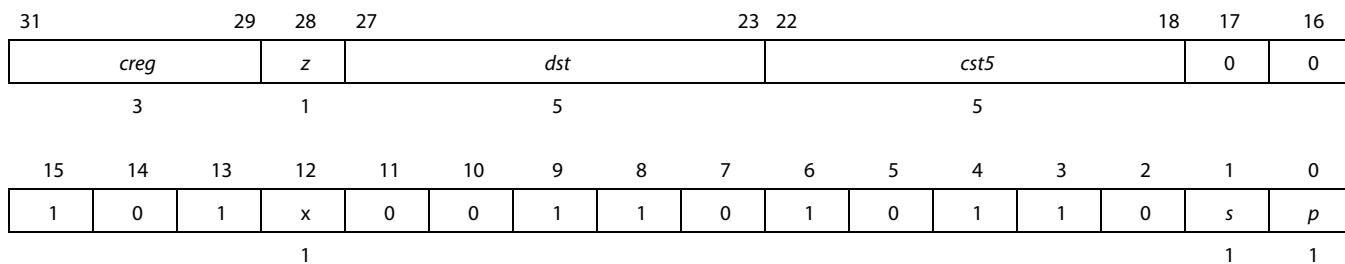
<b>Unit</b>	<b>Opcode Format</b>	<b>Figure</b>
.L	Lx5	<a href="#">Figure D-8</a>
.S	Smvk8	<a href="#">Figure F-19</a>
.L, .S, .D	LSDx1c	<a href="#">Figure G-3</a>
	LSDx1	<a href="#">Figure G-4</a>

*Opcode* .S unit



<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>cst16</i>	<i>scst16</i>	.S1,.S2
<i>dst</i>	<i>sint</i>	

*Opcode* .L unit



<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>cst5</i>	<i>scst5</i>	.L1, .L2
<i>dst</i>	<i>sint</i>	

*Opcode* .D unit

31	29	28	27						23	22						18
<i>creg</i>	<i>z</i>	<i>dst</i>					<i>src2</i>									
3	1	5					5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
<i>cst5</i>	0	0	0	0	0	0	1	0	0	0	0	0	<i>s</i>	<i>p</i>	1	1
	5															

Opcode map field used...	For operand type...	Unit
<i>cst5</i>	<i>scst5</i>	.D1, .D2
<i>dst</i>	<i>sint</i>	

**Description** The constant *cst* is sign extended and placed in *dst*. The .S unit form allows for a 16-bit signed constant.

Since many nonaddress constants fall into a 5-bit sign constant range, this allows the flexibility to schedule the MVK instruction on the .L or .D units. In the .D unit form, the constant is in the position normally used by *src1*, as for address math.

In most cases, the C6000 assembler and linker issue a warning or an error when a constant is outside the range supported by the instruction. In the case of MVK.S, a warning is issued whenever the constant is outside the signed 16-bit range, -32768 to 32767 (or FFFF8000h to 00007FFFh).

For example:

```
MVK .S1 0x00008000, A0
```

will generate a warning; whereas:

```
MVK .S1 0xFFFF8000, A0
```

will not generate a warning.

**Execution**

```
if (cond) scst → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	
Written	<i>dst</i>
Unit in use	.L, .S, or .D

**Instruction Type** Single cycle

**Delay Slots** 0

**See Also** [MVKH/MVKLH, MVKL](#)

**Examples** **Example 1**

MVK .L2 -5, B8

**Before instruction**

B8

xxxx xxxxh

**1 cycle after instruction**

B8

FFFF FFFBh

**Example 2**

MVK .D2 14, B8

**Before instruction**

B8

xxxx xxxxh

**1 cycle after instruction**

B8

0000 000Eh

## 4.226 MVKH/MVKLH

Move 16-Bit Constant Into Upper Bits of Register

**Syntax**    **MVKH** (.unit) *cst, dst*

or

**MVKLH** (.unit) *cst, dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	23									
<i>creg</i>	<i>z</i>			<i>dst</i>									
3	1			5									
22			7 6 5 4 3 2 1 0										
	<i>cst16</i>		<i>h</i> 1 0 1 0 <i>s</i> <i>p</i>										
	16		1	1 1									
<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left; padding: 2px;">Opcode map field used...</th> <th style="text-align: left; padding: 2px;">For operand type...</th> <th style="text-align: right; padding: 2px;">Unit</th> </tr> </thead> <tbody> <tr> <td style="text-align: left; padding: 2px;"><i>cst16</i></td> <td style="text-align: left; padding: 2px;">uscst16</td> <td style="text-align: right; padding: 2px;">.S1, S2</td> </tr> <tr> <td style="text-align: left; padding: 2px;"><i>dst</i></td> <td style="text-align: left; padding: 2px;">sint</td> <td style="text-align: right; padding: 2px;"></td> </tr> </tbody> </table>					Opcode map field used...	For operand type...	Unit	<i>cst16</i>	uscst16	.S1, S2	<i>dst</i>	sint	
Opcode map field used...	For operand type...	Unit											
<i>cst16</i>	uscst16	.S1, S2											
<i>dst</i>	sint												

**Description**

The 16-bit constant, *cst16*, is loaded into the upper 16 bits of *dst*. The 16 LSBs of *dst* are unchanged. For the **MVKH** instruction, the assembler encodes the 16 MSBs of a 32-bit constant into the *cst16* field of the opcode. For the **MVKLH** instruction, the assembler encodes the 16 LSBs of a constant into the *cst16* field of the opcode



**Note**—Use the **MVK** instruction (see [MVK](#)) to load 16-bit constants. The assembler generates a warning for any constant over 16 bits. To load 32-bit constants, such as 1234 5678h, use the following pair of instructions:

MVKL 0x12345678

MVKH 0x12345678

If you are loading the address of a label, use:

MVKL label

MVKH label

**Execution**

For the **MVKLH** instruction:

```
if (cond) ((cst15..0) << 16) or (dst15..0) → dst
else nop
```

For the **MVKH** instruction:

```
if (cond) ((cst31..16) << 16) or (dst15..0) → dst
```

```
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**Examples** **Example 1**

```
MVKH .S1 0A329123h,A1
```

**Before instruction**

A1	<span style="border: 1px solid black; padding: 2px;">00007634h</span>
----	---

**1 cycle after instruction**

A1	<span style="border: 1px solid black; padding: 2px;">0A327634h</span>
----	---

**Example 2**

```
MVKLH .S1 7A8h,A1
```

**Before instruction**

A1	<span style="border: 1px solid black; padding: 2px;">FFFFF25Ah</span>
----	---

**1 cycle after instruction**

A1	<span style="border: 1px solid black; padding: 2px;">07A8F25Ah</span>
----	---

## 4.227 MVKL

Move Signed Constant Into Register and Sign Extend

**Syntax** **MVKL (.unit) cst, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	23
<i>creg</i>	<i>z</i>			<i>dst</i>
3	1			5
22			7    6    5    4    3    2    1    0	
	<i>cst16</i>		0    1    0    1    0    s    p	
	16			1    1

Opcode map field used...	For operand type...	Unit
<i>cst16</i> <i>dst</i>	scst16 sint	.S1, .S2

**Description** The 16-bit constant, *cst16*, is sign extended and placed in *dst*.

The **MVKL** instruction is equivalent to the **MVK** instruction (see [MVK](#)), except that the **MVKL** instruction disables the constant range checking normally performed by the assembler/linker. This allows the **MVKL** instruction to be paired with the **MVKH** instruction (see [MVKH/MVKLH](#)) to generate 32-bit constants.

To load 32-bit constants, such as 1234ABCDh, use the following pair of instructions:

```
MVKL .S1      0x0ABCD, A4
MVKLH .S1     0x1234, A4
```

This could also be used:

```
MVKL .S1      0x1234ABCD, A4
MVKH .S1     0x1234ABCD, A4
```

Use this to load the address of a label:

```
MVKL .S2      label, B5
MVKH .S2     label, B5
```

**Execution** if (cond) scst → dst

```
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single cycle

**Delay Slots** 0

**See Also** [MVK, MVKH/MVKLH](#)

**Examples** [Example 1](#)

```
MVKL .S1 5678h,A8
```

**Before instruction**

A8

xxxx xxxxh

**1 cycle after instruction**

A8

0000 5678h

**Example 2**

```
MVKL .S1 0C678h,A8
```

**Before instruction**

A8

xxxx xxxxh

**1 cycle after instruction**

A8

FFFF C678h

## 4.228 NEG

Negate

**Syntax**    **NEG (.unit) src2, dst**

or

**NEG (.L1 or .L2) src2\_h:src2\_l, dst\_h:dst\_l**

unit = .L1, .L2, .S1, .S2

**Opcode**    .S unit

31	29	28	27	23 22										18	17	16
creg		z	dst					src2					0	0		
3		1		5					5							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	x	0	1	0	1	1	0	1	0	0	0	s	p	
			1											1	1	

Opcode map field used...	For operand type...	Unit
src2	xsint	.S1, S2
dst	sint	

**Opcode**    .L unit

31	29	28	27	23 22										18	17	16
creg		z	dst					src2					0	0		
3		1		5					5							
15	14	13	12	11						5	4	3	2	1	0	
0	0	0	x		op					1	1	0	s	p		
			1						7					1	1	

Opcode map field used...	For operand type...	Unit	Opfield
src2	xsint	.L1, .L2	0000110
dst	sint		
src2	slong	.L1, .L2	0100100
dst	slong		

**Description**    The NEG pseudo-operation negates *src2* and places the result in *dst*. The assembler uses **SUB (.unit) 0, src2, dst** to perform this operation.

**Execution**    if (cond)    0 -s src2 → dst

else nop  
**Instruction Type** Single-cycle  
**Delay Slots** 0

4.229 NOP

## No Operation

**Syntax**      NOP [*count*]

unit = none

## ***Compact Instruction Format***

<b>Unit</b>	<b>Opcode Format</b>	<b>Figure</b>
none	Unop	<a href="#">Figure H-7</a>

## *Opcode*

Opcode map field used...	For operand type...	Unit
<i>src</i>	ucst4	none

**Description** *src* is encoded as *count*-1. For *src*+1 cycles, no operation is performed. The maximum value for *count* is 9. **NOP** with no operand is treated like **NOP 1** with *src* encoded as 0000.

A multicycle **NOP** will not finish if a branch is completed first. For example, if a branch is initiated on cycle  $n$  and a **NOP5** instruction is initiated on cycle  $n + 3$ , the branch is complete on cycle  $n + 6$  and the **NOP** is executed only from cycle  $n + 3$  to cycle  $n + 5$ . A single-cycle **NOP** in parallel with other instructions does not affect operation.

A multicycle **NOP** instruction cannot be paired with any other multicycle **NOP** instruction in the same execute packet. Instructions that generate a multicycle **NOP** are: **ADDKPC**, **BNOP**, **CALLP**, and **IDLE**.

**Execution** No operation for *count* cycles

***Instruction Type***

### *Delay Slots*

### *Examples*      Example 1

NOP MVK .S1 125h,A1

## Before NOP

**1 cycle after NOP  
(No operation executes)**

1 cycle after MVK

A1	12345678h	A1	12345678h	A1	00000125h
----	-----------	----	-----------	----	-----------

### Example 2

```
MVK .S1 1,A1
MVKLH .S1 0,A1
NOP 5
ADD .L1 A1,A2,A1
```

**Before NOP 5**

**1 cycle after ADD instruction  
(6 cycles after NOP 5)**

A1	00000001h	A1	00000004h
A2	00000003h	A2	00000003h

## 4.230 NORM

Normalize Integer

**Syntax**    **NORM (.unit) src2, dst**

or

**NORM (.unit) src2\_h:src2\_l, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27			23	22			18	17	16
				<i>creg</i>		<i>z</i>		<i>dst</i>		<i>src2</i>		0
3				1			5			5		
15	14	13	12	11				5	4	3	2	1
0	0	0	x		<i>op</i>			1	1	0	s	p
			1			7				1		1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	xsint	.L1,.L2	1100011
<i>dst</i>	uint		
<i>src2</i>	slong	.L1,.L2	1100000
<i>dst</i>	uint		

**Description**    The number of redundant sign bits of *src2* is placed in *dst*. Several examples are shown in the following diagram.

In this case, **NORM** returns 0:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

In this case, **NORM** returns 3:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

In this case, **NORM** returns 30:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

In this case, **NORM** returns 31:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

**Execution**      if (cond)    norm(src) → dst  
else nop

**Pipeline**

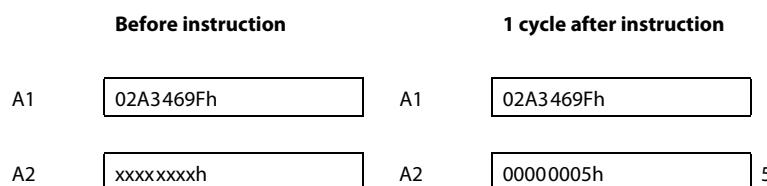
Pipeline Stage	E1
Read	src2
Written	dst
Unit in use	.L

**Instruction Type**   Single-cycle

**Delay Slots**   0

**Examples**   **Example 1**

NORM .L1 A1,A2



**Example 2**

NORM .L1 A1,A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A1	FFFFF25Ah	A1	FFFFF25Ah
A2	xxxxxxxxh	A2	00000013h 19

### Example 3

NORM .L1 A1:A0,A3

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	0000 0007h	A0	0000 0007h
A1	0000 0000h	A1	0000 0000h
A3	xxxx xxxxh	A3	0000 0024h 36

## 4.231 NOT

Bitwise NOT

**Syntax**    **NOT(.unit) src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

31	29	28	27	23 22										18	17	16	
creg	z	dst										src2				1	1
3		1		5								5					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	x	1	1	0	1	1	1	0	1	1	0	s	p		
			1										1	1			

Opcode map field used...	For operand type...	Unit
src2	xuint	.L1, .L2
dst	uint	

**Opcode**    .S unit

31	29	28	27	23 22										18	17	16	
creg	z	dst										src2				1	1
3		1		5								5					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	x	0	0	1	0	1	0	1	0	0	0	s	p		
			1										1	1			

Opcode map field used...	For operand type...	Unit
src2	xuint	.S1, .S2
dst	uint	

**Description**    The **NOT** pseudo-operation performs a bitwise **NOT** on the *src2* operand and places the result in *dst*. The assembler uses **XOR (.unit) -1, src2, dst** to perform this operation.

**Execution**    if (cond)    -1 XOR src2 → dst  
else nop

**Instruction Type**    Single-cycle

**Delay Slots**    0

## 4.232 OR

Bitwise OR

**Syntax**    **OR (.unit) src1, src2, dst**

unit =.D1, .D2, .L1, .L2, .S1, .S2

**Opcode**    Opcode for .S Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opcode	1	0	0	0	s	p	

3                        5                        5                        5                        6

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.S1 or .S2	011011
src1,src2,dst	scst5,xop2,dst	.S1 or .S2	011010

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opcode	1	0	0	0	s	p

5                        5                        5                        6

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	011011

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opcode	1	1	0	s	p	

3                        5                        5                        7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1111111
src1,src2,dst	scst5,xop2,dst	.L1 or .L2	1111110

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0	
0	0	0	1		dst		src2		src1	x		opcode	1	1	0	s	p	

5                        5                        5                        6

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.L1 or .L2	1111111

**Opcode**      Opcode for .D Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	1	0		opcode	1	1	0	0	s	p	

3                    5                    5                    5                    4

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.D1 or .D2	0010
src1,src2,dst	scst5,xop2,dst	.D1 or .D2	0011

**Description**      The OR instruction performs the bit-wise OR between two source registers and stores the result in a third register.

**Execution**      `if(cond) src1 OR src2 -> dst`  
`else nop`

**Instruction Type**      Single cycle

**Delay Slots**      0

**Functional Unit Latency**      1

**See Also**

**Example**      `A0 == 0xfadedb0a`  
`OR .S 7,A0,A15`  
`A15 == 0xfadedb0f`

```
A0 == 0xbeef0000
A1 == 0x0000babe
OR .S A0,A1,A2
A2 == 0xbeefbabe
```

```
A1 == 0xbeef0000
A0 == 0xbeef0000
A3 == 0x0000babe
A2 == 0xfffffbabe
OR .S A1:A0,A3:A2,A9:A8
A9 == 0xbeefbabe
A8 == 0xfffffbabe
```

## 4.233 PACK2

Pack Two 16 LSBs Into Upper and Lower Register Halves

**Syntax** **PACK2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode** .L unit

31	29	28	27							23	22							18	
creg	z	dst						src2											
3		1			5								5						
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0					

src1	x	0	0	0	0	0	0	0	1	1	0	s	p	5	1	1	1	1
5		1												1	1			

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

**Opcode** .S unit

31	29	28	27							23	22							18	
creg	z	dst						src2											
3		1			5								5						
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0					

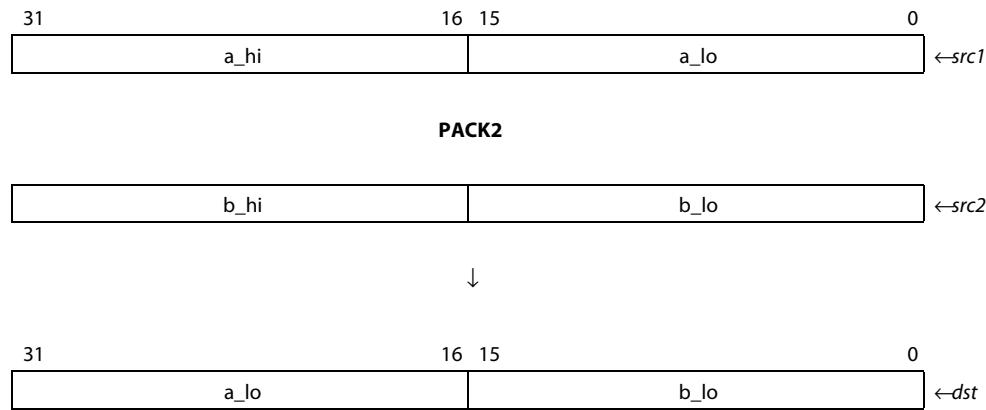
  

src1	x	1	1	1	1	1	1	1	1	1	0	s	p	5	1	1	1	1
5		1												1	1			

Opcode map field used...	For operand type...	Unit
src1	i2	.S1, .S2
src2	xi2	
dst	i2	

**Description** Moves the lower halfwords from *src1* and *src2* and packs them both into *dst*. The lower halfword of *src1* is placed in the upper halfword of *dst*. The lower halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** ([ADD2](#)).



**Execution**

```
if (cond) {
    lsb16(src2) → lsb16(dst);
    lsb16(src1) → msb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

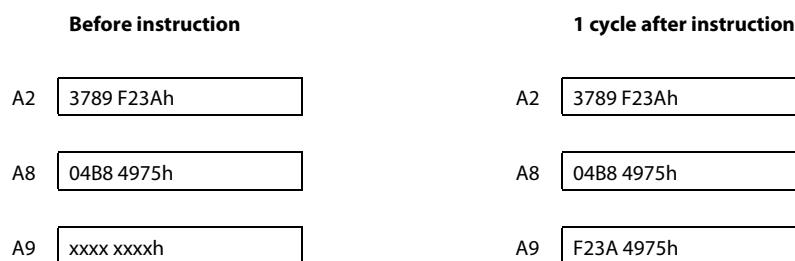
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACKH2](#), [PACKHL2](#), [PACKLH2](#), [SPACK2](#)

**Examples** [Example 1](#)

PACK2 .L1 A2,A8,A9



**Example 2**

PACK2 .S2 B2,B8,B12

**Before instruction****1 cycle after instruction**B2 

0124 2451h
------------

B2 

0124 2451h
------------

B8 

01A6 A051h
------------

B8 

01A6 A051h
------------

B12 

xxxx xxxxh
------------

B12 

2451 A051h
------------

## 4.234 PACKH2

Pack Two 16 MSBs Into Upper and Lower Register Halves

**Syntax** **PACKH2 (.unit) *src1*, *src2*, *dst***

*unit* = .L1, .L2, .S1, .S2

**Opcode** .L unit

31	29	28	27	23 22								18							
<i>creg</i>	<i>z</i>	<i>dst</i>								<i>src2</i>									
3		1		5						5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0					

5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Opcode map field used...	For operand type...	Unit
<i>src1</i>	i2	.L1, .L2
<i>src2</i>	xi2	
<i>dst</i>	i2	

**Opcode** .S unit

31	29	28	27	23 22								18							
<i>creg</i>	<i>z</i>	<i>dst</i>								<i>src2</i>									
3		1		5						5									
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0					

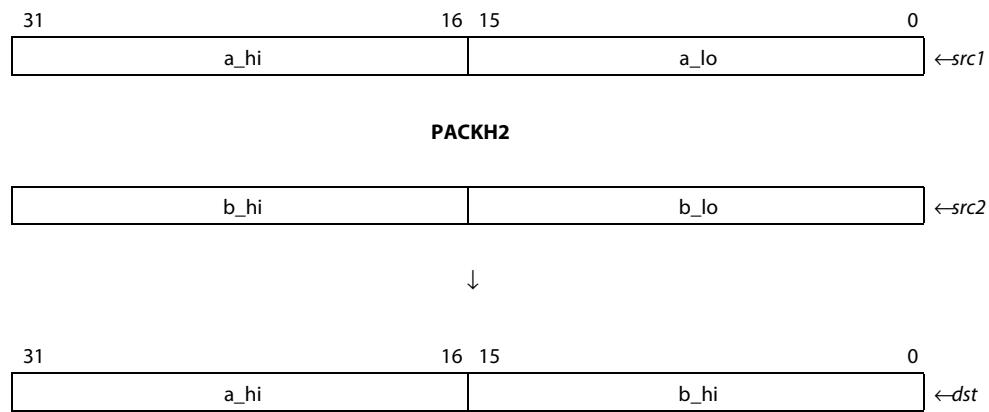
  

5	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Opcode map field used...	For operand type...	Unit
<i>src1</i>	i2	.S1, .S2
<i>src2</i>	xi2	
<i>dst</i>	i2	

**Description** Moves the upper halfwords from *src1* and *src2* and packs them both into *dst*. The upper halfword of *src1* is placed in the upper half-word of *dst*. The upper halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see [ADD2](#)).



**Execution**

```

if (cond) {
    msb16(src2) → lsb16(dst);
    msb16(src1) → msb16(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L, .S

**Instruction Type** Single-cycle

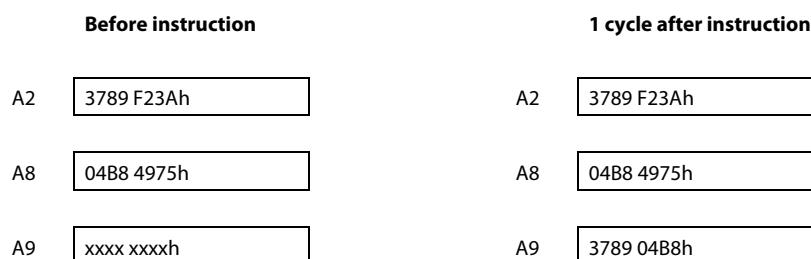
**Delay Slots** 0

**See Also** [PACK2](#), [PACKHL2](#), [PACKLH2](#), [SPACK2](#)

**Examples**

**Example 1**

PACKH2 .L1 A2,A8,A9



**Example 2**

PACKH2 .S2 B2,B8,B12

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	0124 01A6h

## 4.235 PACKH4

Pack Four High Bytes Into Four 8-Bit Halfwords

**Syntax** **PACKH4 (.unit) *src1*, *src2*, *dst***

*unit* = .L1 or .L2

**Opcode**

31 29 28 27		23 22		18	
<i>creg</i>	<i>z</i>	<i>dst</i>			<i>src2</i>
3	1	5		5	
17	13 12	11 10 9 8 7 6 5 4 3 2 1 0			
<i>src1</i>	<i>x</i>	1 1 0 1 0 0 1 1 1 0 1 0 <i>s</i> <i>p</i>			1 1
5	1				

Opcode map field used...	For operand type...	Unit
<i>src1</i>	i4	.L1, .L2
<i>src2</i>	xi4	
<i>dst</i>	i4	

**Description**

Moves the high bytes of the two halfwords in *src1* and *src2*, and packs them into *dst*. The bytes from *src1* are packed into the most-significant bytes of *dst*, and the bytes from *src2* are packed into the least-significant bytes of *dst*.

- The high byte of the upper halfword of *src1* is moved to the upper byte of the upper halfword of *dst*. The high byte of the lower halfword of *src1* is moved to the lower byte of the upper halfword of *dst*.
- The high byte of the upper halfword of *src2* is moved to the upper byte of the lower halfword of *dst*. The high byte of the lower halfword of *src2* is moved to the lower byte of the lower halfword of *dst*.

31	24 23	16 15	8 7	0	← <i>src1</i>
a_3	a_2	a_1	a_0		

**PACKH4**

b_3	b_2	b_1	b_0	← <i>src2</i>
-----	-----	-----	-----	---------------

↓

31	24 23	16 15	8 7	0	← <i>dst</i>
a_3	a_1	b_3	b_1		

**Execution** if (cond) {

```
byte3(src1) → byte3(dst);
byte1(src1) → byte2(dst);
byte3(src2) → byte1(dst);
byte1(src2) → byte0(dst);
```

```

        }
else nop

```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACKL4](#), [SPACKU4](#)

**Examples** [Example 1](#)

PACKH4 .L1 A2,A8,A9

**Before instruction**

A2	37 89 F2 3Ah
A8	04 B8 49 75h
A9	xxxx xxxxh

**1 cycle after instruction**

A2	37 89 F2 3Ah
A8	04 B8 49 75h
A9	37 F2 04 49h

**Example 2**

PACKH4 .L2 B2,B8,B12

**Before instruction**

B2	01 24 24 51h
B8	01 A6 A0 51h
B12	xxxx xxxxh

**1 cycle after instruction**

B2	01 24 24 51h
B8	01 A6 A0 51h
B12	01 24 01 A0h

## 4.236 PACKHL2

Pack 16 MSB Into Upper and 16 LSB Into Lower Register Halves

**Syntax**    **PACKHL2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

31	29	28	27											23	22	18
creg	z	dst										src2				
3		1			5							5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

src1	x	0	0	1	1	1	0	0	1	1	0	s	p	5	1	1
5		1												1		1

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

**Opcode**    .S unit

31	29	28	27											23	22	18
creg	z	dst										src2				
3		1			5							5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		

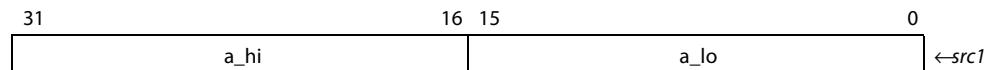
  

src1	x	0	0	1	0	0	0	0	1	0	0	0	s	p	5	1	1
5		1													1		1

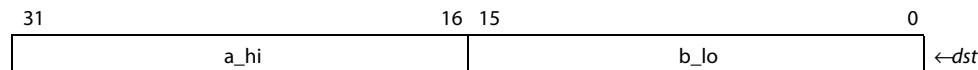
Opcode map field used...	For operand type...	Unit
src1	i2	.S1, .S2
src2	xi2	
dst	i2	

**Description**    Moves the upper halfword from *src1* and the lower halfword from *src2* and packs them both into *dst*. The upper halfword of *src1* is placed in the upper halfword of *dst*. The lower halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see [ADD2](#)).

**PACKHL2**

↓



**Execution**

```
if (cond) {
    lsb16(src2) → lsb16(dst);
    msb16(src1) → msb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L, .S

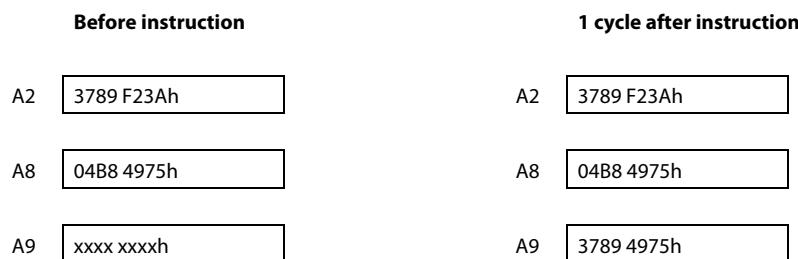
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACK2](#), [PACKH2](#), [PACKLH2](#), [SPACK2](#)

**Examples** [Example 1](#)

PACKHL2 .L1 A2,A8,A9



**Example 2**

PACKHL2 .S2 B2,B8,B12

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	0124 A051h

## 4.237 PACKLH2

Pack 16 LSB Into Upper and 16 MSB Into Lower Register Halves

**Syntax**    **PACKLH2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

31	29	28	27	dst										23	22	18
creg	z													src2		
3		1			5									5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	0	1	1	0	1	1	1	1	1	0	s	p		
5		1											1	1		

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

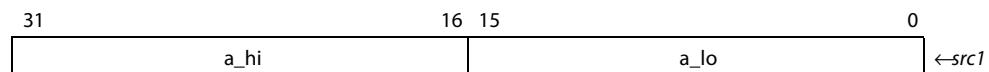
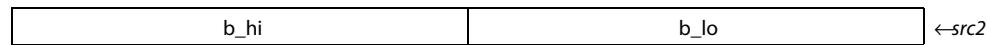
**Opcode**    .S unit

31	29	28	27	dst										23	22	18
creg	z													src2		
3		1			5									5		
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
src1	x	0	1	0	0	0	0	0	1	0	0	0	s	p		
5		1											1	1		

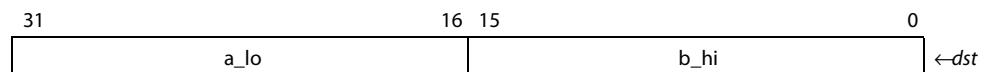
Opcode map field used...	For operand type...	Unit
src1	i2	.S1, .S2
src2	xi2	
dst	i2	

**Description**    Moves the lower halfword from *src1*, and the upper halfword from *src2*, and packs them both into *dst*. The lower halfword of *src1* is placed in the upper halfword of *dst*. The upper halfword of *src2* is placed in the lower halfword of *dst*.

This instruction is useful for manipulating and preparing pairs of 16-bit values to be used by the packed arithmetic operations, such as **ADD2** (see [ADD2](#)).

**PACKLH2**

↓



**Execution**

```
if (cond) {
    msb16(src2) → lsb16(dst);
    lsb16(src1) → msb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

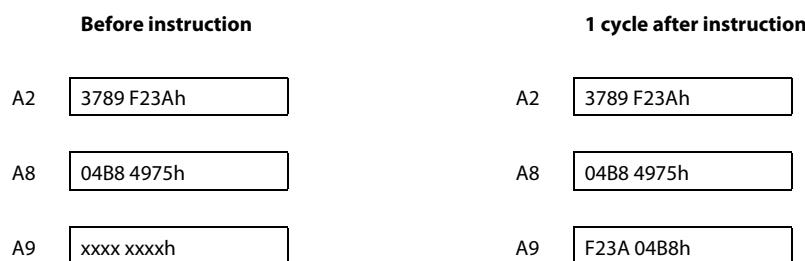
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACK2](#), [PACKH2](#), [PACKHL2](#), [SPACK2](#)

**Examples** [Example 1](#)

PACKLH2 .L1 A2,A8,A9



**Example 2**

PACKLH2 .S2 B2,B8,B12

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	2451 01A6h

## 4.238 PACKL4

Pack Four Low Bytes Into Four 8-Bit Halfwords

**Syntax** **PACKL4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	dst					src2					18
creg	z													
3		1		5					5					
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
src1	x	1	1	0	1	0	0	0	1	1	0	s	p	
5		1										1	1	

Opcode map field used...	For operand type...	Unit
src1	i4	.L1, .L2
src2	xi4	
dst	i4	

**Description** Moves the low bytes of the two halfwords in *src1* and *src2*, and packs them into *dst*. The bytes from *src1* are packed into the most-significant bytes of *dst*, and the bytes from *src2* are packed into the least-significant bytes of *dst*.

- The low byte of the upper halfword of *src1* is moved to the upper byte of the upper halfword of *dst*. The low byte of the lower halfword of *src1* is moved to the lower byte of the upper halfword of *dst*.
- The low byte of the upper halfword of *src2* is moved to the upper byte of the lower halfword of *dst*. The low byte of the lower halfword of *src2* is moved to the lower byte of the lower halfword of *dst*.

31	24 23	16 15	8 7	0	←src1
a_3		a_2	a_1	a_0	

**PACKL4**

b_3	b_2	b_1	b_0	←src2
-----	-----	-----	-----	-------

↓

31	24 23	16 15	8 7	0	←dst
a_2		a_0	b_2	b_0	

**Execution** if (cond) {

```
byte2(src1) → byte3(dst);
byte0(src1) → byte2(dst);
byte2(src2) → byte1(dst);
byte0(src2) → byte0(dst);
```

```

        }
else nop
}
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACKH4](#), [SPACKU4](#)

**Examples** [Example 1](#)

```
PACKL4 .L1 A2,A8,A9
```

**Before instruction**

A2	37 89 F2 3Ah
A8	04 B8 49 75h
A9	xxxx xxxxh

**1 cycle after instruction**

A2	37 89 F2 3Ah
A8	04 B8 49 75h
A9	89 3A B8 75h

**Example 2**

```
PACKL4 .L2 B2,B8,B12
```

**Before instruction**

B2	01 24 24 51h
B8	01 A6 A0 51h
B12	xxxx xxxxh

**1 cycle after instruction**

B2	01 24 24 51h
B8	01 A6 A0 51h
B12	24 51 A6 51h

## 4.239 QMPY32

4-Way SIMD Multiply, Packed Signed 32-bit

**Syntax**    **QMPY32 (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opfield	0	0	0	0	0	s	p

5                         5                         5                         5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	qwop1,qwop2,qwdst	.M1 or .M2	10000

**Description**    This is a 4x SIMD 32 by 32 multiplier operation where the output is the lower 32 bits of the result.

**Execution**  
 $\text{src1}_0 \times \text{src2}_0 \rightarrow \text{dst}_0$   
 $\text{src1}_1 \times \text{src2}_1 \rightarrow \text{dst}_1$   
 $\text{src1}_2 \times \text{src2}_2 \rightarrow \text{dst}_2$   
 $\text{src1}_3 \times \text{src2}_3 \rightarrow \text{dst}_3$

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**

---

**Example**

```
A3 == 0x80000000
A2 == 0x80000000
A1 == 0x7FFFFFFF
A0 == 0xFFFFFFFF

A11 == 0xFFFFFFFF
A10 == 0x80000000
A9 == 0x7FFFFFFF
A8 == 0xFFFFFFFF
QMPY32 .M .....
A15 == 0x80000000
A14 == 0x00000000
A13 == 0x00000001
A12 == 0x00000001
```

## 4.240 QMPYSP

4-Way SIMD Floating Point Multiply, Packed Single-Precision Floating Point

**Syntax**    **QMPYSP (.unit) src1, src2, dst**

unit = .M1 or .M2

**Opcode**    Opcode for .M Unit, 32-bit, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	7	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opfield	0	0	0	0	0	s	p

5                         5                         5                         5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	qwop1,qwop2,qwdst	.M1 or .M2	11101

**Description**    The QMPYSP instruction performs four floating point multiplies of the four pairs of single precision floating point values contained in *qwop1* and *qwop2*. This instruction cannot use the crosspath.

### Special Cases:

1. If one source is SNaN or QNaN, the result is a signed NaN\_out and the NANn bit is set. If either source is SNaN, the INVAL bit is set also. The sign of NaN\_out is the XOR to the input signs.
2. Signed infinity multiplied by signed infinity or a normalized number (other than signed zero) returns signed infinity. Signed infinity multiplied by signed zero (or denormal) returns a signed NaN\_out and sets the INVAL bit.
3. If one or both source are signed zero, the result is signed zero unless the other source is a NaN or signed infinity, in which case the result is signed NaN\_out.
4. If signed zero is multiplied by signed infinity, the result is signed NaN\_out and the INVAL bit is set.
5. A denormalized source is treated as signed zero and the DENn bit is set. The INEX bit is set except when the other source is signed infinity, signed NaN, OR signed zero. Therefore, a signed infinity multiplied by a denormalized number gives a signed NaN\_out and sets the INVAL bit.
6. If rounding is performed, the INEX bit is set.

**Execution**    `src1_0 x src2_0 -> dst_0`  
`src1_1 x src2_1 -> dst_1`  
`src1_2 x src2_2 -> dst_2`  
`src1_3 x src2_3 -> dst_3`

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [MPYDP](#)

---

**Example**

```
FA3 == 0x80000000
A2 == 0x80000000
A1 == 0x7FFFFFFF
A0 == 0xFFFFFFFF

A11 == 0xFFFFFFFF
A10 == 0x80000000
A9 == 0x7FFFFFFF
A8 == 0xFFFFFFFF
QMPY32 .M .....
A15 == 0x80000000
A14 == 0x00000000
A13 == 0x00000001
A12 == 0x00000001
```

## 4.241 QSMPY32R1

4-Way SIMD Multiply with Saturation and Rounding, Packed Signed 32-bit

**Syntax** **QSMPY32R1** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode** Opcode for .M Unit, Compound Results, new opcode space

31	30	29	28	27	23	22	18	17	13	12	11	10	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x	0	opfield	1	1	0	0	s	p	

5                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	qwop1,qwop2,qwdst	.M1 or .M2	11010

**Description** This is a 4x SIMD fractional 32 by 32 multiplier operation where the output is kept at 32 bits precision. This instruction is the same as MPY32 except the result is shifted by 31 bits to the right and rounded. This normalizes the result to lie within -1 and 1 in a Q31 fractional number system. The case where the inputs are maximum negative requires saturation otherwise the result will overflow.

**Execution**

```
(src1_0 x src2_0 + (1<<30))>>31 -> dst_0
(src1_1 x src2_1 + (1<<30))>>31 -> dst_1
(src1_2 x src2_2 + (1<<30))>>31 -> dst_2
(src1_3 x src2_3 + (1<<30))>>31 -> dst_3
```

**Instruction Type** 4-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also**

**Example**

```

CSR == 0x00000000 ; 4
A7 == 0x00000001
A6 == 0x00000001
A5 == 0x00000001
A4 == 0x00000001
A11 == 0x00000001
A10 == 0x00000001
A9 == 0x00000001
A8 == 0x00000001
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x00000000
A2 == 0x00000000
A1 == 0x00000000
A0 == 0x00000000

CSR= 0x00000000 ; 4
;[1]*[1]->[0];
CSR == 0x00000000 ; 4
A7 == 0x00000001
A6 == 0x00000001
A5 == 0x00000001
A4 == 0x00000001
A11 == 0x3FFFFFFF
A10 == 0x3FFFFFFF
A9 == 0x3FFFFFFF
A8 == 0x3FFFFFFF
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x00000000
A2 == 0x00000000
A1 == 0x00000000
A0 == 0x00000000

CSR= 0x00000000 ; 4
;[1]*[2^30-1]->[0];
CSR == 0x00000000 ; 4
A7 == 0x00000001
A6 == 0x00000001
A5 == 0x00000001
A4 == 0x00000001
A11 == 0x40000000
A10 == 0x40000000
A9 == 0x40000000
A8 == 0x40000000
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x00000001
A2 == 0x00000001
A1 == 0x00000001
A0 == 0x00000001

CSR= 0x00000000 ; 4
;[1]*[2^30]->[1];
CSR == 0x00000000 ; 4
A7 == 0x19680828
A6 == 0x19680828
A5 == 0x19680828
A4 == 0x19680828
A11 == 0x19700520
A10 == 0x19700520
A9 == 0x19700520
A8 == 0x19700520
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x050C8DA3
A2 == 0x050C8DA3
A1 == 0x050C8DA3
A0 == 0x050C8DA3

CSR= 0x00000000 ; 4
;[426248232]*[426771744]->[84708771];
CSR == 0x00000000 ; 4
A7 == 0x19680828
A6 == 0x19680828
A5 == 0x19680828
A4 == 0x19680828
A11 == 0xE68FFFAE0
A10 == 0xE68FFFAE0
A9 == 0xE68FFFAE0
A8 == 0xE68FFFAE0
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0xFAF3725D
A2 == 0xFAF3725D
A1 == 0xFAF3725D

```

```

A0 == 0xFAF3725D

CSR= 0x00000000 ; 4
; [426248232]*[-426771744]->[-84708771] ;
CSR == 0x00000000 ; 4
A7 == 0xE697F7D8
A6 == 0xE697F7D8
A5 == 0xE697F7D8
A4 == 0xE697F7D8
A11 == 0xE68FFAE0
A10 == 0xE68FFAE0
A9 == 0xE68FFAE0
A8 == 0xE68FFAE0
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x050C8DA3
A2 == 0x050C8DA3
A1 == 0x050C8DA3
A0 == 0x050C8DA3

CSR= 0x00000000 ; 4
; [-426248232]*[-426771744]->[84708771] ;
CSR == 0x00000000 ; 4
A7 == 0x80000000
A6 == 0x80000000
A5 == 0x80000000
A4 == 0x80000000
A11 == 0x80000000
A10 == 0x80000000
A9 == 0x80000000
A8 == 0x80000000
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x7FFFFFFF
A2 == 0x7FFFFFFF
A1 == 0x7FFFFFFF
A0 == 0x7FFFFFFF

CSR= 0x00000200 ; 4
; [-2^31]*[-2^31]=[2^31-1] ;
CSR == 0x00000000 ; 4
A7 == 0x80000000
A6 == 0x80000000
A5 == 0x80000000
A4 == 0x80000000
A11 == 0x80000001
A10 == 0x80000001
A9 == 0x80000001
A8 == 0x80000001
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x7FFFFFFF
A2 == 0x7FFFFFFF
A1 == 0x7FFFFFFF
A0 == 0x7FFFFFFF

CSR= 0x00000000 ; 4
; [-2^31]*[-2^31+1]=[2^31-1] ;
CSR == 0x00000000 ; 4
A7 == 0x80000001
A6 == 0x80000001
A5 == 0x80000001
A4 == 0x80000001
A11 == 0x80000001
A10 == 0x80000001
A9 == 0x80000001
A8 == 0x80000001
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x7FFFFFFE
A2 == 0x7FFFFFFE
A1 == 0x7FFFFFFE
A0 == 0x7FFFFFFE

CSR= 0x00000000 ; 4
; [-2^31+1]*[-2^31+1]=[2^31-2] ;
CSR == 0x00000000 ; 4
A7 == 0x80000000
A6 == 0x80000000
A5 == 0x80000000
A4 == 0x80000000
A11 == 0x00000000
A10 == 0x00000000
A9 == 0x00000000
A8 == 0x00000000

```

```

QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x00000000
A2 == 0x00000000
A1 == 0x00000000
A0 == 0x00000000

CSR= 0x00000000 ; 4
;[-2^31]*[0]=[0];
CSR == 0x00000000 ; 4
B7 == 0x80000000
B6 == 0x80000000
B5 == 0x80000000
B4 == 0x80000000
B11 == 0x80000000
B10 == 0x80000000
B9 == 0x80000000
B8 == 0x80000000
QSMPY32R1 .M B7:B6:B5:B4,B11:B10:B9:B8,B3:B2:B1:B0
B3 == 0x7FFFFFFF
B2 == 0x7FFFFFFF
B1 == 0x7FFFFFFF
B0 == 0x7FFFFFFF

CSR= 0x00000200 ; 4
;[-2^31]*[-2^31]=[2^31-1];
A7 == 0x80000000
A6 == 0x80000000
A5 == 0x80000000
A4 == 0x80000000
A11 == 0xfffffffff
A10 == 0xfffffffff
A9 == 0xfffffffff
A8 == 0xfffffffff
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x00000001
A2 == 0x00000001
A1 == 0x00000001
A0 == 0x00000001

A7 == 0xfffffffff
A6 == 0xfffffffff
A5 == 0xfffffffff
A4 == 0xfffffffff
A11 == 0x80000000
A10 == 0x80000000
A9 == 0x80000000
A8 == 0x80000000
QSMPY32R1 .M A7:A6:A5:A4,A11:A10:A9:A8,A3:A2:A1:A0
A3 == 0x00000001
A2 == 0x00000001
A1 == 0x00000001
A0 == 0x00000001

B7 == 0x80000000
B6 == 0x80000000
B5 == 0x80000000
B4 == 0x80000000
B11 == 0xfffffffff
B10 == 0xfffffffff
B9 == 0xfffffffff
B8 == 0xfffffffff
QSMPY32R1 .M B7:B6:B5:B4,B11:B10:B9:B8,B3:B2:B1:B0
B3 == 0x00000001
B2 == 0x00000001
B1 == 0x00000001
B0 == 0x00000001

B7 == 0xfffffffff
B6 == 0xfffffffff
B5 == 0xfffffffff
B4 == 0xfffffffff
B11 == 0x80000000
B10 == 0x80000000
B9 == 0x80000000
B8 == 0x80000000
QSMPY32R1 .M B7:B6:B5:B4,B11:B10:B9:B8,B3:B2:B1:B0
B3 == 0x00000001
B2 == 0x00000001
B1 == 0x00000001
B0 == 0x00000001

```

## 4.242 RCPDP

Double-Precision Floating-Point Reciprocal Approximation

**Syntax**    **RCPDP** (.unit) *src2*, *dst*

unit = .S1, .S2

**Opcode**

31	29	28	27	23 22						18				
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>				
3		1		5						5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	x	1	0	1	1	0	1	1	0	0	0	0	s	p
5		1											1	1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	dp	.S1, .S2
<i>dst</i>	dp	

**Description**

The 64-bit double-precision floating-point reciprocal approximation value of *src2* is placed in *dst*. The operand is read in one cycle by using the *src1* port for the 32 LSBs and the *src2* port for the 32 MSBs.

The **RCPDP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than  $2^{-8}$ ). This estimate can be used as a seed value for an algorithm to compute the reciprocal to greater accuracy.

The Newton-Raphson algorithm can further extend the mantissa's precision:

$$x[n + 1] = x[n](2 - v \times x[n])$$

where *v* = the number whose reciprocal is to be found.

*x[0]*, the seed value for the algorithm, is given by **RCPDP**. For each iteration, the accuracy doubles. Thus, with one iteration, accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is 32 bits; with the third iteration, the accuracy is the full 52 bits.



**Note—**

- 1) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 2) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 3) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INFO, OVER, INEX, and DEN2 bits are set.
- 4) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set.
- 5) If *src2* is signed infinity, signed 0 is placed in *dst*.

- 6) If the result underflows, signed 0 is placed in *dst* and the INEX and UNDER bits are set. Underflow occurs when  $2^{1022} < src2 <$  infinity.

**Execution**

```
if (cond) rcp(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2_l, src2_h</i>	
Written	<i>dst_l</i>	<i>dst_h</i>
Unit in use	.S	

If *dst* is used as the source for the ADDDP, CMPEQDP, CMPLTDP, CMPGTDP, MPYDP, or SUBDP instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type** 2-cycle DP

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [RCPSP](#), [RSQRDP](#)

**Example** RCPDP .S1 A1:A0,A3:A2

**Before instruction**

A1:A0	4010 0000h	0000 0000h
A3:A2	xxxx xxxxh	xxxx xxxxh

**2 cycles after instruction**

A1:A0	4010 0000h	0000 0000h	4.00
A3:A2	3FD0 0000h	0000 0000h	0.25

## 4.243 RCPSP

Single-Precision Floating-Point Reciprocal Approximation

**Syntax**    **RCPSP (.unit) src2, dst**

unit = .S1, .S2

**Opcode**

31	29	28	27	dst						src2						18	17	16
																0	0	
3				1				5							5			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	0	0	x	1	1	1	1	0	1	1	0	0	0	s	p	1	1	
				1														

Opcode map field used...	For operand type...	Unit
src2	xsp	.S1, .S2
dst	sp	

**Description**

The single-precision floating-point reciprocal approximation value of *src2* is placed in *dst*.

The **RCPSP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than  $2^{-8}$ ). This estimate can be used as a seed value for an algorithm to compute the reciprocal to greater accuracy.

The Newton-Raphson algorithm can further extend the mantissa's precision:

$$x[n + 1] = x[n](2 - v \times x[n])$$

where *v* = the number whose reciprocal is to be found.

*x[0]*, the seed value for the algorithm, is given by **RCPSP**. For each iteration, the accuracy doubles. Thus, with one iteration, accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is the full 23 bits.



**Note—**

- 1) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 2) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 3) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INFO, OVER, INEX, and DEN2 bits are set.
- 4) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set.
- 5) If *src2* is signed infinity, signed 0 is placed in *dst*.
- 6) If the result underflows, signed 0 is placed in *dst* and the INEX and UNDER bits are set. Underflow occurs when  $2^{126} < src2 < \text{infinity}$ .

**Execution**    if (cond)    rcp(src2) → dst

```
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	. <i>S</i>

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [RCPDP](#), [RSQRSP](#)

**Example** RCPSP .S1 A1,A2

**Before instruction**

A1	<span style="border: 1px solid black; padding: 2px;">4080 0000h</span>
----	--

A2	<span style="border: 1px solid black; padding: 2px;">xxxx xxxxh</span>
----	--

**1 cycle after instruction**

A1	<span style="border: 1px solid black; padding: 2px;">4080 0000h</span>	4.00
----	--	------

A2	<span style="border: 1px solid black; padding: 2px;">3E80 0000h</span>	0.25
----	--	------

## 4.244 RINT

Restore Previous Enable State

**Syntax**    **RINT**

unit = none

**Compatibility**

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	s	p

1      1

**Description**

Copies the contents of the SGIE bit in TSR into the GIE bit in TSR and CSR, and clears the SGIE bit in TSR. The value of the SGIE bit in TSR is used for the current cycle as the GIE indication; if restoring the GIE bit to 1, interrupts are enabled and can be taken after the E1 phase containing the **RINT** instruction.

The CPU may service a maskable interrupt in the cycle immediately following the **RINT** instruction. See section 5.2 for details.

The **RINT** instruction cannot be placed in parallel with: **MVC reg, TSR; MVC reg, CSR; B IRP; B NRP; NOP n; DINT; SPKERNEL; SPKERNELR; SPLOOP; SPLOOPD; SPLOOPW; SPMASK; or SPMASKR.**

This instruction executes unconditionally and cannot be predicated.



**Note**—The use of the **DINT** and **RINT** instructions in a nested manner, like the following code:

DINT

DINT

RINT

RINT

leaves interrupts disabled. The first **DINT** leaves TSR.GIE cleared to 0, so the second **DINT** leaves TSR.SGIE cleared to 0. The **RINT** instructions, therefore, copy zero to TSR.GIE (leaving interrupts disabled).

**Execution**

Enable interrupts in current cycle

SGIE bit in TSR → GIE bit in TSR

SGIE bit in TSR → GIE bit in CSR

0 → SGIE bit in TSR

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**

[DINT](#)

## 4.245 ROTL

Rotate Left

**Syntax**    **ROTL (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

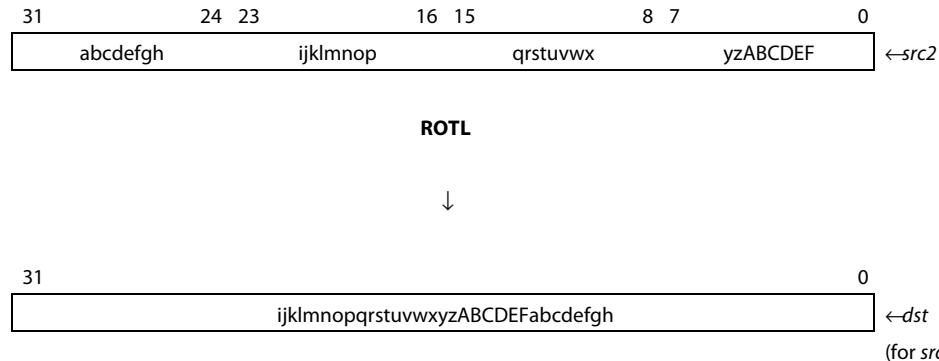
31		29	28	27	23 22		18			
creg	z	dst			src2					
3	1	5			5					
17	13	12	11	10	6	5	4	3		
src1	x	0	op		1	1	0	0		
5	1		5				1	1		

Opcode map field used...	For operand type...	Unit	Opfield
src1	uint	.M1, .M2	11101
src2	xuint		
dst	uint		
src1	ucst5	.M1, .M2	11110
src2	xuint		
dst	uint		

**Description**

Rotates the 32-bit value of *src2* to the left, and places the result in *dst*. The number of bits to rotate is given in the 5 least-significant bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero.

In the following figure, *src1* is equal to 8.



**Note**—The ROTL instruction is useful in cryptographic applications.

**Execution**    if (cond)    (*src2* << *src1*) | (*src2* >> (32 - *src1*)) → *dst*

---

else nop

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [SHL](#), [SHLMB](#), [SHRMB](#), [SHR](#), [SHRU](#)
**Examples** [Example 1](#)

ROTL .M2 B2,B4,B5

<b>Before instruction</b>		<b>2 cycles after instruction</b>	
B2	A6E2 C179h	B2	A6E2 C179h
B4	1458 3B69h	B4	1458 3B69h
B5	xxxx xxxxh	B5	C582 F34Dh

[Example 2](#)

ROTL .M1 A4,10h,A5

<b>Before instruction</b>		<b>2 cycles after instruction</b>	
A4	187A 65FCh	A4	187A 65FCh
A5	xxxx xxxxh	A5	65FC 187Ah

## 4.246 RPACK2

Shift With Saturation and Pack Two 16 MSBs Into Upper and Lower Register Halves

**Syntax**    **RPACK2 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Compatibility**

**Opcode**

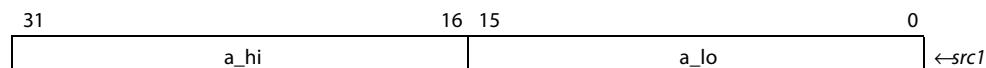
31	30	29	28	27			23	22											18
0	0	0	1		<i>dst</i>				<i>src2</i>										
					5					5									
17		13	12	11	10	9	8	7	6	5	4	3	2	1	0				
	<i>src1</i>		x	1	1	1	0	1	1	1	1	0	0	s	p				
			5	1										1	1				

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sint	.S1, .S2
<i>src2</i>	xsint	
<i>dst</i>	s2	

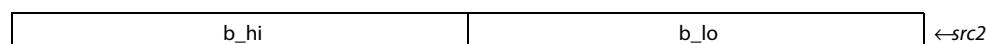
**Description**    *src1* and *src2* are shifted left by 1 with saturation. The 16 most-significant bits of the shifted *src1* value are placed in the 16 most-significant bits of *dst*. The 16 most-significant bits of the shifted *src2* value are placed in the 16 least-significant bits of *dst*.

If either value saturates, the S1 or S2 bit in SSR and the SAT bit in CSR are written one cycle after the result is written to *dst*.

This instruction executes unconditionally and cannot be predicated.

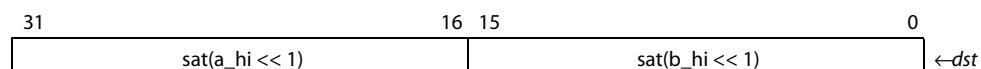


**RPACK2**



↓

↓



**Execution**     $\text{msb16}(\text{sat}(\text{src1} << 1)) \rightarrow \text{msb16}(\text{dst})$

`msb16(sat(src2 << 1)) → lsb16(dst)`

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACK2](#), [PACKH2](#), [SPACK2](#)

**Examples** [Example 1](#)

`RPACK2 .S1 A0,A1,A2`

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	FEDCBA98h	A2	FDB92468h
A1	12345678h		
CSR	00010100h	CSR <sup>1</sup>	00010100h
SSR	00000000h	SSR <sup>1</sup>	00000000h

1. CSR.SAT and SSR.S1 unchanged by operation

### Example 2

`RPACK2 .S2X B0,A1,B2`

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B0	87654321h	B2	80002468h
A1	12345678h		
CSR	00010100h	CSR <sup>1</sup>	00010300h
SSR	00000000h	SSR <sup>1</sup>	00000008h

1. CSR.SAT and SSR.S2 set to 1, 2 cycles after instruction

## 4.247 RSQRDP

Double-Precision Floating-Point Square-Root Reciprocal Approximation

**Syntax**    **RSQRDP** (.unit) *src2*, *dst*

unit = .S1, .S2

**Opcode**

31	29	28	27	23 22						18				
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>				
3		1		5						5				
17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	x	1	0	1	1	1	1	0	1	0	0	0	s	p
5		1										1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	dp	.S1, .S2
<i>dst</i>	dp	

**Description**

The 64-bit double-precision floating-point square-root reciprocal approximation value of *src2* is placed in *dst*. The operand is read in one cycle by using the *src1* port for the 32 LSBs and the *src2* port for the 32 MSBs.

The **RSQRDP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than  $2^{-8}$ ). This estimate can be used as a seed value for an algorithm to compute the reciprocal square root to greater accuracy.

The Newton-Raphson algorithm can further extend the mantissa's precision:

$$x[n+1] = x[n](1.5 - (v/2) \times x[n] \times x[n])$$

where *v* = the number whose reciprocal square root is to be found.

*x[0]*, the seed value for the algorithm is given by **RSQRDP**. For each iteration the accuracy doubles. Thus, with one iteration, the accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is 32 bits; with the third iteration, the accuracy is the full 52 bits.



**Note—**

- 1) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 2) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 3) If *src2* is a negative, nonzero, nondenormalized number, NaN\_out is placed in *dst* and the INVAL bit is set.
- 4) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INEX, and DEN2 bits are set.
- 5) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set. The Newton-Raphson approximation cannot be used to calculate the square root of 0 because infinity multiplied by 0 is invalid.

6) If *src2* is positive infinity, positive 0 is placed in *dst*.

**Execution**

```
if (cond)    sqrcp(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2_l, src2_h</i>	
Written	<i>dst_l</i>	<i>dst_h</i>
Unit in use	.S	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type** 2-cycle DP

**Delay Slots** 1

**Functional Unit Latency** 1

**See Also** [RCPDP](#), [RSQRSP](#)

**Example** RSQRDP .S1 A1:A0,A3:A2

**Before instruction**

A1:A0	4010 0000h	0000 0000h
A3:A2	xxxx xxxxh	xxxx xxxxh

**2 cycles after instruction**

A1:A0	4010 0000h	0000 0000h	4.0
A3:A2	3FE0 0000h	0000 0000h	0.5

## 4.248 RSQRSP

Single-Precision Floating-Point Square-Root Reciprocal Approximation

**Syntax**    **RSQRSP** (.unit) *src2*, *dst*

unit = .S1, .S2

**Opcode**

31	29	28	27	23 22						18	17	16			
				<i>dst</i>						<i>src2</i>		0	0		
3				5						5					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	x	1	1	1	1	1	0	1	0	0	0	s	p
				1									1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xsp	.S1, .S2
<i>dst</i>	sp	

**Description**

The single-precision floating-point square-root reciprocal approximation value of *src2* is placed in *dst*.

The **RSQRSP** instruction provides the correct exponent, and the mantissa is accurate to the eighth binary position (therefore, mantissa error is less than  $2^{-8}$ ). This estimate can be used as a seed value for an algorithm to compute the reciprocal square root to greater accuracy.

The Newton-Raphson algorithm can further extend the mantissa's precision:

$$x[n+1] = x[n](1.5 - (v/2) \times x[n] \times x[n])$$

where v = the number whose reciprocal square root is to be found.

x[0], the seed value for the algorithm, is given by **RSQRSP**. For each iteration, the accuracy doubles. Thus, with one iteration, accuracy is 16 bits in the mantissa; with the second iteration, the accuracy is the full 23 bits.



**Note—**

- 1) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 2) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 3) If *src2* is a negative, nonzero, nondenormalized number, NaN\_out is placed in *dst* and the INVAL bit is set.
- 4) If *src2* is a signed denormalized number, signed infinity is placed in *dst* and the DIV0, INEX, and DEN2 bits are set.
- 5) If *src2* is signed 0, signed infinity is placed in *dst* and the DIV0 and INFO bits are set. The Newton-Raphson approximation cannot be used to calculate the square root of 0 because infinity multiplied by 0 is invalid.

6) If *src2* is positive infinity, positive 0 is placed in *dst*.

**Execution**

```
if (cond)  sqrcp(src2) → dst
else      nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [RCPSP](#), [RSQRDP](#)

**Examples** [Example 1](#)

RSQRSP .S1 A1,A2

Before instruction		1 cycle after instruction	
A1	4080 0000h	A1	4080 0000h 4.0
A2	xxxx xxxxh	A2	3F00 0000h 0.5

## Example 2

RSQRSP .S2X A1,B2

Before instruction		1 cycle after instruction	
A1	4109 999Ah	A1	4109 999Ah 8.6
B2	xxxx xxxxh	B2	3EAE 8000h 0.34082031

## 4.249 SADD

Add Two Signed Integers With Saturation

**Syntax**    **SADD (.unit) src1, src2, dst**

or

**SADD (.L1 or .L2) src1, src2\_h:src2\_l, dst\_h:dst\_l**

unit = .L1, .L2, .S1, .S2

**Compact Instruction Format**

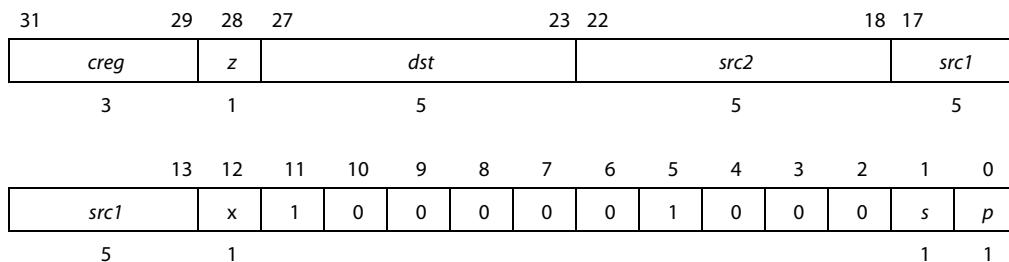
Unit	Opcode Format	Figure
.L	L3	<a href="#">Figure D-4</a>
.S	S3	<a href="#">Figure F-17</a>

**Opcode**    .L unit



Opcode map field used...	For operand type...	Unit	Opfield
src1	sint	.L1, .L2	0010011
src2	xsint		
dst	sint		
src1	xsint	.L1, .L2	0110001
src2	slong		
dst	slong		
src1	scst5	.L1, .L2	0010010
src2	xsint		
dst	sint		
src1	scst5	.L1, .L2	0110000
src2	slong		
dst	slong		

**Opcode**    .S unit



Opcode map field used...	For operand type...	Unit
<i>src1</i>	sint	.S1, .S2
<i>src2</i>	xsint	
<i>dst</i>	sint	

**Description** *src1* is added to *src2* and saturated, if an overflow occurs according to the following rules:

1. If the *dst* is an int and  $src1 + src2 > 2^{31} - 1$ , then the result is  $2^{31} - 1$ .
2. If the *dst* is an int and  $src1 + src2 < -2^{31}$ , then the result is  $-2^{31}$ .
3. If the *dst* is a long and  $src1 + src2 > 2^{39} - 1$ , then the result is  $2^{39} - 1$ .
4. If the *dst* is a long and  $src1 + src2 < -2^{39}$ , then the result is  $-2^{39}$ .

The result is placed in *dst*. If a saturate occurs, the SAT bit in the control status register (CSR) is set one cycle after *dst* is written.

**Execution**

```
if (cond) src1 +s src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD](#)

**Examples** **Example 1**

SADD .L1 A1,A2,A3

**Before instruction**

**1 cycle after instruction**

A1	<table border="1"><tr><td>5A2E51A3h</td></tr></table>	5A2E51A3h	1,512,984,995	A1	<table border="1"><tr><td>5A2E51A3h</td></tr></table>	5A2E51A3h
5A2E51A3h						
5A2E51A3h						
A2	<table border="1"><tr><td>012A3FA2h</td></tr></table>	012A3FA2h	19,546,018	A2	<table border="1"><tr><td>012A3FA2h</td></tr></table>	012A3FA2h
012A3FA2h						
012A3FA2h						
A3	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		A3	<table border="1"><tr><td>5B589145h</td></tr></table>	5B589145h
xxxxxxxxh						
5B589145h						
CSR	<table border="1"><tr><td>00010100h</td></tr></table>	00010100h		CSR	<table border="1"><tr><td>00010100h</td></tr></table>	00010100h
00010100h						
00010100h						
SSR	<table border="1"><tr><td>00000000h</td></tr></table>	00000000h		SSR	<table border="1"><tr><td>00000000h</td></tr></table>	00000000h
00000000h						
00000000h						

**2 cycles after instruction**

A1	<table border="1"><tr><td>5A2E51A3h</td></tr></table>	5A2E51A3h
5A2E51A3h		

A2	012A3FA2h	
A3	5B589145h	
CSR	0001 0100h	Not saturated
SSR	0000 0000h	

### Example 2

SADD .L1 A1,A2,A3

#### Before instruction

A1	436771F2h	1,130,852,850
A2	5A2E51A3h	1,512,984,995
A3	xxxxxxxxh	
CSR	0001 0100h	
SSR	0000 0000h	

#### 1 cycle after instruction

A1	436771F2h	
A2	5A2E51A3h	
A3	7FFFFFFFh	2,147,483,647
CSR	0001 0100h	
SSR	0000 0000h	

#### 2 cycles after instruction

A1	436771F2h	
A2	5A2E51A3h	
A3	7FFFFFFFh	
CSR	0001 0300h	Saturated
SSR	0000 0001h	

### Example 3

SADD .L1X B2,A5:A4,A7:A6

**Before instruction****1 cycle after instruction**

A5:A4	0000000h	7C8339B1h	A5:A4	0000000h	7C8339B1h
2,088,974,769 <sup>1</sup>					
A7:A6	xxxxxxxxh	xxxxxxxxh	A7:A6	0000000h	8DAD 7953h
	2,376,956,243 <sup>1</sup>				
B2	112A3FA2h	287,981,474	B2	112A3FA2h	
CSR	00010100h		CSR	00010100h	
SSR	0000000h		SSR	0000000h	

**2 cycles after instruction**

A5:A4	0000000h	7C8339B1h
A7:A6	0000 0000h	8DAD 7953h
B2	112A3FA2h	
CSR	0001 0100h	Not saturated
SSR	0000 0000h	

1. Signed 40-bit (long) integer

## 4.250 SADD2

Add Two Signed 16-Bit Integers on Upper and Lower Register Halves With Saturation

**Syntax** **SADD2 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		
3		z													18	17
				5												
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	1	1	0	0	0	0	1	1	0	0	s	p	1	1
		5		1												

Opcode map field used...	For operand type...	Unit
src1	s2	.S1, .S2
src2	xs2	
dst	s2	

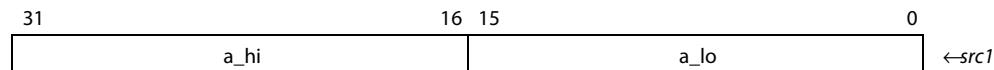
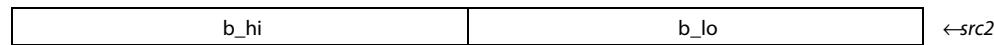
**Description**

Performs 2s-complement addition between signed, packed 16-bit quantities in *src1* and *src2*. The results are placed in a signed, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the signed 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

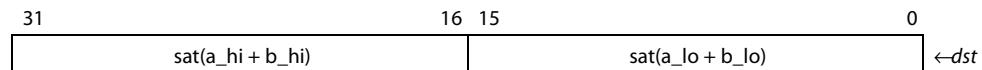
Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range  $-2^{15}$  to  $2^{15} - 1$ , inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than  $2^{15} - 1$ , then the result is set to  $2^{15} - 1$ .
- If the sum is less than  $-2^{15}$ , then the result is set to  $-2^{15}$ .

**SADD2**

↓

↓



**Note**—This operation is performed on each halfword separately. This instruction does not affect the SAT bit in CSR.

**Execution**

```
if (cond) {
    sat(msb16(src1) + msb16(src2)) → msb16(dst);
    sat(lsb16(src1) + lsb16(src2)) → lsb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD2](#), [SADD](#), [SADDUS2](#), [SADDU4](#), [SUB2](#)

**Examples** [Example 1](#)

SADD2 .S1 A2,A8,A9

**Before instruction****1 cycle after instruction**

A2	<table border="1"><tr><td>5789 F23Ah</td></tr></table>	5789 F23Ah	22409 -3526	A2	<table border="1"><tr><td>5789 F23Ah</td></tr></table>	5789 F23Ah
5789 F23Ah						
5789 F23Ah						
A8	<table border="1"><tr><td>74B8 4975h</td></tr></table>	74B8 4975h	29880 18805	A8	<table border="1"><tr><td>74B8 4975h</td></tr></table>	74B8 4975h
74B8 4975h						
74B8 4975h						
A9	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh		A9	<table border="1"><tr><td>7FFF 3BAFh</td></tr></table>	7FFF 3BAFh
xxxx xxxxh						
7FFF 3BAFh						

**Example 2**

SADD2 .S2 B2,B8,B12

**Before instruction****1 cycle after instruction**

B2	0124 847Ch	292 -31260	B2	0124 847Ch	
B8	01A6 A051h	422 -24495	B8	01A6 A051h	
B12	xxxx xxxxh		B12	02AC 8000h	684 -32768

## 4.251 SADDSUB

Parallel SADD and SSUB Operations On Common Inputs

**Syntax** **SADDSUB** (.unit) *src1, src2, dst\_o:dst\_e*

unit = .L1 or .L2

**Opcode**

31	30	29	28	27	dst				24	23	22	src2				src1		18	17	
0	0	0	1						0									5	5	
					4				5											
<i>src1</i>	x	0	0	0	1	1	1	0	1	1	0	s	p			1	1	0		
		5		1												1	1			

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sint	.L1, .L2
<i>src2</i>	xsint	
<i>dst</i>	dint	

**Description**

The following is performed in parallel:

1. *src2* is added with saturation to *src1*. The result is placed in *dst\_o*.
2. *src2* is subtracted with saturation from *src1*. The result is placed in *dst\_e*.

If either result saturates, the L1 or L2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst\_o:dst\_e*.

This instruction executes unconditionally and cannot be predicated.

**Execution**

$\text{sat}(\text{src1} + \text{src2}) \rightarrow \text{dst\_o}$   
 $\text{sat}(\text{src1} - \text{src2}) \rightarrow \text{dst\_e}$

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**

[ADDSSUB](#), [SADDSUB2](#)

**Examples**

**Example 1**

SADDSUB .L1 A0,A1,A3:A2

Before instruction		1 cycle after instruction	
A0	0700C005h	A2	0700C006h
A1	FFFFFFFFFFh	A3	0700C004h
CSR	00010100h	CSR <sup>1</sup>	00010300h

SSR	00000000h	SSR <sup>1</sup>	00000001h
-----	-----------	------------------	-----------

1. CSR.SAT and SSR.L1 set to 1, 2 cycles after instruction

### Example 2

SADDSUB .L2X B0,A1,B3:B2

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
B0	7FFFFFFFh	B2	7FFFFFEh
A1	00000001h	B3	7FFFFFFFh
CSR	00010100h	CSR <sup>1</sup>	00010300h
SSR	00000000h	SSR <sup>1</sup>	00000002h

1. CSR.SAT and SSR.L2 set to 1, 2 cycles after instruction

### Example 3

SADDSUB .L1X A0,B1,A3:A2

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
A0	80000000h	A2	80000000h
B1	00000001h	A3	80000001h
CSR	00010100h	CSR <sup>1</sup>	00010300h
SSR	00000000h	SSR <sup>1</sup>	00000001h

1. CSR.SAT and SSR.L1 set to 1, 2 cycles after instruction

## 4.252 SADDSUB2

Parallel SADD2 and SSUB2 Operations On Common Inputs

**Syntax** **SADDSUB2 (.unit) src1, src2, dst\_o:dst\_e**

unit = .L1 or .L2

**Opcode**

31	30	29	28	27	dst				24	23	22	src2				src1			
0	0	0	1						0										
					4							5				5			
						13	12	11	10	9	8	7	6	5	4	3	2	1	0
						src1	x	0	0	0	1	1	1	1	1	1	0	s	p
						5		1									1	1	

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

**Description**

A **SADD2** and a **SSUB2** operation are done in parallel.

For the **SADD2** operation, the upper and lower halves of the *src2* operand are added with saturation to the upper and lower halves of the *src1* operand. The values in *src1* and *src2* are treated as signed, packed 16-bit data and the results are written in signed, packed 16-bit format into *dst\_o*.

For the **SSUB2** operation, the upper and lower halves of the *src2* operand are subtracted with saturation from the upper and lower halves of the *src1* operand. The values in *src1* and *src2* are treated as signed, packed 16-bit data and the results are written in signed, packed 16-bit format into *dst\_e*.

This instruction executes unconditionally and cannot be predicated.



**Note**—These operations are performed separately on each halfword. This instruction does not affect the SAT bit in CSR or the L1 or L2 bits in SSR.

**Execution**

```
sat (lsb16(src1) + lsb16(src2)) → lsb16(dst_o)
sat (msb16(src1) + msb16(src2)) → msb16(dst_o)
sat (lsb16(src1) - lsb16(src2)) → lsb16(dst_e)
sat (msb16(src1) - msb16(src2)) → msb16(dst_e)
```

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**

[ADDSUB2](#), [SADDSUB2](#)

**Examples**

**Example 1**

SADDSUB2 .L1 A0,A1,A3:A2

		<b>Before instruction</b>	<b>1 cycle after instruction</b>	
A0		0700 C005h	A2	0701 C004h
A1		FFFF 0001h	A3	06FF C006h
CSR		00010100h	CSR <sup>1</sup>	00010100h
SSR		00000000h	SSR <sup>1</sup>	00000000h

1. CSR.SAT and SSR.L1 unchanged by operation

### Example 2

```
SADDSUB2 .L2X B0,A1,B3:B2
```

		<b>Before instruction</b>	<b>1 cycle after instruction</b>	
B0		7FFF 8000h	B2	7FFF 8001h
A1		FFFF FFFFh	B3	7FFE 8000h
CSR		00010100h	CSR <sup>1</sup>	00010100h
SSR		00000000h	SSR <sup>1</sup>	00000000h

1. CSR.SAT and SSR.L2 unchanged by operation

## 4.253 SADDSU2

Add Two Signed and Unsigned 16-Bit Integers on Register Halves With Saturation

**SADDSU2 (.unit) *src2*, *src1*, *dst***

*unit* = .S1 or .S2

**Opcode**

31	29	28	27	23 22					18 17					
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>					
3	1			5					5				5	
<i>src1</i>	<i>x</i>	1	1	0	0	0	1	1	1	0	0	<i>s</i>	<i>p</i>	
5	1											1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	u2	.S1, .S2
<i>src2</i>	xs2	
<i>dst</i>	u2	

**Description**

The **SADDSU2** pseudo-operation performs 2s-complement addition between unsigned and signed packed 16-bit quantities. The values in *src1* are treated as unsigned packed 16-bit quantities, and the values in *src2* are treated as signed packed 16-bit quantities. The results are placed in an unsigned packed 16-bit format into *dst*. The assembler uses the **SADDUS2 (.unit) *src1*, *src2*, *dst*** instruction to perform this operation.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the unsigned 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to  $2^{16} - 1$ , inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than  $2^{16} - 1$ , then the result is set to  $2^{16} - 1$ .

- If the sum is less than 0, then the result is cleared to 0.

**Execution**

```
if (cond) {
    sat(smsb16(src2) + umsb16(src1)) → umsb16(dst);
    sat(slsb16(src2) + ulsb16(src1)) → ulsb16(dst)
}
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SADD](#), [SADD2](#), [SADDUS2](#), [SADDU4](#)

## 4.254 SADDUS2

Add Two Unsigned and Signed 16-Bit Integers on Register Halves With Saturation

**Syntax** **SADDUS2** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	23 22					18 17					
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>					
3	1			5					5				5	
<i>src1</i>	<i>x</i>	1	1	0	0	0	1	1	1	0	0	<i>s</i>	<i>p</i>	
5	1											1	1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	u2	.S1, .S2
<i>src2</i>	xs2	
<i>dst</i>	u2	

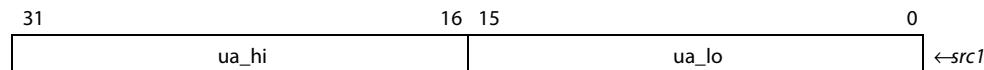
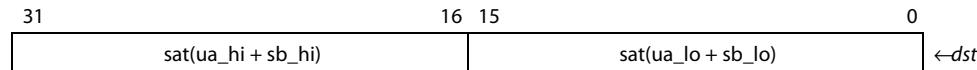
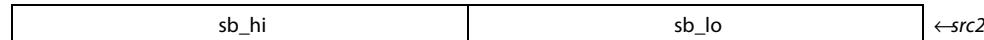
**Description**

Performs 2s-complement addition between unsigned and signed, packed 16-bit quantities. The values in *src1* are treated as unsigned, packed 16-bit quantities; and the values in *src2* are treated as signed, packed 16-bit quantities. The results are placed in an unsigned, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the sum between the unsigned 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to  $2^{16} - 1$ , inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than  $2^{16} - 1$ , then the result is set to  $2^{16} - 1$ .
- If the sum is less than 0, then the result is cleared to 0.

**SADDUS2**

**Note**—This operation is performed on each halfword separately. This instruction does not affect the SAT bit in CSR.

**Execution**

```
if (cond) {
    sat(umsb16(src1) + smsb16(src2)) → umsb16(dst);
    sat(ulsb16(src1) + slsb16(src2)) → ulsb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD2](#), [SADD](#), [SADD2](#), [SADDU4](#)

**Examples****Example 1**

SADDUS2 .S1 A2, A8, A9

Before instruction			1 cycle after instruction		
A2	<input type="text" value="5789 F23Ah"/>	22409 62010 unsigned	A2	<input type="text" value="5789 F23Ah"/>	
A8	<input type="text" value="74B8 4975h"/>	29880 18805 signed	A8	<input type="text" value="74B8 4975h"/>	
A9	<input type="text" value="xxxx xxxxh"/>		A9	<input type="text" value="CC41 FFFF"/>	52289 65535 unsigned

**Example 2**

SADDUS2 .S2 B2, B8, B12

	<b>Before instruction</b>	<b>1 cycle after instruction</b>		
B2	<table border="1"><tr><td>147C 0124h</td></tr></table> 5244 292 unsigned	147C 0124h	<table border="1"><tr><td>147C 0124h</td></tr></table>	147C 0124h
147C 0124h				
147C 0124h				
B8	<table border="1"><tr><td>A051 01A6h</td></tr></table> -24495 422 signed	A051 01A6h	<table border="1"><tr><td>A051 01A6h</td></tr></table>	A051 01A6h
A051 01A6h				
A051 01A6h				
B12	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh	<table border="1"><tr><td>0000 02ACh</td></tr></table> 0 684 unsigned	0000 02ACh
xxxx xxxxh				
0000 02ACh				

## 4.255 SADDU4

Add With Saturation, Four Unsigned 8-Bit Pairs for Four 8-Bit Results

**Syntax** **SADDU4 (.unit) *src1*, *src2*, *dst***

*unit* = .S1 or .S2

**Opcode**

31	29	28	27	dst					src2					src1		
3															5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	<i>src1</i>	x	1	1	0	0	1	1	1	1	0	0	s	p		
	5		1												1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	u4	.S1, .S2
<i>src2</i>	xu4	
<i>dst</i>	u4	

**Description**

Performs 2s-complement addition between unsigned, packed 8-bit quantities. The values in *src1* and *src2* are treated as unsigned, packed 8-bit quantities and the results are written into *dst* in an unsigned, packed 8-bit format.

For each pair of 8-bit quantities in *src1* and *src2*, the sum between the unsigned 8-bit value from *src1* and the unsigned 8-bit value from *src2* is calculated and saturated to produce an unsigned 8-bit result. The result is placed in the corresponding position in *dst*.

Saturation is performed on each 8-bit result independently. For each sum, the following tests are applied:

- If the sum is in the range 0 to  $2^8 - 1$ , inclusive, then no saturation is performed and the sum is left unchanged.
- If the sum is greater than  $2^8 - 1$ , then the result is set to  $2^8 - 1$ .

31	24	23	16	15	8	7	0
ua_3		ua_2		ua_1		ua_0	←src1

**SADDU4**

ub_3	ub_2	ub_1	ub_0				
↓	↓	↓	↓				
31	24	23	16	15	8	7	0
sat(ua_3 + ub_3)	sat(ua_2 + ub_2)	sat(ua_1 + ub_1)	sat(ua_0 + ub_0)				←dst



**Note**—This operation is performed on each 8-bit quantity separately. This instruction does not affect the SAT bit in CSR.

**Execution**

```
if (cond) {
    sat(ubyte0(src1) + ubyte0(src2)) → ubyte0(dst);
    sat(ubyte1(src1) + ubyte1(src2)) → ubyte1(dst);
    sat(ubyte2(src1) + ubyte2(src2)) → ubyte2(dst);
    sat(ubyte3(src1) + ubyte3(src2)) → ubyte3(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**[ADD4](#), [SADD](#), [SADD2](#), [SADDUS2](#), [SUB4](#)**Examples****Example 1**

SADDU4 .S1 A2, A8, A9

**Before instruction**

A2	57 89 F2 3Ah	87 137 242 58
		unsigned

**1 cycle after instruction**

A2	57 89 F2 3Ah
----	--------------

A8	74 B8 49 75h	116 184 73 117
		unsigned

A8	74 B8 49 75h
----	--------------

A9	xxxx xxxxh
----	------------

A9	CB FF FF AFh	203 255 255 175
		unsigned

### Example 2

SADDU4 .S2 B2, B8, B12

Before instruction		1 cycle after instruction			
B2	<table border="1"><tr><td>14 7C 01 24h</td></tr></table>	14 7C 01 24h	20 124 1 36	B2 <table border="1"><tr><td>14 7C 01 24h</td></tr></table>	14 7C 01 24h
14 7C 01 24h					
14 7C 01 24h					
B8	<table border="1"><tr><td>A0 51 01 A6h</td></tr></table>	A0 51 01 A6h	160 81 1 166	B8 <table border="1"><tr><td>A0 51 01 A6h</td></tr></table>	A0 51 01 A6h
A0 51 01 A6h					
A0 51 01 A6h					
B12	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh		B12 <table border="1"><tr><td>B4 CD 02 CA</td></tr></table> 180 205 2 202	B4 CD 02 CA
xxxx xxxxh					
B4 CD 02 CA					

## 4.256 SAT

Saturate a 40-Bit Integer to a 32-Bit Integer

**Syntax** **SAT (.unit) src2\_h:src2\_l, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	dst					src2					18	17	16
creg	z													0	0	
3		1			5					5						
0	0	0	x	1	0	0	0	0	0	0	1	1	0	s	p	1

Opcode map field used...	For operand type...	Unit
src2 dst	slong sint	.L1, .L2

**Description** A 40-bit *src2* value is converted to a 32-bit value. If the value in *src2* is greater than what can be represented in 32-bits, *src2* is saturated. The result is placed in *dst*. If a saturate occurs, the SAT bit in the control status register (CSR) is set one cycle after *dst* is written.

**Execution**

```
if (cond){
    if (src2 > (231 - 1)), (231 - 1) → dst
    else if (src2 < -231), -231 → dst
    else src231..0 → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**Examples** **Example 1**

SAT .L2 B1:B0,B5

**Before instruction**

B1:B0	0000001Fh	3413539Ah
-------	-----------	-----------

B5	xxxxxxxxh
----	-----------

CSR	00010100h
-----	-----------

**1 cycle after instruction**

B1:B0	0000001Fh	3413539Ah
-------	-----------	-----------

B5	7FFFFFFFh
----	-----------

CSR	00010100h
-----	-----------

SSR 

00000000h
-----------

SSR 

00000000h
-----------

**2 cycles after instruction**B1:B0 

0000001Fh	3413539Ah
-----------	-----------

B5 

7FFFFFFFh
-----------

CSR 

00010300h
-----------

 SaturatedSSR 

00000002h
-----------

**Example 2**

SAT .L2 B1:B0 , B5

**Before instruction**B1:B0 

00000000h	A1907321h
-----------	-----------

B5 

xxxxxxxxh
-----------

CSR 

00010100h
-----------

SSR 

00000000h
-----------

**1 cycle after instruction**B1:B0 

00000000h	A1907321h
-----------	-----------

B5 

7FFFFFFFh
-----------

CSR 

00010100h
-----------

SSR 

00000000h
-----------

**2 cycles after instruction**B1:B0 

00000000h	A1907321h
-----------	-----------

B5 

7FFFFFFFh
-----------

CSR 

00010300h
-----------

 SaturatedSSR 

00000002h
-----------

**Example 3**

SAT .L2 B1:B0 , B5

**Before instruction****1 cycle after instruction**

B1:B0	<table border="1"><tr><td>000000FFh</td><td>A1907321h</td></tr></table>	000000FFh	A1907321h	B1:B0	<table border="1"><tr><td>000000FFh</td><td>A1907321h</td></tr></table>	000000FFh	A1907321h
000000FFh	A1907321h						
000000FFh	A1907321h						
B5	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh	B5	<table border="1"><tr><td>A1907321h</td></tr></table>	A1907321h		
xxxxxxxxh							
A1907321h							
CSR	<table border="1"><tr><td>00010100h</td></tr></table>	00010100h	CSR	<table border="1"><tr><td>00010100h</td></tr></table>	00010100h		
00010100h							
00010100h							
SSR	<table border="1"><tr><td>00000000h</td></tr></table>	00000000h	SSR	<table border="1"><tr><td>00000000h</td></tr></table>	00000000h		
00000000h							
00000000h							

**2 cycles after instruction**

B1:B0	<table border="1"><tr><td>000000FFh</td><td>A1907321h</td></tr></table>	000000FFh	A1907321h
000000FFh	A1907321h		
B5	<table border="1"><tr><td>A1907321h</td></tr></table>	A1907321h	
A1907321h			
CSR	<table border="1"><tr><td>00010100h</td></tr></table>	00010100h	
00010100h			
SSR	<table border="1"><tr><td>00000000h</td></tr></table>	00000000h	
00000000h			

Not saturated

## 4.257 SET

Set a Bit Field

**Syntax**    **SET (.unit) src2, csta, cstb, dst**

or

**SET (.unit) src2, src1, dst**

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Sc5	<a href="#">Figure F-22</a>

**Opcode**    Constant form:

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3	1		5			5	
13	12		8	7	6	5	4
<i>csta</i>		<i>cstb</i>	1	0	0	0	1
5		5					
1						1	0
1							1
							1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	uint	.S1, .S2
<i>csta</i>	ucst5	
<i>cstb</i>	ucst5	
<i>dst</i>	uint	

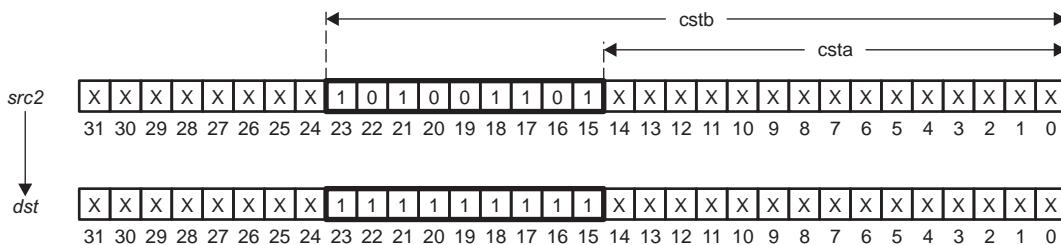
**Opcode**    Register form:

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3	1		5			5	
13	12	11	10	9	8	7	6
<i>src1</i>	x	1	1	1	0	1	1
5	1						
1						0	0
1						s	p
						1	1

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>src2</i>	xuint	.S1, .S2
<i>src1</i>	uint	
<i>dst</i>	uint	

**Description** For  $cstb > csta$ , the field in  $src2$  as specified by  $csta$  to  $cstb$  is set to all 1s in  $dst$ . The  $csta$  and  $cstb$  operands may be specified as constants or in the 10 LSBs of the  $src1$  register, with  $cstb$  being bits 0-4 ( $src1_{4..0}$ ) and  $csta$  being bits 5-9 ( $src1_{9..5}$ ).  $csta$  is the LSB of the field and  $cstb$  is the MSB of the field. In other words,  $csta$  and  $cstb$  represent the beginning and ending bits, respectively, of the field to be set to all 1s in  $dst$ . The LSB location of  $src2$  is bit 0 and the MSB location of  $src2$  is bit 31.

In the following example,  $csta$  is 15 and  $cstb$  is 23. For the register version of the instruction, only the 10 LSBs of the  $src1$  register are valid. If any of the 22 MSBs are non-zero, the result is invalid.



For  $cstb < csta$ , the  $src2$  register is copied to  $dst$ . The  $csta$  and  $cstb$  operands may be specified as constants or in the 10 LSBs of the  $src1$  register, with  $cstb$  being bits 0-4 ( $src1_{4..0}$ ) and  $csta$  being bits 5-9 ( $src1_{9..5}$ ).

**Execution** If the constant form is used when  $cstb > csta$ :

if (cond)  $src_2$  SET  $csta, cstb \rightarrow dst$   
else nop

If the register form is used when  $cstb > csta$ :

if (cond)  $src2 \text{ SET } src1_{9..5}, src1_{4..0} \rightarrow dst$

**ANSWER** The answer is 1000.

## *Pipeline*

<b>Pipeline Stage</b>	<b>E1</b>
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	<i>.S</i>

**Instruction Type** Single-cycle

*Delay Slots* 0

**See Also** [CLR](#)

## *Examples*      Example 1

SET .S1 A0,7,21,A1

**Before instruction****1 cycle after instruction**A0      

4B134A1Eh
-----------

A0      

4B134A1Eh
-----------

A1      

xxxxxxxxh
-----------

A1      

4B3FFF9Eh
-----------

**Example 2**

SET .S2      B0,B1,B2

**Before instruction****1 cycle after instruction**B0      

9ED31A31h
-----------

B0      

9ED31A31h
-----------

B1      

0000C197h
-----------

B1      

0000C197h
-----------

B2      

xxxxxxxxh
-----------

B2      

9EFFFA31h
-----------

## 4.258 SHFL

Shuffle

**Syntax**    **SHFL (.unit) src2, dst**

unit = .M1 or .M2

**Opcode**

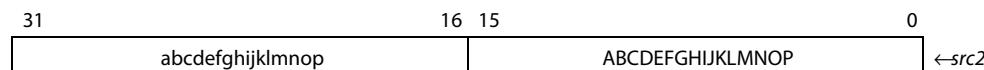
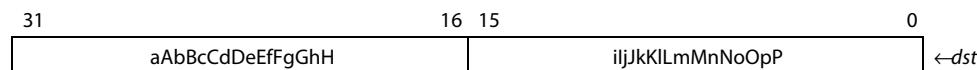
31	29	28	27							23	22							18	17	16
				<i>dst</i>								<i>src2</i>							1	1
3				1						5								5		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.M1, .M2
<i>dst</i>	uint	

**Description**

Performs an interleave operation on the two halfwords in *src2*. The bits in the lower halfword of *src2* are placed in the even bit positions in *dst*, and the bits in the upper halfword of *src2* are placed in the odd bit positions in *dst*.

As a result, bits 0, 1, 2, ..., 14, 15 of *src2* are placed in bits 0, 2, 4, ..., 28, 30 of *dst*. Likewise, bits 16, 17, 18, .. 30, 31 of *src2* are placed in bits 1, 3, 5, ..., 29, 31 of *dst*.

**SHFL**

**Note**—The **SHFL** instruction is the exact inverse of the **DEAL** instruction (see [DEAL](#)).

**Execution**

```
if (cond) {
    src231,30,29...16 → dst31,29,27...1
    src215,14,13...0 → dst30,28,26...0
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src2	
Written		dst
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [DEAL](#)

**Example** SHFL .M1 A1,A2

**Before instruction**

A1	B174 6CA4h
A2	xxxx xxxxh

**2 cycles after instruction**

A1	B174 6CA4h
A2	9E52 6E30h

## 4.259 SHFL3

3-Way Bit Interleave On Three 16-Bit Values Into a 48-Bit Result

**Syntax** **SHFL3 (.unit) src1, src2, dst\_o:dst\_e**

unit = .L1 or .L2

**Opcode**

31	30	29	28	27	dst					src2					src1			
0	0	0	1		5					5					5			
					13	12	11	10	9	8	7	6	5	4	3	2	1	0
					src1	x	0	1	1	0	1	1	0	1	1	0	s	p

Opcode map field used...	For operand type...	Unit
src1	sint	.L1, .L2
src2	xsint	
dst	dint	

**Description** Performs a 3-way bit interleave on three 16-bit values and creating a 48-bit result.

This instruction executes unconditionally and cannot be predicated.

31	16 15										0		
a15	a14	a13	...	a2	a1	a0	b15	b14	b13	...	b2	b1	b0

←src1

c15	c14	c13	...	c2	c1	c0	d15	d14	d13	...	d2	d1	d0
-----	-----	-----	-----	----	----	----	-----	-----	-----	-----	----	----	----

←src2

**SHFL3**



31	16 15										0		
0	0	0	...	0	0	0	a15	b15	d15	...	b11	d11	a10

←dst\_o

b10	d10	a9	...	d6	a5	b5	d5	a4	b4	...	a0	b0	d0
-----	-----	----	-----	----	----	----	----	----	----	-----	----	----	----

←dst\_e

**Execution**

```

int inp0, inp1, inp2
dword result;
inp0 = src2 & FFFFh;
inp1 = src1 & FFFFh;
inp2 = src1 >> 16 & FFFFh;
result = 0;
for (I = 0; I < 16; I++)

```

```
{
    result |= (inp0 >> I & 1) << (I × 3) ;
    result |= (inp1 >> I & 1) << ((I × 3) + 1);
    result |= (inp2 >> I & 1) << I ((I × 3) + 2)
}
```

**Instruction Type** Single-cycle

**Delay Slots** 0

**Example** SHFL3 .L1 A0,A1,A3:A2

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A0	87654321h	A2	7E179306h
A1	12345678h	A3	00008C11h

## 4.260 SHL

Arithmetic Shift Left

**Syntax**    **SHL (.unit) src2, src1, dst**

or

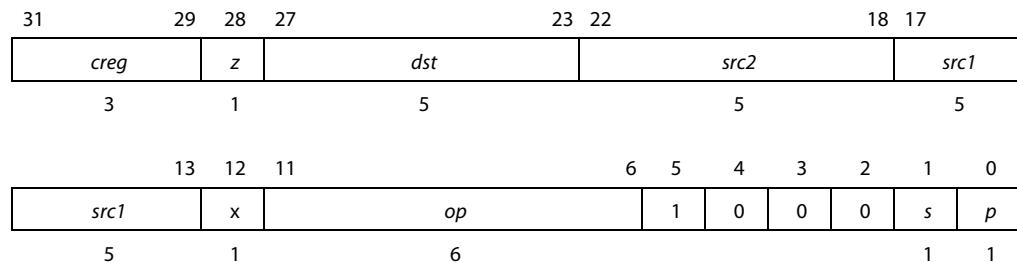
**SHL (.unit) src2\_h:src2\_l, src1, dst\_h:dst\_l**

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	S3i	<a href="#">Figure F-18</a>
	Ssh5	<a href="#">Figure F-20</a>
	S2sh	<a href="#">Figure F-21</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	xsint	.S1, .S2	110011
<i>src1</i>	uint		
<i>dst</i>	sint		
<i>src2</i>	slong	.S1, .S2	110001
<i>src1</i>	uint		
<i>dst</i>	slong		
<i>src2</i>	xuint	.S1, .S2	010011
<i>src1</i>	uint		
<i>dst</i>	ulong		
<i>src2</i>	xsint	.S1, .S2	110010
<i>src1</i>	ucst5		
<i>dst</i>	sint		
<i>src2</i>	slong	.S1, .S2	110000
<i>src1</i>	ucst5		
<i>dst</i>	slong		
<i>src2</i>	xuint	.S1, .S2	010010
<i>src1</i>	ucst5		
<i>dst</i>	ulong		

**Description**

The *src2* operand is shifted to the left by the *src1* operand. The result is placed in *dst*. When a register is used, the six LSBs specify the shift amount and valid values are 0-40. When an immediate is used, valid shift amounts are 0-31. If *src2* is a register pair, only the bottom 40 bits of the register pair are shifted. The upper 24 bits of the register pair are unused.

If  $39 < src1 < 64$ ,  $src2$  is shifted to the left by 40. Only the six LSBs of  $src1$  are used by the shifter, so any bits set above bit 5 do not affect execution.

**Execution**

```
if (cond) (src2 & 0xFFFFFFF) << src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	$src1, src2$
Written	$dst$
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ROTL](#), [SHLMB](#), [SHR](#), [SSHIL](#), [SSHVL](#)

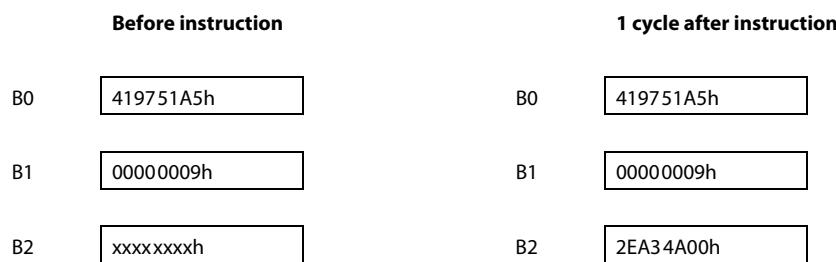
**Examples** [Example 1](#)

SHL .S1 A0,4,A1



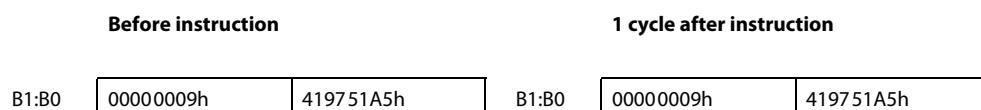
[Example 2](#)

SHL .S2 B0,B1,B2



[Example 3](#)

SHL .S2 B1:B0,B2,B3:B2



B2	<span style="border: 1px solid black; padding: 2px;">00000022h</span>	B2	<span style="border: 1px solid black; padding: 2px;">00000000h</span>
B3:B2	<span style="border: 1px solid black; padding: 2px;">xxxxxxxxh</span>	<span style="border: 1px solid black; padding: 2px;">xxxxxxxxh</span>	B3:B2 <span style="border: 1px solid black; padding: 2px;">00000094h</span> <span style="border: 1px solid black; padding: 2px;">00000000h</span>

#### Example 4

SHL .S1 A5:A4 , 0 , A1:A0

		<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A5:A4	FFFF FFFFh	FFFF FFFFh		A5:A4	FFFF FFFFh
A1:A0	xxxxxxxxh	xxxxxxxxh		A1:A0	000000FFh

## 4.261 SHL2

2-Way SIMD Shift Left, Packed Signed 16-bit

**Syntax** **SHL2 (.unit) src1, src2, dst**

unit = .S1 or .S2

**Opcode** Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opfield	1	0	0	0	s	p

5                    5                    5                    6

Opcode map field used...	For operand type...	Unit	Opfield
src2,src1,dst	xop1,op2,dst	.S1 or .S2	010011
src2,src1,dst	xop1,ucst5,dst	.S1 or .S2	000011

**Description** The SHL2 instruction performs a left right on packed 16-bit quantities. The values in *xop1* are viewed as two packed 16-bit quantities. The lower four bits of *op2* or *ucst4* are treated as a shift amount. The same shift amount is applied to both input data. The results are placed in a signed packed 16-bit format.

For each unsigned 16-bit quantity in *xop1*, the quantity is shifted left by the specified number of bits. Bits shifted out of the most-significant bit of each 16-bit quantity are discarded.

For correct operation bit 4 (the fifth bit) of the constant field must be set to 0.



**Execution** `smsb16(src2) << src1 -> smsb16(dst)`  
`s1sb16(src2) << src1 -> s1sb16(dst)`

<b>Instruction Type</b>	Single cycle
<b>Delay Slots</b>	0
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">SHR2</a> , <a href="#">SHRU2</a> , <a href="#">DSHL2</a>
<b>Example</b>	<pre>A0 == 0x1234fedc SHL2 .S A0,4,A15 A15 == 0x2340edc0  A0 == 0x1234fedc A1 == 0x00000004 SHL2 .S A0,A1,A5 A5 == 0x2340edc0  B0 == 0xabcd1234 B1 == 0x00000008 SHL2 .S B0,B1,B5 B5 == 0xcd003400</pre>

## 4.262 SHLMB

Shift Left and Merge Byte

**Syntax** **SHLMB** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, .S2

**Opcode** .L unit

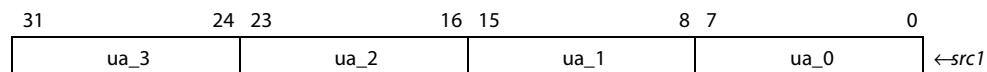
31	29	28	27						23	22						18	17
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					<i>src1</i>	
3		1			5						5					5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
<i>src1</i>	x	1	1	0	0	0	0	1	1	1	1	0	s	p			
5		1											1	1			
<b>Opcode map field used...</b>										<b>For operand type...</b>					<b>Unit</b>		
<i>src1</i>										u4					.L1, .L2		
<i>src2</i>										xu4							
<i>dst</i>										u4							

**Opcode** .S unit

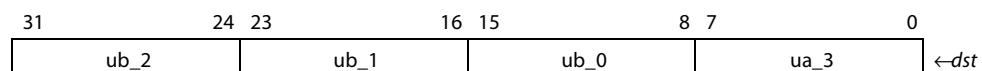
31	29	28	27						23	22						18	17
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					<i>src1</i>	
3		1			5						5					5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
<i>src1</i>	x	1	1	1	0	0	1	1	1	0	0	0	s	p			
5		1											1	1			
<b>Opcode map field used...</b>										<b>For operand type...</b>					<b>Unit</b>		
<i>src1</i>										u4					.S1, S2		
<i>src2</i>										xu4							
<i>dst</i>										u4							

**Description**

Shifts the contents of *src2* left by 1 byte, and then the most-significant byte of *src1* is merged into the least-significant byte position. The result is placed in *dst*.

**SHLMB**

$\downarrow$



**Execution**

```

if (cond) {
    ubyte2(src2) → ubyte3(dst);
    ubyte1(src2) → ubyte2(dst);
    ubyte0(src2) → ubyte1(dst);
    ubyte3(src1) → ubyte0(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ROTL](#), [SHL](#), [SHRMB](#)

**Examples****Example 1**

SHLMB .L1 A2, A8, A9

Before instruction		1 cycle after instruction	
A2	3789 F23Ah	A2	3789 F23Ah
A8	04B8 4975h	A8	04B8 4975h
A9	xxxx xxxxh	A9	B849 7537h

**Example 2**

SHLMB .S2 B2,B8, B12

	<b>Before instruction</b>		<b>1 cycle after instruction</b>
B2	0124 2451h		0124 2451h
B8	01A6 A051h		01A6 A051h
B12	xxxx xxxxh		A6A0 5101h

## 4.263 SHR

Arithmetic Shift Right

**Syntax** **SHR** (.unit) *src2*, *src1*, *dst*

or

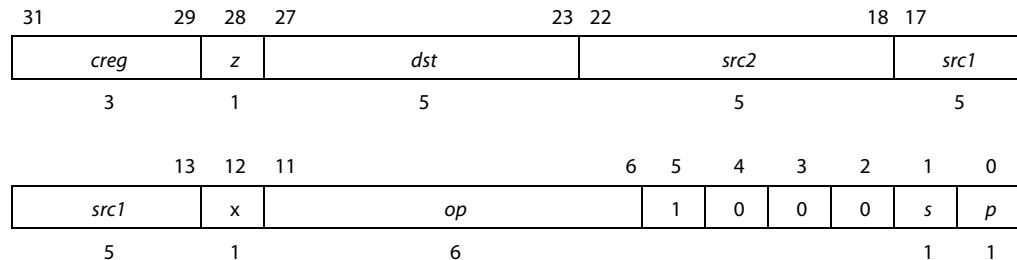
**SHR** (.unit) *src2\_h:src2\_l*, *src1*, *dst*

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	S3i	<a href="#">Figure F-18</a>
	Ssh5	<a href="#">Figure F-20</a>
	S2sh	<a href="#">Figure F-21</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	xsint	.S1, .S2	110111
<i>src1</i>	uint		
<i>dst</i>	sint		
<i>src2</i>	slong	.S1, .S2	110101
<i>src1</i>	uint		
<i>dst</i>	slong		
<i>src2</i>	xsint	.S1, .S2	110110
<i>src1</i>	ucst5		
<i>dst</i>	sint		
<i>src2</i>	slong	.S1, .S2	110100
<i>src1</i>	ucst5		
<i>dst</i>	slong		

**Description**

The *src2* operand is shifted to the right by the *src1* operand. The sign-extended result is placed in *dst*. When a register is used, the six LSBs specify the shift amount and valid values are 0-40. When an immediate value is used, valid shift amounts are 0-31. If *src2* is a register pair, only the bottom 40 bits of the register pair are shifted. The upper 24 bits of the register pair are unused.

If  $39 < src1 < 64$ ,  $src2$  is shifted to the right by 40. Only the six LSBs of  $src1$  are used by the shifter, so any bits set above bit 5 do not affect execution.

**Execution**

```
if (cond) (src2 & 0xFFFFFFF) >>s src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	$src1, src2$
Written	$dst$
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SHL](#), [SHR2](#), [SHRMB](#), [SHRU](#), [SHRU2](#), [SSHVR](#)

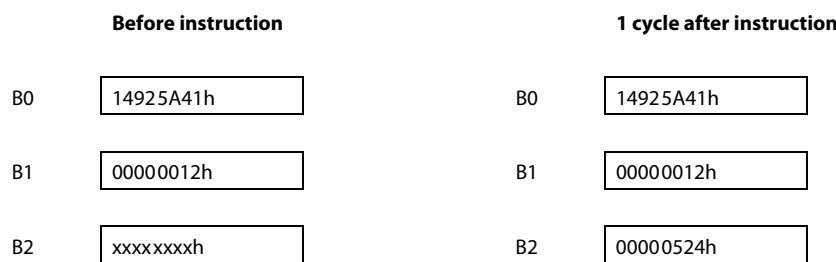
**Examples** [Example 1](#)

SHR .S1 A0,8,A1



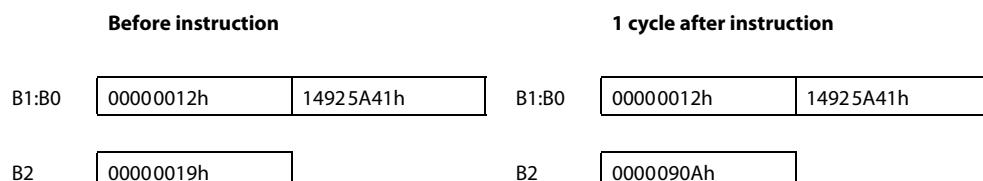
### Example 2

SHR .S2 B0,B1,B2



### Example 3

SHR .S2 B1:B0,B2,B3:B2



B3:B2	xxxxxxxxh	xxxxxxxxh	B3:B2	0000000h	0000090Ah
-------	-----------	-----------	-------	----------	-----------

### Example 4

SHR .S1 A5:A4,0,A1:A0

Before instruction		1 cycle after instruction			
A5:A4	FFFF FFFFh	FFFF FFFFh	A5:A4	FFFF FFFFh	FFFF FFFFh
A1:A0	xxxxxxxxh	xxxxxxxxh	A1:A0	0000 00FFh	FFFF FFFFh

4.264 SHR2

## Arithmetic Shift Right, Signed, Packed 16-Bit

**Syntax**      **SHR2** (.unit) *src2*, *src1*, *dst*

unit = .S1 or .S2

**Opcode** .S unit (uint form)

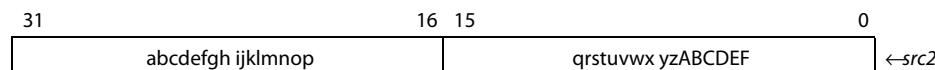
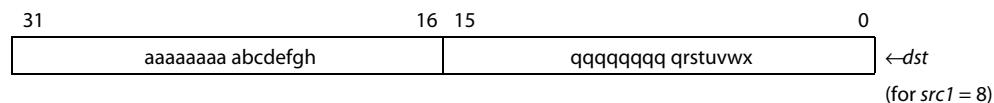
31	29	28	27	23 22				18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>		
3	1	5				5				5		
13	12	11	10	9	8	7	6	5	4	3	2	1 0
<i>src1</i>	x	1	1	0	1	1	1	1	1	0	0	<i>s p</i>
5	1									1	1	

**Opcode** .S unit (cst form)

### Description

Performs an arithmetic shift right on signed, packed 16-bit quantities. The values in *src2* are treated as signed, packed 16-bit quantities. The lower 5 bits of *src1* are treated as the shift amount. The results are placed in a signed, packed 16-bit format into *dst*.

For each signed 16-bit quantity in *src2*, the quantity is shifted right by the number of bits specified in the lower 5 bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero. The shifted quantity is sign-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of the signed 16-bit quantity are discarded.

**SHR2**

**Note**—If the shift amount specified in *src1* is in the range 16 to 31, the behavior is identical to a shift value of 15.

**Execution**

```
if (cond) {
    smsb16(src2) >> src1 → smsb16(dst);
    s1sb16(src2) >> src1 → s1sb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

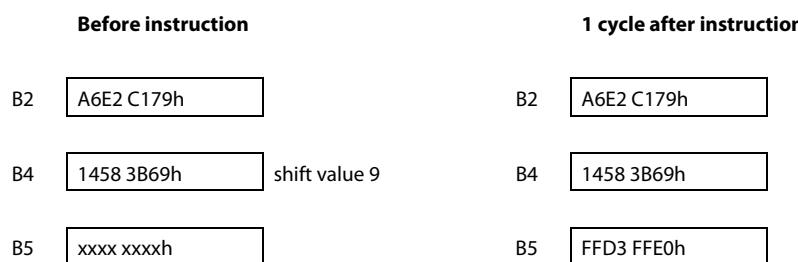
**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SHL](#), [SHR](#), [SHRMB](#), [SHRU](#), [SHRU2](#)

**Examples** [Example 1](#)

SHR2 .S2 B2,B4,B5



**Example 2**

SHR2 .S1 A4,0fh,A5 ; shift value is 15

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
A4	000A 87AFh	A4	000A 87AFh
A5	xxxx xxxxh	A5	0000 FFFFh

## 4.265 SHRMB

Shift Right and Merge Byte

**Syntax** **SHRMB (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2

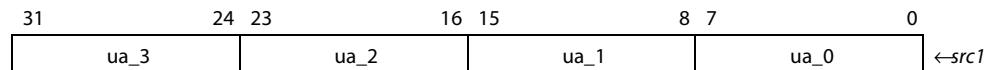
**Opcode** .L unit

31	29	28	27						23	22						
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					
3		1		5					5		5				5	
<i>src1</i>	<i>x</i>	1	1	0	0	0	1	0	1	1	0	<i>s</i>	<i>p</i>		0	
5		1										1	1			
<b>Opcode map field used...</b>																
<b>For operand type...</b>																
<b>Unit</b>																
<i>src1</i> u4 .L1, .L2																
<i>src2</i> xu4																
<i>dst</i> u4																

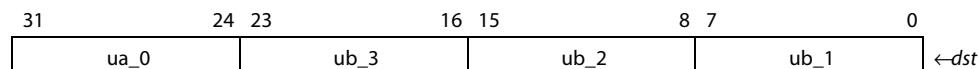
**Opcode** .S unit

31	29	28	27						23	22					
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>				
3		1		5					5		5				5
<i>src1</i>	<i>x</i>	1	1	1	0	1	0	1	1	1	0	0	<i>s</i>	<i>p</i>	0
5		1										1	1		
<b>Opcode map field used...</b>															
<b>For operand type...</b>															
<b>Unit</b>															
<i>src1</i> u4 .S1, .S2															
<i>src2</i> xu4															
<i>dst</i> u4															

**Description** Shifts the contents of *src2* right by 1 byte, and then the least-significant byte of *src1* is merged into the most-significant byte position. The result is placed in *dst*.

**SHRMB**

↓



**Execution**

```
if (cond) {
    ubyte0(src1) → ubyte3(dst);
    ubyte3(src2) → ubyte2(dst);
    ubyte2(src2) → ubyte1(dst);
    ubyte1(src2) → ubyte0(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SHL](#), [SHLMB](#), [SHR](#), [SHR2](#), [SHRU](#), [SHRU2](#)

**Examples** [Example 1](#)

SHRMB .L1 A2,A8,A9

**Before instruction**

A2	3789 F23Ah
A8	04B8 4975h
A9	xxxx xxxxh

**1 cycle after instruction**

A2	3789 F23Ah
A8	04B8 4975h
A9	3A04 B849h

**Example 2**

SHRMB .S2 B2,B8,B12

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B2	0124 2451h	B2	0124 2451h
B8	01A6 A051h	B8	01A6 A051h
B12	xxxx xxxxh	B12	5101 A6A0h

## 4.266 SHRU

Logical Shift Right

**Syntax** **SHRU** (.unit) *src2, src1, dst*

or

**SHRU** (.unit) *src2\_h:src2\_l, src1, dst\_h:dst\_l*

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Ssh5	<a href="#">Figure F-20</a>
	S2sh	<a href="#">Figure F-21</a>

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
13	12	11		6	5	4	3
<i>src1</i>	<i>x</i>		<i>op</i>	1	0	0	0
5	1		6				
1				1	1	0	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	xuint	.S1, .S2	100111
<i>src1</i>	uint		
<i>dst</i>	uint		
<i>src2</i>	ulong	.S1, .S2	100101
<i>src1</i>	uint		
<i>dst</i>	ulong		
<i>src2</i>	xuint	.S1, .S2	100110
<i>src1</i>	ucst5		
<i>dst</i>	uint		
<i>src2</i>	ulong	.S1, .S2	100100
<i>src1</i>	ucst5		
<i>dst</i>	ulong		

**Description**

The *src2* operand is shifted to the right by the *src1* operand. The zero-extended result is placed in *dst*. When a register is used, the six LSBs specify the shift amount and valid values are 0-40. When an immediate value is used, valid shift amounts are 0-31. If *src2* is a register pair, only the bottom 40 bits of the register pair are shifted. The upper 24 bits of the register pair are unused.

If  $39 < src1 < 64$ ,  $src2$  is shifted to the right by 40. Only the six LSBs of  $src1$  are used by the shifter, so any bits set above bit 5 do not affect execution.

**Execution**

```
if (cond) (src2 & 0xFFFFFFFF) >>z src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	$src1, src2$
Written	$dst$
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SHL](#), [SHR](#), [SHR2](#), [SHRMB](#), [SHRU2](#)

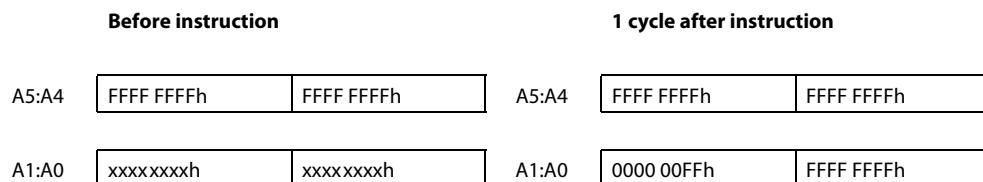
**Examples** [Example 1](#)

SHRU .S1 A0,8,A1



### Example 2

SHRU .S1 A5:A4,0,A1:A0



4.267 SHRU2

## Arithmetic Shift Right, Unsigned, Packed 16-Bit

**Syntax**    **SHRU2** (.unit) *src2*, *src1*, *dst*

unit = .S1 or .S2

**Opcode** .S unit (uint form)

31	29	28	27	23 22				18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>		
3	1	5				5				5		
13	12	11	10	9	8	7	6	5	4	3	2	1    0
<i>src1</i>	x	1	1	1	0	0	0	1	1	0	0	s    p
5	1									1	1	

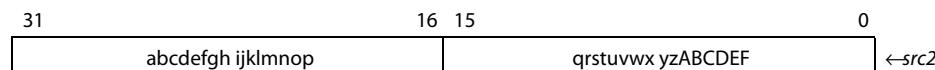
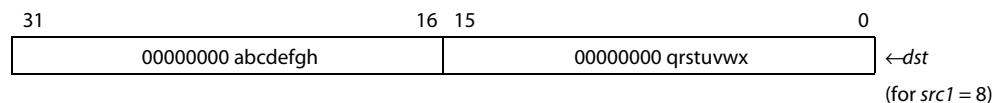
**Opcode** .S unit (cst form)

31	29	28	27	23 22				18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>		
3	1	5				5				5		
13	12	11	10	9	8	7	6	5	4	3	2	1 0
<i>src1</i>	x	0	1	1	0	0	1	1	0	0	0	s p
5	1									1	1	

### Description

Performs an arithmetic shift right on unsigned, packed 16-bit quantities. The values in *src2* are treated as unsigned, packed 16-bit quantities. The lower 5 bits of *src1* are treated as the shift amount. The results are placed in an unsigned, packed 16-bit format into *dst*.

For each unsigned 16-bit quantity in *src2*, the quantity is shifted right by the number of bits specified in the lower 5 bits of *src1*. Bits 5 through 31 of *src1* are ignored and may be non-zero. The shifted quantity is zero-extended, and placed in the corresponding position in *dst*. Bits shifted out of the least-significant bit of the signed 16-bit quantity are discarded.

**SHRU2**

**Note**—If the shift amount specified in  $src1$  is in the range of 16 to 31, the  $dst$  will be cleared to all zeros.

**Execution**

```
if (cond) {
    umsb16(src2) >> src1 → umsb16(dst);
    ulsb16(src2) >> src1 → ulsb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	$src1, src2$
Written	$dst$
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SHL](#), [SHR](#), [SHR2](#), [SHRMB](#), [SHRU](#)

**Examples** [Example 1](#)

SHRU2 .S2 B2,B4,B5

Before instruction		1 cycle after instruction	
B2	A6E2 C179h	B2	A6E2 C179h
B4	1458 3B69h	Shift value 9	1458 3B69h
B5	xxxx xxxxh		0053 0060h

**Example 2**

SHRU2 .S1 A4,0Fh,A5 ; Shift value is 15

**Before instruction**A4 

000A 87AFh
------------

A5 

xxxx xxxxh
------------

**1 cycle after instruction**A4 

000A 87AFh
------------

A5 

0000 0001h
------------

## 4.268 SMPY

Multiply Signed 16 LSB × Signed 16 LSB With Left Shift and Saturation

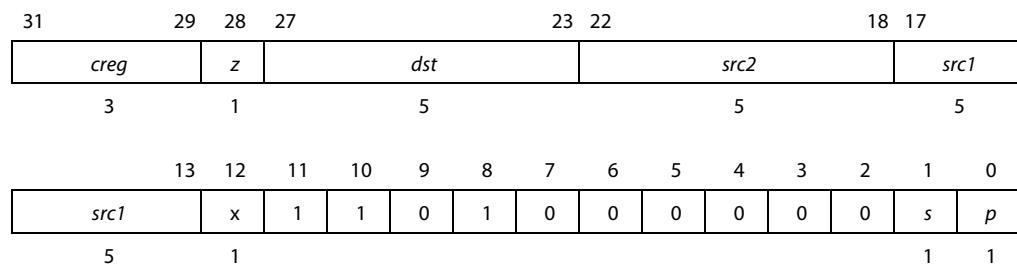
**Syntax** **SMPY (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

**Opcode**



Opcode map field used...	For operand type...	Unit
<i>src1</i>	slsb16	.M1, .M2
<i>src2</i>	xlsb16	
<i>dst</i>	sint	

**Description**

The 16 least-significant bits of *src1* operand is multiplied by the 16 least-significant bits of the *src2* operand. The result is left shifted by 1 and placed in *dst*. If the left-shifted result is 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturate occurs, the SAT bit in CSR is set one cycle after *dst* is written. The source operands are signed by default.

**Execution**

```
if (cond) {
    if (((lsb16(src1) × lsb16(src2)) << 1) != 8000 0000h),
        ((lsb16(src1) × lsb16(src2)) << 1) → dst
    else
        7FFF FFFFh → dst
}
else
    nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type**

Single-cycle (16 × 16)

**Delay Slots**

1

**See Also**

[MPY](#), [SMPYH](#), [SMPYHL](#), [SMPYLH](#)

**Example**

SMPY .M1 A1,A2,A3

	<b>Before instruction</b>		<b>2 cycles after instruction</b>	
A1	00000123h	291 <sup>1</sup>	A1	00000123h
A2	01E0FA81h	-1407 <sup>1</sup>	A2	01E0FA81h
A3	xxxxxxxxh		A3	FFF38146h -818,874
CSR	00010100h		CSR	00010100h Not saturated
SSR	00000000h		SSR	00000000h

1. Signed 16-LSB integer

## 4.269 SMPYH

Multiply Signed 16 MSB × Signed 16 MSB With Left Shift and Saturation

**Syntax** **SMPYH** (.unit) *src1*, *src2*, *dst*

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

**Opcode**

31	29	28	27		23	22		18	17
<i>creg</i>	<i>z</i>			<i>dst</i>			<i>src2</i>		<i>src1</i>
3	1			5			5		5
<i>src1</i>	<i>x</i>	0	0	0	1	0	0	0	0
5	1							1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	smsb16	.M1, .M2
<i>src2</i>	xsmsb16	
<i>dst</i>	sint	

**Description**

The 16 most-significant bits of *src1* operand is multiplied by the 16 most-significant bits of the *src2* operand. The result is left shifted by 1 and placed in *dst*. If the left-shifted result is 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturation occurs, the SAT bit in CSR is set one cycle after *dst* is written. The source operands are signed by default.

**Execution**

```
if (cond) {
    if (((msb16(src1) × msb16(src2)) << 1) != 8000 0000h),
        ((msb16(src1) × msb16(src2)) << 1) → dst
    else
        7FFF FFFFh → dst
}
else
    nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1</i> , <i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type**

Single-cycle (16 × 16)

**Delay Slots**

1

**See Also**

[MPYH](#), [SMPY](#), [SMPYHL](#), [SMPYLN](#)

## 4.270 SMPYHL

Multiply Signed 16 MSB × Signed 16 LSB With Left Shift and Saturation

**Syntax** **SMPYHL (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

**Opcode**

31	29	28	27	23 22					18	17		
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>		<i>src1</i>	
3	1			5					5		5	
src1	x	0	1	0	1	0	0	0	0	0	s	p

Opcode map field used...	For operand type...	Unit
<i>src1</i>	smsb16	.M1, .M2
<i>src2</i>	xslsb16	
<i>dst</i>	sint	

**Description**

The 16 most-significant bits of the *src1* operand is multiplied by the 16 least-significant bits of the *src2* operand. The result is left shifted by 1 and placed in *dst*. If the left-shifted result is 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturation occurs, the SAT bit in CSR is set one cycle after *dst* is written.

**Execution**

```
if (cond) {
    if (((msb16(src1) × lsb16(src2)) << 1) != 8000 0000h),
        ((msb16(src1) × lsb16(src2)) << 1) → dst
    else
        nop
}
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type**

Single-cycle (16 × 16)

**Delay Slots**

1

**See Also**

[MPYHL](#), [SMPY](#), [SMPYH](#), [SMPYHL](#)

**Example**

SMPYHL .M1 A1,A2,A3

**Before instruction**

A1	008A0000h	138 <sup>1</sup>
----	-----------	------------------

**2 cycles after instruction**

A1	008A0000h
----	-----------

A2	<input type="text" value="000000A7h"/>	167 <sup>2</sup>	A2	<input type="text" value="000000A7h"/>	
A3	<input type="text" value="xxxxxxxxh"/>		A3	<input type="text" value="0000B40Ch"/>	46,092
CSR	<input type="text" value="00010100h"/>		CSR	<input type="text" value="00010100h"/>	Not saturated
SSR	<input type="text" value="00000000h"/>		SSR	<input type="text" value="00000000h"/>	

1. Signed 16-MSB integer
2. Signed 16-LSB integer

## 4.271 SMPYLNH

Multiply Signed 16 LSB × Signed 16 MSB With Left Shift and Saturation

**Syntax** **SMPYLNH (.unit) src1, src2, dst**

unit = .M1 or .M2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.M	M3	<a href="#">Figure E-4</a>

**Opcode**

31	29	28	27	23 22				18 17				
<i>creg</i>	<i>z</i>	<i>dst</i>				<i>src2</i>				<i>src1</i>		
3	1	5				5				5		
13	12	11	10	9	8	7	6	5	4	3	2	1 0
<i>src1</i>	<i>x</i>	1	0	0	1	0	0	0	0	0	<i>s</i>	<i>p</i>
5	1											

Opcode map field used...	For operand type...	Unit
<i>src1</i>	slsb16	.M1, .M2
<i>src2</i>	xsmsb16	
<i>dst</i>	sint	

**Description**

The 16 least-significant bits of the *src1* operand is multiplied by the 16 most-significant bits of the *src2* operand. The result is left shifted by 1 and placed in *dst*. If the left-shifted result is 8000 0000h, then the result is saturated to 7FFF FFFFh. If a saturation occurs, the SAT bit in CSR is set one cycle after *dst* is written.

**Execution**

```
if (cond) {
    if (((lsb16(src1) × msb16(src2)) << 1) != 8000 0000h),
        ((lsb16(src1) × msb16(src2)) << 1) → dst
    else 7FFF FFFFh → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Instruction Type**

Single-cycle (16 × 16)

**Delay Slots**

1

**See Also**

[MPYLNH](#), [SMPY](#), [SMPYH](#), [SMPYHL](#)

**Example**

SMPYLNH .M1 A1,A2,A3

**Before instruction**

A1

00008000h

**2 cycles after instruction**

A1

00008000h

A2	<input type="text" value="80000000h"/>	-32,768 <sup>2</sup>	A2	<input type="text" value="80000000h"/>
A3	<input type="text" value="xxxxxxxxh"/>		A3	<input type="text" value="7FFFFFFFh"/> 2,147,483,647
CSR	<input type="text" value="00010100h"/>		CSR	<input type="text" value="00010300h"/> Saturated
SSR	<input type="text" value="00000000h"/>		SSR	<input type="text" value="00000010h"/>

1. Signed 16-MSB integer
2. Signed 16-LSB integer

## 4.272 SMPY2

Multiply Signed by Signed, 16 LSB  $\times$  16 LSB and 16 MSB  $\times$  16 MSB With Left Shift and Saturation

**Syntax** **SMPY2** (.unit) *src1*, *src2*, *dst\_o:dst\_e*

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst		23	22	src2		src1		18	17
3	1			5				5		5		5	
	13	12	11	10	9	8	7	6	5	4	3	2	1 0
	<i>src1</i>	x	0	0	0	0	0	1	1	1	0	0	s p

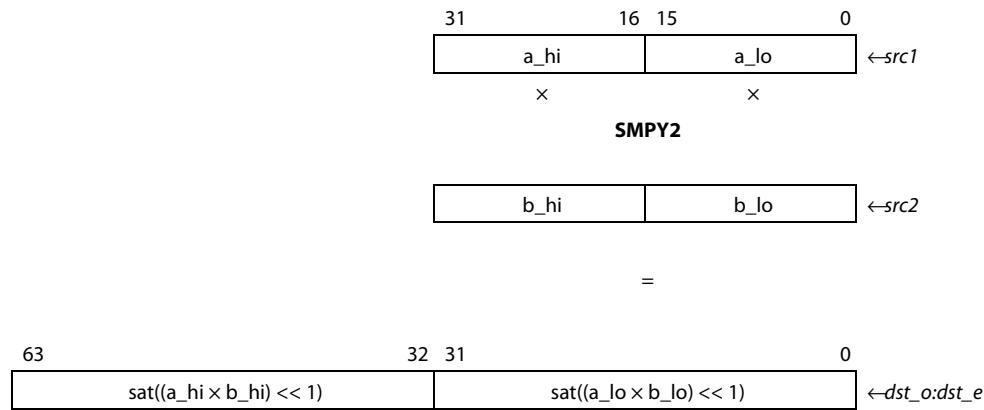
Opcode map field used...	For operand type...	Unit
<i>src1</i>	s2	.M1, .M2
<i>src2</i>	xs2	
<i>dst</i>	sllong	

**Description**

Performs two 16-bit by 16-bit multiplies between two pairs of signed, packed 16-bit values, with an additional left-shift and saturate. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The two 32-bit results are written into a 64-bit register pair.

The **SMPY2** instruction produces two  $16 \times 16$  products. Each product is shifted left by 1. If the left-shifted result is 80000000h, the output value is saturated to 7FFFFFFFh.

The saturated product of the lower halfwords of *src1* and *src2* is written to the even destination register, *dst\_e*. The saturated product of the upper halfwords of *src1* and *src2* is written to the odd destination register, *dst\_o*.



**Note**—If either product saturates, the SAT bit is set in CSR one cycle after the cycle that the result is written to  $dst\_o:dst\_e$ . If neither product saturates, the SAT bit in CSR remains unaffected.

The **SMPY2** instruction helps reduce the number of instructions required to perform two 16-bit by 16-bit saturated multiplies on both the lower and upper halves of two registers.

The following code:

```
SMPY .M1A0, A1, A2
SMPYH .M1A0, A1, A3
```

may be replaced by:

```
SMPY2 .M1A0, A1, A3:A2
```

**Execution**

```
if (cond) {
    sat((lsb16(src1) x lsb16(src2)) << 1) → dst_e;
    sat((msb16(src1) x msb16(src2)) << 1) → dst_o
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read	$src1, src2$			
Written				$dst$
Unit in use		$.M$		

**Instruction Type** Four-cycle

**Delay Slots** 3

**See Also** [MPY2](#), [SMPY](#)

**Examples** [Example 1](#)

```
SMPY2 .M1 A5,A6,A9:A8
```

**Before instruction****4 cycles after instruction**

A5	<table border="1"><tr><td>6A32 1193h</td></tr></table>	6A32 1193h	27186 4499	A5	<table border="1"><tr><td>6A32 1193h</td></tr></table>	6A32 1193h		
6A32 1193h								
6A32 1193h								
A6	<table border="1"><tr><td>B174 6CA4h</td></tr></table>	B174 6CA4h	-20108 27812	A6	<table border="1"><tr><td>B174 6CA4h</td></tr></table>	B174 6CA4h		
B174 6CA4h								
B174 6CA4h								
A9:A8	<table border="1"><tr><td>xxxx xxxxh</td><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh	xxxx xxxxh		A9:A8	<table border="1"><tr><td>BED5 6150h</td><td>0EEA 8C58h</td></tr></table>	BED5 6150h	0EEA 8C58h
xxxx xxxxh	xxxx xxxxh							
BED5 6150h	0EEA 8C58h							

-1,093,312,176      250,252,376

**Example 2**

SMPY2 .M2 B2, B5, B9:B8

**Before instruction****4 cycles after instruction**

B2	<table border="1"><tr><td>1234 3497h</td></tr></table>	1234 3497h	4660 13463	B2	<table border="1"><tr><td>1234 3497h</td></tr></table>	1234 3497h		
1234 3497h								
1234 3497h								
B5	<table border="1"><tr><td>21FF 50A7h</td></tr></table>	21FF 50A7h	8703 20647	B5	<table border="1"><tr><td>21FF 50A7h</td></tr></table>	21FF 50A7h		
21FF 50A7h								
21FF 50A7h								
B9:B8	<table border="1"><tr><td>xxxx xxxxh</td><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh	xxxx xxxxh		B9:B8	<table border="1"><tr><td>04D5 AB98h</td><td>2122 FD02h</td></tr></table>	04D5 AB98h	2122 FD02h
xxxx xxxxh	xxxx xxxxh							
04D5 AB98h	2122 FD02h							

81,111,960      555,941,122

## 4.273 SMPY32

Multiply Signed 32-Bit  $\times$  Signed 32-Bit Into 64-Bit Result With Left Shift and Saturation

**Syntax** **SMPY32** (.unit) *src2*, *src1*, *dst*

unit = .M1 or .M2

**Opcode**

31	30	29	28	27	dst					src2					src1	
0	0	0	1													
5					5					5					5	
13	12	11	10	9	8	7	6	5	4	3	2	1	0	s	p	
src1	x	0	1	1	0	0	1	1	1	0	0	1	1	1	1	1
5		1													1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	int	

**Description**

Performs a 32-bit by 32-bit multiply. *src1* and *src2* are signed 32-bit values. The 64-bit result is shifted left by 1 with saturation, and the 32 most-significant bits of the shifted value are written to *dst*.

If the result saturates either on the multiply or the shift, the M1 or M2 bit in SSR and the SAT bit in CSR are written one cycle after the results are written to *dst*.

This instruction executes unconditionally and cannot be predicated.



**Note**—When both inputs are 8000 0000h, the shifted result cannot be represented as a 32-bit signed value. In this case, the saturation value 7FFF FFFFh is written into *dst*.

**Execution**

`msb32(sat((src2  $\times$  src1) << 1)) → dst`

**Instruction Type**

Four-cycle

**Delay Slots**

3

**See Also**

[MPY32](#), [SMPY2](#)

**Examples**

[Example 1](#)

SMPY32 .M1 A0,A1,A2

**Before instruction**

A0

87654321h

**4 cycle after instruction**

A2

EED8ED1Ah

A1	<input type="text" value="12345678h"/>		
CSR	<input type="text" value="00010100h"/>	CSR <sup>1</sup>	<input type="text" value="00010100h"/>
SSR	<input type="text" value="00000000h"/>	SSR <sup>1</sup>	<input type="text" value="00000000h"/>

1. CSR.SAT and SSR.M1 unchanged by operation

### Example 2

SMPY32 .L1 A0,A1,A2

<b>Before instruction</b>		<b>4 cycles after instruction</b>	
A0	<input type="text" value="80000000h"/>	A2	<input type="text" value="7FFFFFFFh"/>
A1	<input type="text" value="80000000h"/>		
CSR	<input type="text" value="00010100h"/>	CSR <sup>1</sup>	<input type="text" value="00010300h"/>
SSR	<input type="text" value="00000000h"/>	SSR <sup>1</sup>	<input type="text" value="00000010h"/>

1. CSR.SAT and SSR.M1 set to 1, 5 cycles after instruction

## 4.274 SPACK2

Saturate and Pack Two 16 LSBs Into Upper and Lower Register Halves

**Syntax** **SPACK2** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	23 22					18 17						
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>						
3	1			5					5					5	
<i>src1</i>	<i>x</i>	1	1	0	0	1	0	1	1	0	0	<i>s</i>	<i>p</i>	1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.S1, .S2
<i>src2</i>	xint	
<i>dst</i>	s2	

**Description**

Takes two signed 32-bit quantities in *src1* and *src2* and saturates them to signed 16-bit quantities. The signed 16-bit results are then packed into a signed, packed 16-bit format and written to *dst*. Specifically, the saturated 16-bit signed value of *src1* is written to the upper halfword of *dst*, and the saturated 16-bit signed value of *src2* is written to the lower halfword of *dst*.

Saturation is performed on each input value independently. The input values start as signed 32-bit quantities, and are saturated to 16-bit quantities according to the following rules:

- If the value is in the range  $-2^{15}$  to  $2^{15} - 1$ , inclusive, then no saturation is performed and the value is merely truncated to 16 bits.
- If the value is greater than  $2^{15} - 1$ , then the result is set to  $2^{15} - 1$ .
- If the value is less than  $-2^{15}$ , then the result is set to  $-2^{15}$ .

31	16 15	0
00000000 ABCDEFGH	IJKLMNOP QRSTUVWX	← <i>src1</i>

**SPACK2**

00000000 00000000	00YZ1234 56789ABC	← <i>src2</i>
-------------------	-------------------	---------------

↓

31	16 15	0
01111111 11111111	00YZ1234 56789ABC	← <i>dst</i>

The **SPACK2** instruction is useful in code that manipulates 16-bit data at 32-bit precision for its intermediate steps, but that requires the final results to be in a 16-bit representation. The saturate step ensures that any values outside the signed 16-bit range are clamped to the high or low end of the range before being truncated to 16 bits.



**Note**—This operation is performed on each 16-bit value separately. This instruction does not affect the SAT bit in CSR.

**Execution**

```
if (cond) {
    if (src2 > 00007FFFh), 7FFFh → lsb16(dst) or
    if (src2 < FFFF8000h), 8000h → lsb16(dst)
        else truncate(src2) → lsb16(dst);
    if (src1 > 00007FFFh), 7FFFh → msb16(dst) or
    if (src1 < FFFF8000h), 8000h → msb16(dst)
        else truncate(src1) → msb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACK2](#), [PACKH2](#), [PACKHL2](#), [PACKLH2](#), [RPACK2](#), [SPACKU4](#)

**Examples**
**Example 1**

SPACK2 .S1 A2,A8,A9

<b>Before instruction</b>		<b>1 cycle after instruction</b>			
A2	3789 F23Ah	931,787,322	A2	3789 F23Ah	
A8	04B8 4975h	79,186,293	A8	04B8 4975h	
A9	xxxx xxxxh		A9	7FFF 7FFFh	32767 32767

**Example 2**

SPACK2 .S2 B2,B8,B12

<b>Before instruction</b>		<b>1 cycle after instruction</b>		
B2	A124 2451h	-1,591,466,927	B2	A124 2451h
B8	01A6 A051h	27,697,233	B8	01A6 A051h

B12 B12  -32768 32767

## 4.275 SPACKU4

Saturate and Pack Four Signed 16-Bit Integers Into Four Unsigned 8-Bit Halfwords

**Syntax** **SPACKU4** (.unit) *src1*, *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27	23 22						18 17			
<i>creg</i>	<i>z</i>			<i>dst</i>						<i>src2</i>		<i>src1</i>	
3	1			5						5			5
<i>src1</i>	<i>x</i>	1	1	0	1	0	0	1	1	0	0	<i>s</i>	<i>p</i>
5	1											1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	<i>s2</i>	.S1, .S2
<i>src2</i>	<i>xs2</i>	
<i>dst</i>	<i>u4</i>	

**Description**

Takes four signed 16-bit values and saturates them to unsigned 8-bit quantities. The values in *src1* and *src2* are treated as signed, packed 16-bit quantities. The results are written into *dst* in an unsigned, packed 8-bit format.

Each signed 16-bit quantity in *src1* and *src2* is saturated to an unsigned 8-bit quantity as described below. The resulting quantities are then packed into an unsigned, packed 8-bit format. Specifically, the upper halfword of *src1* is used to produce the most-significant byte of *dst*. The lower halfword of *src1* is used to produce the second most-significant byte (bits 16 to 23) of *dst*. The upper halfword of *src2* is used to produce the third most-significant byte (bits 8 to 15) of *dst*. The lower halfword of *src2* is used to produce the least-significant byte of *dst*.

Saturation is performed on each signed 16-bit input independently, producing separate unsigned 8-bit results. For each value, the following tests are applied:

- If the value is in the range 0 to  $2^8 - 1$ , inclusive, then no saturation is performed and the result is truncated to 8 bits.
- If the value is greater than  $2^8 - 1$ , then the result is set to  $2^8 - 1$ .
- If the value is less than 0, the result is cleared to 0.

31	16 15	0
00000000 ABCDEFGH	00000000 IJKLMNOP	← <i>src1</i>

**SPACKU4**

00000000 YZ123456	11111111 QRSTUVWX	← <i>src2</i>
-------------------	-------------------	---------------



31	24 23	16 15	8 7	0
ABCDEFGH	FFFFFFF	YZ123456	00000000	←dst

The **SPACKU4** instruction is useful in code that manipulates 8-bit data at 16-bit precision for its intermediate steps, but that requires the final results to be in an 8-bit representation. The saturate step ensures that any values outside the unsigned 8-bit range are clamped to the high or low end of the range before being truncated to 8 bits.



**Note**—This operation is performed on each 8-bit quantity separately. This instruction does not affect the SAT bit in CSR.

#### Execution

```
if (cond) {
    if (msb16(src1) >= 000000FFh), FFh → ubyte3(dst) or
    if (msb16(src1) << 0), 0 → ubyte3(dst)
        else truncate(msb16(src1)) → ubyte3(dst);
    if (lsb16(src1) >> 000000FFh), FFh → ubyte2(dst) or
    if (lsb16(src1) << 0), 0 → ubyte2(dst)
        else truncate(lsb16(src1)) → ubyte2(dst);
    if (msb16(src2) >> 000000FFh), FFh → ubyte1(dst) or
    if (msb16(src2) << 0), 0 → ubyte1(dst)
        else truncate(msb16(src2)) → ubyte1(dst);
    if (lsb16(src2) >> 000000FFh), FFh → ubyte0(dst) or
    if (lsb16(src2) <= 0), 0 → ubyte0(dst)
        else truncate(lsb16(src2)) → ubyte0(dst)
    }
else nop
```

#### Pipeline

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [PACKH4](#), [PACKL4](#), [SPACK2](#)

#### Examples

##### Example 1

SPACKU4 .S1 A2,A8,A9

Before instruction			1 cycle after instruction		
A2	3789 F23Ah	14217 -3526	A2	3789 F23Ah	
A8	04B8 4975h	1208 18805	A8	04B8 4975h	
A9	xxxx xxxxh		A9	FF 00 FF FFh	255 0 255 255

##### Example 2

SPACKU4 .S2 B2,B8,B12

<b>Before instruction</b>		<b>1 cycle after instruction</b>	
B2	A124 2451h	-24284 9297	B2 A124 2451h
B8	01A6 A051h	422 -24495	B8 01A6 A051h
B12	xxxx xxxxh		B12 00 FF FF 00h 0 255 255 0

## 4.276 SPDP

Convert Single-Precision Floating-Point Value to Double-Precision Floating-Point Value

**Syntax** **SPDP** (.unit) *src2*, *dst*

unit = .S1 or .S2

**Opcode**

31	29	28	27						23	22						18	17	16
				<i>dst</i>							<i>src2</i>					0	0	
3		1						5						5				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xsp	.S1, .S2
<i>dst</i>	dp	

**Description** The single-precision value in *src2* is converted to a double-precision value and placed in *dst*.



**Note—**

- 1) If *src2* is SNaN, NaN\_out is placed in *dst* and the INVAL and NAN2 bits are set.
- 2) If *src2* is QNaN, NaN\_out is placed in *dst* and the NAN2 bit is set.
- 3) If *src2* is a signed denormalized number, signed 0 is placed in *dst* and the INEX and DEN2 bits are set.
- 4) If *src2* is signed infinity, INFO bit is set.
- 5) No overflow or underflow can occur.

**Execution**

```
if (cond) dp(src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2</i>	
Written	<i>dst_l</i>	<i>dst_h</i>
Unit in use	. <i>S</i>	

If *dst* is used as the source for the **ADDDP**, **CMPEQDP**, **CMPLTDP**, **CMPGTDP**, **MPYDP**, or **SUBDP** instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

<b>Instruction Type</b>	2-cycle DP
<b>Delay Slots</b>	1
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">DPSP</a> , <a href="#">INTDP</a> , <a href="#">SPINT</a> , <a href="#">SPTRUNC</a>
<b>Example</b>	<code>SPDP .S1X B2,A1:A0</code>

<b>Before instruction</b>		<b>2 cycles after instruction</b>						
B2	<table border="1"><tr><td>4109 999Ah</td></tr></table>	4109 999Ah	B2	<table border="1"><tr><td>4109 999Ah</td></tr></table>	4109 999Ah	8.6		
4109 999Ah								
4109 999Ah								
A1:A0	<table border="1"><tr><td>xxxx xxxxh</td><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh	xxxx xxxxh	A1:A0	<table border="1"><tr><td>4021 3333h</td><td>4000 0000h</td></tr></table>	4021 3333h	4000 0000h	8.6
xxxx xxxxh	xxxx xxxxh							
4021 3333h	4000 0000h							

## 4.277 SPINT

Convert Single-Precision Floating Point to Integer

**Syntax**    **SPINT (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, or .S2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23 22	18 17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2	src1	x		opfield	1	1	0	s	p

3                                 5                                 5                                 7

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dst	.L1 or .L2	0001010

**Opcode**    Opcode for .S Unit, 1 src

31	29	28	27	23 22	18 17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2	opfield	x	1	1	1	1	0	0	1	0	0	0	s	p

3                                 5                                 5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dst	.S1 or .S2	00100

**Description**    The single precision value in src2 is converted to an integer and placed in dst.

**Execution**

```
if(cond)
{
    int(src2) -> dst
}
else nop
```

**Instruction Type**    4-cycle

**Delay Slots**    3

**Functional Unit Latency**    1

**See Also**    [DPINT](#), [DPTRUNC](#), [INTDP](#), [INTDPU](#), [INTSP](#), [INTSPU](#), [SPTRUNC](#)

**Example**

```
FADCR== 0x00000000
A1    == 0x4109999a
SPINT .L A1,A0
A0    == 0x00000009

FADCR= 0x00000080

FADCR== 0x00000200
A1    == 0x4109999a
SPINT .L A1,A0
A0    == 0x00000008

FADCR= 0x00000280

FADCR== 0x00000400
A1    == 0x4109999a
SPINT .L A1,A0
A0    == 0x00000009

FADCR= 0x00000480

FADCR== 0x00000600
A1    == 0x4109999a
SPINT .L A1,A0
A0    == 0x00000008

FADCR= 0x00000680
```

## 4.278 SPKERNEL

Software Pipelined Loop (SPLOOP) Buffer Operation Code Boundary

**Syntax**

**SPKERNEL** (*fstg,fcyc*)

unit = none

**Compact Instruction Format**

Unit	Opcode Format	Figure
none	Uspk	<a href="#">Figure H-5</a>

**Opcode**

31	30	29	28	27	22	21	20	19	18	17	16
0	0	0	0		<i>fstg/fcyc</i>				0	0	0
6											
15	14	13	12	11	10	9	8	7	6	5	4
0	1	0	0	0	0	0	0	0	0	0	0
1											
0											

**Description**

The **SPKERNEL** instruction is placed in parallel with the last execute packet of the SPLOOP code body indicating there are no more instructions to load into the loop buffer. The **SPKERNEL** instruction also controls at what point in the epilog the execution of post-SPLOOP instructions begins. This point is specified in terms of stage and cycle counts, and is derived from the *fstg/fcyc* field.

The stage and cycle values for both the post-SPLOOP fetch and reload cases are derived from the *fstg/fcyc* field. The 6-bit field is interpreted as a function of the *ii* value from the associated **SPLOOP(D)** instruction. The number of bits allocated to stage and cycle vary according to *ii*. The value for cycle starts from the least-significant end; the value for stage starts from the most-significant end, and they grow together. The number of epilog stages and the number of cycles within those stages are shown in [Table 4-11](#) on page 4-603. The exact bit allocation to stage and cycle is shown in [Table 4-12](#) on page 4-603.

The following restrictions apply to the use of the **SPKERNEL** instruction:

- The **SPKERNEL** instruction must be the first instruction in the execute packet containing it.
- The **SPKERNEL** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP n** ( $n > 1$ ), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPKERNEL** instruction cannot be placed in the execute packet immediately following an execute packet containing any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP n** ( $n > 1$ ), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPKERNEL** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.

- The **SPKERNEL** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR**, **SPLOOP**, **SPLOOPD**, or **SPLOOPW** instructions.
- When the **SPKERNEL** instruction is used with the **SPLOOPW** instruction, *fstg* and *fcyc* should both be zero.

 **Note**—The delay specified by the **SPKERNEL** *fstg/fcyc* parameters will not extend beyond the end of the kernel epilog. If the end of the kernel epilog is reached prior to the end of the delay specified by *fstg/fcyc* parameters due to either an excessively large value specified for parameters or due to an early exit from the loop, program fetch will begin immediately and the value specified by the *fstg/fcyc* will be ignored

**Table 4-11 Field Allocation in stg/cyc Field**

ii	Number of Bits for Stage	Number of Bits for Cycle
1	6	0
2	5	1
3-4	4	2
5-8	3	3
9-14	2	4

**Table 4-12 Bit Allocations to Stage and Cycle in stg/cyc Field**

ii	stg/cyc[5]	stg/cyc[4]	stg/cyc[3]	stg/cyc[2]	stg/cyc[1]	stg/cyc[0]
1	stage[0]	stage[1]	stage[2]	stage[3]	stage[4]	stage[5]
2	stage[0]	stage[1]	stage[2]	stage[3]	stage[4]	cycle[0]
3-4	stage[0]	stage[1]	stage[2]	stage[3]	cycle[1]	cycle[0]
5-8	stage[0]	stage[1]	stage[2]	cycle[2]	cycle[1]	cycle[0]
9-14	stage[0]	stage[1]	cycle[3]	cycle[2]	cycle[1]	cycle[0]

**Execution** See [Chapter 8](#) on page 8-1 for more information

**See Also** [SPKERNEL](#)

## 4.279 SPKERNELR

Software Pipelined Loop (SPLOOP) Buffer Operation Code Boundary

**Syntax** **SPKERNELR**

unit = none

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	0	0	0	0	0	0	s	p

**Description** The **SPKERNELR** instruction is placed in parallel with the last execute packet of the SPLOOP code body indicating there are no more instructions to load into the loop buffer. The **SPKERNELR** instruction also indicates that the execution of both post-SPLOOP instructions and instructions reloaded from the buffer begin in the first cycle of the epilog.

The following restrictions apply to the use of the **SPKERNELR** instruction:

- The **SPKERNELR** instruction must be the first instruction in the execute packet containing it.
- The **SPKERNELR** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP**  $n$  ( $n > 1$ ), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPKERNELR** instruction cannot be placed in the execute packet immediately following an execute packet containing any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP**  $n$  ( $n > 1$ ), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPKERNELR** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPKERNELR** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR**, **SPLOOP**, **SPLOOPD**, or **SPLOOPW** instructions.
- The **SPKERNELR** instruction can only be used when the **SPLOOP** instruction that began the SPLOOP buffer operation was predicated.
- The **SPKERNELR** instruction cannot be paired with an **SPLOOPW** instruction.

This instruction executes unconditionally and cannot be predicated.

**Execution** See [Chapter 8](#) on page 8-1 for more information.

**See Also** [SPKERNEL](#)

## 4.280 SPLOOP

Software Pipelined Loop (SPLOOP) Buffer Operation

### Syntax

**SPLOOP *ii***

unit = none

#### Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspl	<a href="#">Figure H-3</a>

#### Opcode

31	29	28	27					23	22	21	20	19	18	17	16
				<i>creg</i>	<i>z</i>		<i>ii - 1</i>		0	0	0	0	0	1	1
3		1					5								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>

#### Description

The **SPLOOP** instruction invokes the loop buffer mechanism. See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more details.

When the **SPLOOP** instruction is predicated, it indicates that the loop is a nested loop using the SPLOOP reload capability. The decision of whether to reload is determined by the predicate register selected by the *creg* and *z* fields.

The following restrictions apply to the use of the **SPLOOP** instruction:

- The **SPLOOP** instruction must be the first instruction in the execute packet containing it.
- The **SPLOOP** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **CALLP**, **NOP *n*** (*n* > 1), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPLOOP** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPLOOP** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR**, **SPKERNEL**, or **SPKERNELR** instructions.

#### Execution

See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more information.

#### See Also

[SPLOOPD](#), [SPLOOPW](#)

## 4.281 SPLOOPD

Software Pipelined Loop (SPLOOP) Buffer Operation With Delayed Testing

### Syntax

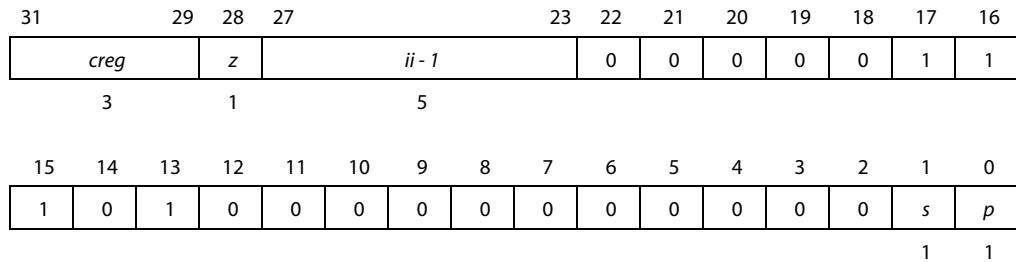
**SPLOOPD *ii***

unit = none

#### Compact Instruction Format

Unit	Opcode Format	Figure
none	Uspl	<a href="#">Figure H-3</a>
	Uspldr	<a href="#">Figure H-4</a>

#### Opcode



#### Description

The **SPLOOPD** instruction invokes the loop buffer mechanism. The testing of the termination condition is delayed for four cycles. See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more details.

When the **SPLOOPD** instruction is predicated, it indicates that the loop is a nested loop using the SPLOOP reload capability. The decision of whether to reload is determined by the predicate register selected by the *creg* and *z* fields.

The following restrictions apply to the use of the **SPLOOPD** instruction:

- The **SPLOOPD** instruction must be the first instruction in the execute packet containing it.
- The **SPLOOPD** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes BNOP, CALLP, NOP *n* (*n* > 1), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPLOOPD** instruction cannot be placed in parallel with DINT or RINT instructions.
- The **SPLOOPD** instruction cannot be placed in parallel with SPMASK, SPMASKR, SPKERNEL, or SPKERNELR instructions.

#### Execution

See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more information.

#### See Also

[SPLOOP](#), [SPLOOPW](#)

## 4.282 SPLOOPW

Software Pipelined Loop (SPLOOP) Buffer Operation With Delayed Testing and No Epilog

**Syntax**    **SPLOOPW** *ii*

unit = none

**Opcode**

31	29	28	27						23	22	21	20	19	18	17	16
				<i>creg</i>	<i>z</i>		<i>ii - 1</i>		0	0	0	0	0	0	1	1
	3				1			5								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	0	0	0	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>	
														1	1	

**Description**

The **SPLOOPW** instruction invokes the loop buffer mechanism. The testing of the termination condition is delayed for four cycles. See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more details.

The **SPLOOPW** instruction is always predicated. The termination condition is the value of the predicate register selected by the *creg* and *z* fields.

The following restrictions apply to the use of the **SPLOOPW** instruction:

- The **SPLOOPW** instruction must be the first instruction in the execute packet containing it.
- The **SPLOOPW** instruction cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs. This includes **BNOP**, **NOP** *n* (*n* > 1), and protected loads (see compact instruction discussion in “[Compact Instructions on the CPU](#)” on page 3-29).
- The **SPLOOPW** instruction cannot be placed in parallel with **DINT** or **RINT** instructions.
- The **SPLOOPW** instruction cannot be placed in parallel with **SPMASK**, **SPMASKR**, **SPKERNEL**, or **SPKERNELR** instructions.

**Execution**

See Chapter 8 “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more information.

**See Also**

[SPLOOP](#), [SPLOOPD](#)

## 4.283 SPMASK

Software Pipelined Loop (SPLOOP) Buffer Operation Load/Execution Control

**Syntax**

**SPMASK unitmask**

unit = none

**Compact Instruction Format**

Unit	Opcode Format	Figure
none	Uspm	<a href="#">Figure H-6</a>

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	M2	M1	D2	D1	S2	S1	L2	L1	1	1
								1	1	1	1	1	1	1	1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	s	p
														1	1

**Description**

The **SPMASK** instruction serves two purposes within the SPLOOP mechanism:

1. The **SPMASK** instruction inhibits the execution of specified instructions from the buffer within the current execute packet.
2. The **SPMASK** inhibits the loading of specified instructions into the buffer during loading phase, although the instruction will execute normally.

If the SPLOOP is reloading after returning from an interrupt, the **SPMASKed** instructions coming from the buffer execute, but the **SPMASKed** instructions from program memory do not execute and are not loaded into the buffer.

An **SPMASKed** instruction encountered outside of the SPLOOP mechanism shall be treated as a NOP.

The **SPMASKed** instruction must be the first instruction in the execute packet containing it.

The **SPMASK** instruction cannot be placed in parallel with **SPLOOP**, **SPLOOPD**, **SPKERNEL**, or **SPKERNELR** instructions.

The **SPMASK** instruction executes unconditionally and cannot be predicated.

There are two ways to specify which instructions within the current execute packet will be masked:

1. The functional units of the instruction can be specified as the SPMASK argument.
2. The instruction to be masked can be marked with a caret (^) in the instruction code. The following three examples are equivalent:

SPMASKD2,L1		
	MV.D2	B0,B1
	MV.L1	A0,A1

```
SPMASKD2
|| MV.D2    B0,B1
|| ^ MV.L1    A0,A1
```

```
SPMASK
|| ^ MV.D2    B0,B1
|| ^ MV.L1    A0,A1
```

The following two examples mask two **MV** instructions, but do not mask the **MPY** instruction.

```
|| SPMASK D1, D2
|| MV.D1    A0,A1 ;This unit is SPMASKED
|| MV.D2    B0,B1 ;This unit is SPMASKED
|| MPY.L1    A0,B1 ;This unit is NOT SPMASKED
```

```
|| ^ SPMASK
|| ^ MV.D1    A0,A1 ;This unit is SPMASKED
|| ^ MV.D2    B0,B1 ;This unit is SPMASKED
|| ^ MPY.L1    A0,B1 ;This unit is NOT SPMASKED
```

**Execution** See Chapter 8 “Software Pipelined Loop (SPLOOP) Buffer” on page 8-1

**See Also** [SPMASKR](#)

## 4.284 SPMASKR

Software Pipelined Loop (SPLOOP) Buffer Operation Load/Execution Control

**Syntax**

**SPMASKR unitmask**

unit = none

**Compact Instruction Format**

Unit	Opcode Format	Figure
none	Uspm	<a href="#">Figure H-6</a>

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	M2	M1	D2	D1	S2	S1	L2	L1	1	1
						1	1	1	1	1	1	1	1	1	1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	s	p
														1	1

**Description**

The **SPMASKR** instruction serves three purposes within the SPLOOP mechanism. Similar to the **SPMASK** instruction:

1. The **SPMASKR** instruction inhibits the execution of specified instructions from the buffer within the current execute packet.
2. The **SPMASKR** instruction inhibits the loading of specified instructions into the buffer during loading phase, although the instruction will execute normally.

In addition to the functionality of the **SPMASK** instruction:

3. The **SPMASKR** instruction controls the reload point for nested loops.

The **SPMASKR** instruction is placed in the execute packet (in the post-SPKERNEL code) preceding the execute packet that will overlap with the first cycle of the reload operation.

The **SPKERNELR** and the **SPMASKR** instructions cannot coexist in the same SPLOOP operation. In the case where reload is intended to start in the first epilog cycle, the **SPKERNELR** instruction is used and the **SPMASKR** instruction is not used for that nested loop.

The **SPMASKR** instruction cannot be used in a loop using the **SPLOOPW** instruction.

An **SPMASKR** instruction encountered outside of the SPLOOP mechanism shall be treated as a NOP.

The **SPMASKR** instruction executes unconditionally and cannot be predicated.

The **SPMASKR** instruction must be the first instruction in the execute packet containing it.

The **SPMASKR** instruction cannot be placed in parallel with **SPLOOP**, **SPLOOPD**, **SPKERNEL**, or **SPKERNELR** instructions.

There are two ways to specify which instructions within the current execute packet will be masked:

1. The functional units of the instruction can be specified as the SPMASKR argument.
2. The instruction to be masked can be marked with a caret (^) in the instruction code. The following three examples are equivalent:

```
SPMASKR D2,L1
|| MV.D2    B0,B1
|| MV.L1    A0,A1
```

```
SPMASKR
|| MV.D2    B0,B1
|| ^MV.L1   A0,A1
```

```
SPMASKR
|| ^MV.D2   B0,B1
|| ^MV.L1   A0,A1
```

The following two examples mask two **MV** instructions, but do not mask the **MPY** instruction. The presence of a caret (^) in the instruction code specifies which instructions are **SPMASKed**.

```
SPMASKRD1,D2
|| MV .D1    A0,A1 ;This unit is SPMASKED
|| MV .D2    B0,B1 ;This unit is SPMASKED
|| MPY .L1   A0,B1 ;This unit is NOT SPMASKED

SPMASKR
|| ^ MV .D1    A0,A1 ;This unit is SPMASKED
|| ^ MV .D2    B0,B1 ;This unit is SPMASKED
|| MPY .L1   A0,B1 ;This unit is NOT SPMASKED
```

**Execution** See Chapter 8 “Software Pipelined Loop (**SPLOOP**) Buffer” on page 8-1

**See Also** [SPMASK](#)

**Example**

```
SPMASKR
|| ^ LDW .D1 *A0,A1 ;This unit is SPMASKed
|| ^ LDW .D2 *B0,B1 ;This unit is SPMASKed
|| MPY .M1 A3,A4,A5;This unit is NOT SPMASKed
```

## 4.285 SPTRUNC

Convert Single-Precision Floating-Point Value to Integer With Truncation

**Syntax** **SPTRUNC (.unit) src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	dst					src2					18	17	16
														0	0	
3		1					5						5			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	x	0	0	0	1	0	1	1	1	1	0	s	p	
							1							1	1	

Opcode map field used...	For operand type...	Unit
src2 dst	xsp sint	.L1, .L2

**Description**

The single-precision value in *src2* is converted to an integer and placed in *dst*. This instruction operates like **SPINT** except that the rounding modes in the floating-point adder configuration register (FADCR) are ignored, and round toward zero (truncate) is always used.



**Note—**

- 1) If *src2* is NaN, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INVAL bit is set.
- 2) If *src2* is signed infinity or if overflow occurs, the maximum signed integer (7FFF FFFFh or 8000 0000h) is placed in *dst* and the INEX and OVER bits are set. Overflow occurs if *src2* is greater than  $2^{31}-1$  or less than  $-2^{31}$ .
- 3) If *src2* is denormalized, 0000 0000h is placed in *dst* and INEX and DEN2 bits are set.

4) If rounding is performed, the INEX bit is set.

**Execution**

```
if (cond) int (src2) → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4
Read		src2		
Written				dst
Unit in use	.L			

**Instruction Type** Four-cycle

**Delay Slots** 3

**Functional Unit Latency** 1

**See Also** [DPTRUNC](#), [SPDP](#), [SPINT](#)

**Example** SPTRUNC .L1X B1,A2



## 4.286 **SSHL**

Shift Left With Saturation

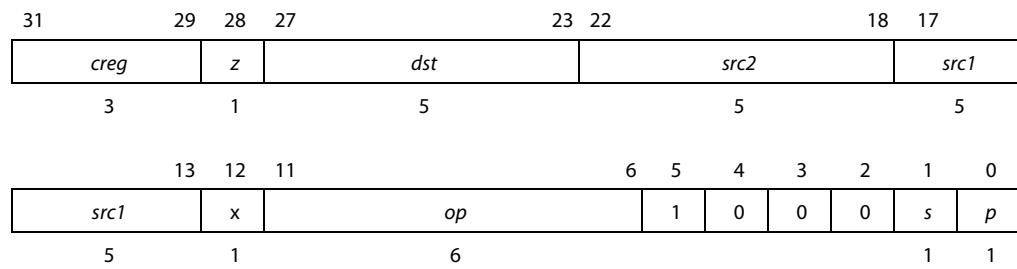
**Syntax**    **SSHL** (.unit) *src2*, *src1*, *dst*

unit = .S1 or .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.S	Ssh5 S2sh	<a href="#">Figure F-20</a> <a href="#">Figure F-21</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i> <i>src1</i> <i>dst</i>	xsint uint sint	.S1, .S2	100011
<i>src2</i> <i>src1</i> <i>dst</i>	xsint ucst5 sint	.S1, .S2	100010

**Description** The *src2* operand is shifted to the left by the *src1* operand. The result is placed in *dst*. When a register is used to specify the shift, the 5 least-significant bits specify the shift amount. Valid values are 0 through 31, and the result of the shift is invalid if the shift amount is greater than 31. The result of the shift is saturated to 32 bits. If a saturate occurs, the SAT bit in CSR is set one cycle after *dst* is written.



**Note**—When a register is used to specify the shift, the 6 least-significant bits specify the shift amount. Valid values are 0 through 63. If the shift count value is greater than 32, then the result is saturated to 32 bits when *src2* is non-zero.

**Execution**

```
if (cond) {
    if (bit(31) through bit(31 - src1) of src2 are all 1s or all 0s),
        dst = src2 << src1;
    else if (src2 > 0), saturate dst to 7FFFFFFFh;
    else if (src2 < 0), saturate dst to 80000000h
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.S

**Instruction Type** Single-cycle

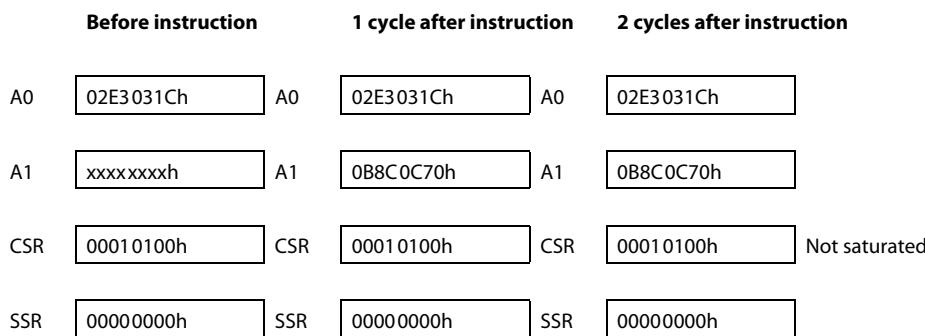
**Delay Slots** 0

**See Also** [ROTL](#), [SHL](#), [SHLMB](#), [SHR](#), [SSHVL](#)

**Examples**

**Example 1**

SSHLL .S1 A0,2,A1



### Example 2

*SSHL .S1 A0,A1,A2*

	<b>Before instruction</b>	<b>1 cycle after instruction</b>		<b>2 cycles after instruction</b>	
A0	47191925h	A0	47191925h	A0	47191925h
A1	00000006h	A1	00000006h	A1	00000006h
A2	xxxxxxxxh	A2	7FFFFFFFh	A2	7FFFFFFFh
CSR	00010100h	CSR	00010100h	CSR	00010300h Saturated
SSR	00000000h	SSR	00000000h	SSR	00000004h

## 4.287 SSHVL

Variable Shift Left

**Syntax**    **SSHVL (.unit) src2, src1, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst					src2					src1		
				3	1			5						5		5
				13	12	11	10	9	8	7	6	5	4	3	2	1
				src1	x	0	1	1	1	0	0	1	1	0	0	s
																p
				5	1											1

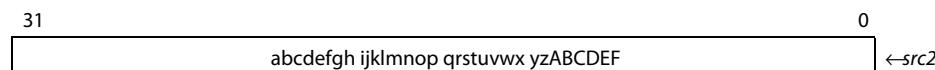
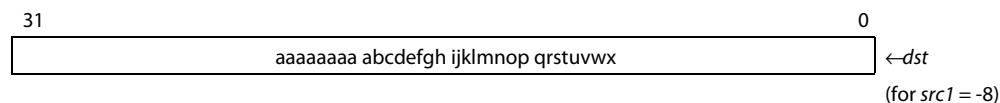
Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	int	

**Description**    Shifts the signed 32-bit value in *src2* to the left or right by the number of bits specified by *src1*, and places the result in *dst*.

The *src1* argument is treated as a 2s-complement shift value which is automatically limited to the range -31 to 31. If *src1* is positive, *src2* is shifted to the left. If *src1* is negative, *src2* is shifted to the right by the absolute value of the shift amount, with the sign-extended shifted value being placed in *dst*. It should also be noted that when *src1* is negative, the bits shifted right past bit 0 are lost.

Saturation is performed when the value is shifted left under the following conditions:

- If the shifted value is in the range  $-2^{31}$  to  $2^{31} - 1$ , inclusive, then no saturation is performed, and the result is truncated to 32 bits.
- If the shifted value is greater than  $2^{31} - 1$ , then the result is saturated to  $2^{31} - 1$ .
- If the shifted value is less than  $-2^{31}$ , then the result is saturated to  $-2^{31}$ .

**SSHVL**

**Note**—If the shifted value is saturated, then the SAT bit is set in CSR one cycle after the result is written to dst. If the shifted value is not saturated, then the SAT bit is unaffected.

**Execution**

```
if (cond) {
    if (0 <= src1 <= 31), sat(src2 << src1) → dst ;
    if (-31 <= src1 < 0), (src2 >> abs(src1)) → dst;
    if (src1 > 31), sat(src2 << 31) → dst;
    if (src1 < -31), (src2 >> 31) → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	src1, src2	
Written		dst
Unit in use	.M	

**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [SHL](#), [SHLMB](#), [SSH](#), [SSHVR](#)

**Examples****Example 1**

SSHVL .M2 B2, B4, B5

**Before instruction****2 cycles after instruction**

B2		B2	
B4	-31	B4	
B5		B5	

**Example 2**

**SSHVL .M1 A2,A4,A5**

<b>Before instruction</b>		<b>2 cycles after instruction</b>			
A2	<table border="1"><tr><td>F14C 2108h</td></tr></table>	F14C 2108h	A2	<table border="1"><tr><td>F14C 2108h</td></tr></table>	F14C 2108h
F14C 2108h					
F14C 2108h					
A4	<table border="1"><tr><td>0000 0001Fh</td></tr></table>	0000 0001Fh	31	<table border="1"><tr><td>0000 0001Fh</td></tr></table>	0000 0001Fh
0000 0001Fh					
0000 0001Fh					
A5	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh	A5	<table border="1"><tr><td>8000 0000h</td></tr></table> Saturated to most negative value	8000 0000h
xxxx xxxxh					
8000 0000h					

### Example 3

**SSHVL .M2 B12, B24, B25**

<b>Before instruction</b>		<b>2 cycles after instruction</b>			
B12	<table border="1"><tr><td>187A 65FCh</td></tr></table>	187A 65FCh	B12	<table border="1"><tr><td>187A 65FCh</td></tr></table>	187A 65FCh
187A 65FCh					
187A 65FCh					
B24	<table border="1"><tr><td>FFFF FFFFh</td></tr></table>	FFFF FFFFh	-1	<table border="1"><tr><td>FFFF FFFFh</td></tr></table>	FFFF FFFFh
FFFF FFFFh					
FFFF FFFFh					
B25	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh	B25	<table border="1"><tr><td>03CD 32FEh</td></tr></table>	03CD 32FEh
xxxx xxxxh					
03CD 32FEh					

## 4.288 SSHVR

Variable Shift Right

**Syntax**    **SSHVR** (.unit) *src2*, *src1*, *dst*

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst					src2					src1		18	17
3		1						5						5			5
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
	<i>src1</i>	x	0	1	1	1	0	1	1	1	0	0	s	p		1	1
		5		1													

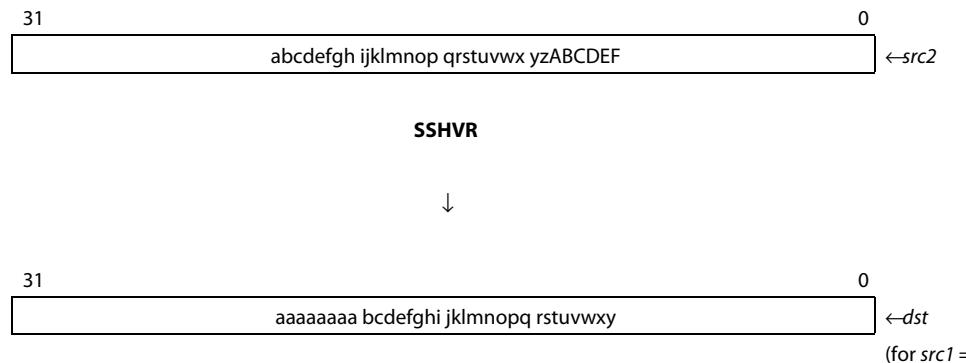
Opcode map field used...	For operand type...	Unit
<i>src1</i>	int	.M1, .M2
<i>src2</i>	xint	
<i>dst</i>	int	

**Description**    Shifts the signed 32-bit value in *src2* to the left or right by the number of bits specified by *src1*, and places the result in *dst*.

The *src1* argument is treated as a 2s-complement shift value that is automatically limited to the range -31 to 31. If *src1* is positive, *src2* is shifted to the right by the value specified with the sign-extended shifted value being placed in *dst*. It should also be noted that when *src1* is positive, the bits shifted right past bit 0 are lost. If *src1* is negative, *src2* is shifted to the left by the absolute value of the shift amount value and the result is placed in *dst*.

Saturation is performed when the value is shifted left under the following conditions:

- If the shifted value is in the range  $-2^{31}$  to  $2^{31} - 1$ , inclusive, then no saturation is performed, and the result is truncated to 32 bits.
- If the shifted value is greater than  $2^{31} - 1$ , then the result is saturated to  $2^{31} - 1$ .
- If the shifted value is less than  $-2^{31}$ , then the result is saturated to  $-2^{31}$ .



**Note**—If the shifted value is saturated, then the SAT bit is set in CSR one cycle after the result is written to dst. If the shifted value is not saturated, then the SAT bit is unaffected.

**Execution**

```
if (cond) {
    if (0 <= src1 <= 31), (src2 >> src1) → dst;
    if (-31 <= src1 < 0), sat(src2 << abs(src1)) → dst;
    if (src1 > 31), (src2 >> 31) → dst;
    if (src1 < -31), sat(src2 << 31) → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

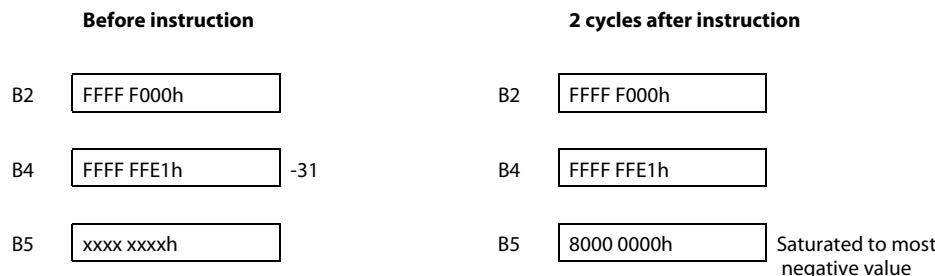
**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [SHR](#), [SHR2](#), [SHRMB](#), [SHRU](#), [SHRU2](#), [SSHVL](#)

**Examples** [Example 1](#)

SSHVR .M2 B2,B4,B5



**Example 2**

SSHVR .M1 A2,A4,A5

**Before instruction**

A2	F14C 2108h
----	------------

A4	0000 0001Fh	31
----	-------------	----

A5	xxxx xxxxh
----	------------

**2 cycles after instruction**

A2	F14C 2108h
----	------------

A4	0000 0001Fh
----	-------------

A5	FFFF FFFFh
----	------------

**Example 3**

SSHVR .M2 B12, B24, B25

**Before instruction**

B12	187A 65FCh
-----	------------

B24	FFFF FFFFh	-1
-----	------------	----

B25	xxxx xxxxh
-----	------------

**2 cycles after instruction**

B12	187A 65FCh
-----	------------

B24	FFFF FFFFh
-----	------------

B25	30F4 CBF8h
-----	------------

## 4.289 SSUB

Subtract Two Signed Integers With Saturation

**Syntax**    **SSUB (.unit) src1, src2, dst**

or

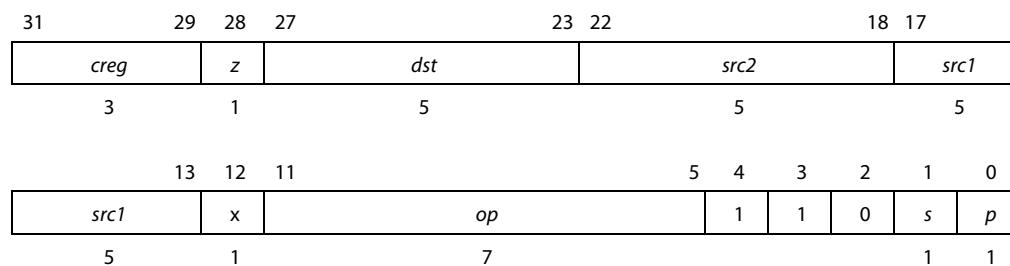
**SSUB (.unit) src1, src2\_h:src2\_l, dst\_h:dst\_l**

unit = .L1 or .L2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L3	<a href="#">Figure D-4</a>

**Opcode**



Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sint	.L1, .L2	0001111
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	xsint	.L1, .L2	0011111
<i>src2</i>	sint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.L1, .L2	0001110
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.L1, .L2	0101100
<i>src2</i>	slong		
<i>dst</i>	slong		

**Description**

*src2* is subtracted from *src1* and is saturated to the result size according to the following rules:

1. If the result is an int and  $src1 - src2 > 2^{31} - 1$ , then the result is  $2^{31} - 1$ .
2. If the result is an int and  $src1 - src2 < -2^{31}$ , then the result is  $-2^{31}$ .
3. If the result is a long and  $src1 - src2 > 2^{39} - 1$ , then the result is  $2^{39} - 1$ .
4. If the result is a long and  $src1 - src2 < -2^{39}$ , then the result is  $-2^{39}$ .

The result is placed in *dst*. If a saturate occurs, the SAT bit in CSR is set one cycle after *dst* is written.

**Execution**  
`if (cond) src1 -s src2 → dst  
else nop`

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SUB](#), [SSUB2](#)

**Examples** [Example 1](#)

SSUB .L2 B1,B2,B3

**Before instruction**

B1	<input type="text" value="5A2E51A3h"/>	1,512,984,995
B2	<input type="text" value="802A3FA2h"/>	-2,144,714,846
B3	<input type="text" value="xxxxxxxxh"/>	
CSR	<input type="text" value="00010100h"/>	
SSR	<input type="text" value="00000000h"/>	

**1 cycle after instruction**

B1	<input type="text" value="5A2E51A3h"/>
B2	<input type="text" value="802A3FA2h"/>
B3	<input type="text" value="7FFFFFFFh"/>
CSR	<input type="text" value="00010100h"/>
SSR	<input type="text" value="00000000h"/>

**2 cycles after instruction**

B1	<input type="text" value="5A2E51A3h"/>
B2	<input type="text" value="802A3FA2h"/>
B3	<input type="text" value="7FFFFFFFh"/>
CSR	<input type="text" value="00010300h"/> Saturated
SSR	<input type="text" value="00000002h"/>

**Example 2**

SSUB .L1 A0,A1,A2

**Before instruction**

A0	<input type="text" value="436771F2h"/>	1,130,852,850
A1	<input type="text" value="5A2E51A3h"/>	1,512,984,995
A2	<input type="text" value="xxxxxxxxh"/>	
CSR	<input type="text" value="00010100h"/>	
SSR	<input type="text" value="00000000h"/>	

**1 cycle after instruction**

A0	<input type="text" value="436771F2h"/>	
A1	<input type="text" value="5A2E51A3h"/>	
A2	<input type="text" value="E939204Fh"/>	-382,132,145
CSR	<input type="text" value="00010100h"/>	
SSR	<input type="text" value="00000000h"/>	

**2 cycles after instruction**

A0	<input type="text" value="436771F2h"/>	
A1	<input type="text" value="5A2E51A3h"/>	
A2	<input type="text" value="E939204Fh"/>	
CSR	<input type="text" value="00010100h"/>	Not saturated
SSR	<input type="text" value="00000000h"/>	

## 4.290 SSUB2

Subtract Two Signed 16-Bit Integers on Upper and Lower Register Halves With Saturation

**Syntax**    **SSUB2 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22					18 17					
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>					
3	1			5					5				5	
src1	x	1	1	0	0	1	0	0	1	1	0	s	p	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	s2	.L1, .L2
<i>src2</i>	xs2	
<i>dst</i>	s2	

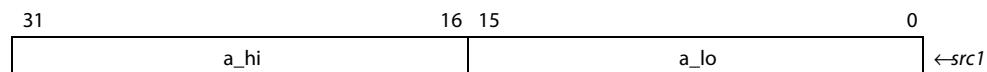
**Description**

Performs 2s-complement subtraction between signed, packed 16-bit quantities in *src1* and *src2*. The results are placed in a signed, packed 16-bit format into *dst*.

For each pair of 16-bit quantities in *src1* and *src2*, the difference between the signed 16-bit value from *src1* and the signed 16-bit value from *src2* is calculated and saturated to produce a signed 16-bit result. The result is placed in the corresponding position in *dst*.

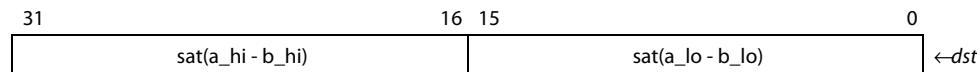
Saturation is performed on each 16-bit result independently. For each sum, the following tests are applied:

- If the difference is in the range  $-2^{15}$  to  $2^{15} - 1$ , inclusive, then no saturation is performed and the sum is left unchanged.
- If the difference is greater than  $2^{15} - 1$ , then the result is set to  $2^{15} - 1$ .
- If the difference is less than  $-2^{15}$ , then the result is set to  $-2^{15}$ .

**SSUB2**

=

=



**Note**—This operation is performed on each halfword separately. This instruction does not affect the SAT bit in CSR or the L1 or L2 bit in SSR.

**Execution**

```
if (cond) {
    sat(msb16(src1) - msb16(src2)) → msb16(dst);
    sat(lsb16(src1) - lsb16(src2)) → lsb16(dst)
}
else nop
```

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**[ADD2](#), [SUB](#), [SUB4](#), [SSUB2](#)**Examples****Example 1**

SSUB2 .L1 A0,A1,A2

**Before instruction**A0 

00070005h
-----------

A1 

FFFFFFFFFFh
-------------

**1 cycle after instruction**A2 

00080006h
-----------

**Example 2**

SSUB2 .L1 A0,A1,A2

**Before instruction**A0 

00070005h
-----------

A1 

8000FFFFFFh
-------------

**1 cycle after instruction**A2 

7FFF0006h
-----------

## 4.291 STB

Store Byte to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

### Syntax

#### Register Offset

**STB** (.unit) *src*, \*+*baseR*[*offsetR*]

unit = .D1 or .D2

#### Unsigned Constant Offset

**STB** (.unit) *src*, \*+*baseR*[*ucst5*]

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>

### Opcode

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>src</i>		<i>baseR</i>		<i>offsetR/ucst5</i>
3	1		5		5		5
13	12	9	8	7	6	5	4
<i>offsetR/ucst5</i>	<i>mode</i>	0	<i>y</i>	0	1	1	0
5	4	1		1	1	1	0
						1	1

### Description

Stores a byte to memory from a general-purpose register (*src*). [Table 3-11](#) on page 3-28 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

*offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

*offsetR/ucst5* is scaled by a left-shift of 0 bits. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is sent to memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see [“Addressing Mode Register \(AMR\)”](#) on page 2-12).

For **STB**, the 8 LSBs of the *src* register are stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* is read from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file. The *r* bit should be cleared to 0.

Increments and decrements default to 1 and offsets default to zero when no bracketed register or constant is specified. Stores that do no modification to the *baseR* can use the syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

**Execution**

```
if (cond) src → mem
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>baseR, offsetR, src</i>
Written	<i>baseR</i>
Unit in use	.D2

**Instruction Type** Store

**Delay Slots** 0

For more information on delay slots for a store, see Chapter 5 “[Pipeline](#)” on page 5-1.

**See Also** [STH, STW](#)

**Examples** [Example 1](#)

STB .D1 A1, \*A10

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A327634h	A1	9A327634h
A10	00000100h	A10	00000100h
mem 100h	11h	mem 100h	11h
		mem 100h	34h

**Example 2**

STB .D1 A8, \*++A4 [5]

	Before instruction	1 cycle after instruction	3 cycles after instruction
A4	0000 4020h	A4	0000 4025h
A8	0123 4567h	A8	0123 4567h
mem 4024:27h	xxxx xxxxh	mem 4024:27h	xxxx xxxxh
		mem 4024:27h	xxxx 67xxh

### Example 3

STB .D1 A8, \*A4++ [5]

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>3 cycles after instruction</b>
A4	0000 4020h	A4	0000 4025h	A4	0000 4025h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4020:23h	xxxx xxxxh	mem 4020:23h	xxxx xxxxh	mem 4020:23h	xxxx 67xxh

### Example 4

STB .D1 A8, \*++A4 [A12]

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>3 cycles after instruction</b>
A4	0000 4020h	A4	0000 4026h	A4	0000 4026h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
mem 4024:27h	xxxx xxxxh	mem 4024:27h	xxxx xxxxh	mem 4024:27h	xx67 xxxxh

## 4.292 STB

Store Byte to Memory With a 15-Bit Unsigned Constant Offset

**Syntax** **STB(.unit) src, \*+B14/B15[ucst15]**

unit = .D2

**Opcode**

31	29	28	27			23	22													
creg	z			<i>src</i>				<i>ucst15</i>												
3	1			5													15			
										8	7	6	5	4	3	2	1	0		
										ucst15	y	0	1	1	1	1	s	p		
										15	1						1	1		

**Description**

Stores a byte to memory from a general-purpose register (*src*). The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 0 bits. After scaling, *ucst15* is added to *baseR*. The result of the calculation is the address that is sent to memory. The addressing arithmetic is always performed in linear mode.

For **STB**, the 8 LSBs of the *src* register are stored. *src* can be in either register file. The *s* bit determines which file *src* is read from: *s* = 0 indicates *src* is in the A register file and *s* = 1 indicates *src* is in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 0. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

**Execution**

```
if (cond) src → mem
else nop
```



**Note**—This instruction executes only on the B side (.D2).

**Pipeline**

Pipeline Stage	E1
Read	B14/B15, <i>src</i>
Written	
Unit in use	.D2

**Instruction Type** Store

**Delay Slots** 0

**See Also** [STH, STW](#)

**Example** STB .D2 B1,\*+B14 [40]

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>3 cycles after instruction</b>
B1	1234 5678h	B1	12345678h	B1	12345678h
B14	00001000h	B14	00001000h	B14	00001000h
mem 1028h	42h	mem 1028h	42h	mem 1028h	78h

## 4.293 STDW

Store Doubleword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

### Syntax

#### Register Offset

**STDW** (.unit) *src*, \*+*baseR*[*offsetR*]

unit = .D1 or .D2

#### Unsigned Constant Offset

**STDW** (.unit) *src*, \*+*baseR*[*ucst5*]

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	<a href="#">Figure C-9</a>
	DindDW	<a href="#">Figure C-11</a>
	DincDW	<a href="#">Figure C-13</a>
	DdecDW	<a href="#">Figure C-15</a>
	Dpp	<a href="#">Figure C-21</a>

### Opcode

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>src</i>		<i>baseR</i>		<i>offsetR/ucst5</i>
3	1		5		5		5
13	12	9	8	7	6	5	4
<i>offsetR/ucst5</i>		<i>mode</i>	1	<i>y</i>	1	0	0
5		4		1		0	1
						1	0
						1	1

Opcode map field used...	For operand type...	Unit
<i>src</i>	ulong	.D1, .D2
<i>baseR</i>	uint	
<i>offsetR</i>	uint	
<i>src</i>	ulong	.D1, .D2
<i>baseR</i>	uint	
<i>offsetR</i>	ucst5	

### Description

Stores a 64-bit quantity to memory from a 64-bit register, *src*. [Table 3-11](#) on page 3-28 describes the addressing generator options. Alignment to a 64-bit boundary is required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*). If an offset is not given, the assembler assigns an offset of zero.

Both *offsetR* and *baseR* must be in the same register file, and on the same side, as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file. The *r* bit has a value of 1 for the **STDW** instruction.

The *offsetR/ucst5* is scaled by a left shift of 3 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

The *src* pair can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* will be loaded from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file.

#### **Assembler Notes**

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Stores that do no modification to the *baseR* can use the assembler syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assembler right shifts the constant by 3 bits for doubleword stores before using it for the *ucst5* field. After scaling by the **STDW** instruction, this results in the same constant offset as the assembler source if the least-significant three bits are zeros.

For example, **STDW** (.unit) *src*, \*+*baseR* (16) represents an offset of 16 bytes (2 doublewords), and the assembler writes out the instruction with *ucst5* = 2. **STDW** (.unit) *src*, \*+*baseR* [16] represents an offset of 16 doublewords, or 128 bytes, and the assembler writes out the instruction with *ucst5* = 16.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used. The register pair syntax always places the odd-numbered register first, a colon, followed by the even-numbered register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

#### **Execution**

```
if (cond) src → mem
else nop
```

#### **Pipeline**

Pipeline Stage	E1
Read	<i>baseR, offsetR, src</i>
Written	<i>baseR</i>
Unit in use	.D

**Instruction Type** Store

**Delay Slots** 0

**See Also** [LDDW](#), [STW](#)

**Examples** [Example 1](#)

```
STDW .D1    A3:A2,*A0++
```

**Before instruction****1 cycle after instruction**A0      

0000 1000h
------------

A0      

0000 1008h
------------

A3:A2      

A176 3B28h	6041 AD65h
------------	------------

A3:A2      

A176 3B28h	6041 AD65h
------------	------------

Byte Memory Address	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	A1	76	3B	28	60	41	AD	65

**Example 2**

STDW .D1 A3:A2, \*A0++

**Before instruction****1 cycle after instruction**A0      

0000 1004h
------------

A0      

0000 100Ch
------------

A3:A2      

A176 3B28h	6041 AD65h
------------	------------

A3:A2      

A176 3B28h	6041 AD65h
------------	------------

Byte Memory Address	100D	100C	100B	100A	1009	1008	1007	1006	1005	1004	1003
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	A1	76	3B	28	60	41	AD	65	00

**Example 3**

STDW .D1 A9:A8, \*++A4 [5]

**Before instruction****1 cycle after instruction**A4      

0000 4020h
------------

A4      

0000 4048h
------------

A9:A8      

ABCD EF98h	0123 4567h
------------	------------

A9:A8      

ABCD EF98h	0123 4567h
------------	------------

Byte Memory Address	4051	4050	404F	404E	404D	404C	404B	404A	4049	4048	4047
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	AB	CD	EF	98	01	23	45	67	00

### Example 4

*STDW .D1 A9:A8, \*++A4 (16)*

#### Before instruction

A4	0000 4020h
----	------------

#### 1 cycle after instruction

A4	0000 4030h
----	------------

A9:A8	ABCD EF98h	0123 4567h	A9:A8	ABCD EF98h	0123 4567h
-------	------------	------------	-------	------------	------------

Byte Memory Address	4039	4038	4037	4036	4035	4034	4033	4032	4031	4030	402F
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	AB	CD	EF	98	01	23	45	67	00

### Example 5

*STDW .D1 A9:A8, \*++A4 [A12]*

#### Before instruction

A4	0000 4020h
----	------------

#### 1 cycle after instruction

A4	0000 4030h
----	------------

A9:A8	ABCD EF98h	0123 4567h	A9:A8	ABCD EF98h	0123 4567h
-------	------------	------------	-------	------------	------------

A12	0000 0006h	A12	0000 0006h
-----	------------	-----	------------

Byte Memory Address	4059	4058	4057	4056	4055	4054	4053	4052	4051	4050	404F
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	00	AB	CD	EF	98	01	23	45	67	00

## 4.294 STH

Store Halfword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

### Syntax

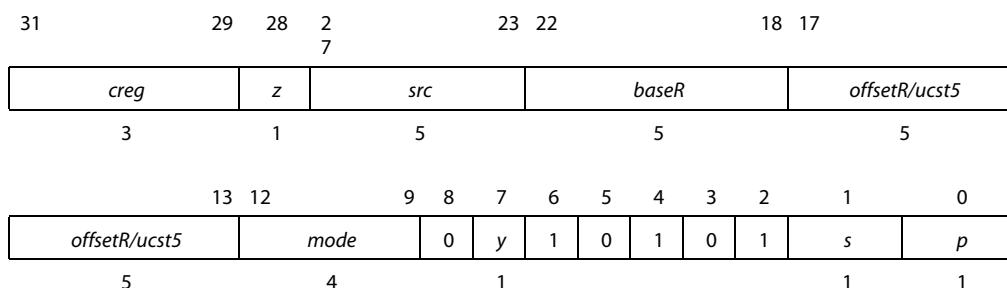
**Register Offset**  
**STH (.unit) src, \*+baseR[offsetR]**  
 unit = .D1 or .D2

**Unsigned Constant Offset**  
**STH (.unit) src, \*+baseR[ucst5]**

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>

### Opcode



### Description

Stores a halfword to memory from a general-purpose register (*src*). [Table 3-10](#) on page 3-28 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

*offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

*offsetR/ucst5* is scaled by a left-shift of 1 bit. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is sent to memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see [“Addressing Mode Register \(AMR\)”](#) on page 2-12).

For STH, the 16 LSBs of the *src* register are stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* is read from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file. The *r* bit should be cleared to 0.

Increments and decrements default to 1 and offsets default to zero when no bracketed register or constant is specified. Stores that do no modification to the *baseR* can use the syntax \**R*. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 1.

Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

**Execution**

```
if (cond) src → mem
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>baseR, offsetR, src</i>
Written	<i>baseR</i>
Unit in use	.D2

**Instruction Type** Store

**Delay Slots** 0

For more information on delay slots for a store, see Chapter 5 “[Pipeline](#)” on page 5-1.

**See Also** [STB, STW](#)

**Examples** [Example 1](#)

STH .D1 A1,\*+A10(4)

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A327634h	A1	9A327634h
B10	00001000h	A10	00001000h
mem 104h	1134h	mem 104h	1134h
			mem 104h 7634h

**Example 2**

STH .D1 A1,\*A10--[A11]

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A322634h	A1	9A322634h
A10	00000100h	A10	000000F8h
A11	00000004h	A11	00000004h

mem F8h	0000h	mem F8h	0000h	mem F8h	0000h
mem 100h	0000h	mem 100h	0000h	mem 100h	2634h

## 4.295 STH

Store Halfword to Memory With a 15-Bit Unsigned Constant Offset

**Syntax** **STH(.unit) src, \*+B14/B15[ucst15]**

unit = .D2

**Opcode**

31	29	28	27			23	22														
creg	z			src				ucst15													
3	1			5				15													
								8	7	6	5	4	3	2	1	0					
								ucst15	y	1	0	1	1	1	s	p					
								15	1						1	1					

**Description**

Stores a halfword to memory from a general-purpose register (*src*). The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 1 bit. After scaling, *ucst15* is added to *baseR*. The result of the calculation is the address that is sent to memory. The addressing arithmetic is always performed in linear mode.

For **STH**, the 16 LSBs of the *src* register are stored. *src* can be in either register file. The *s* bit determines which file *src* is read from: *s* = 0 indicates *src* is in the A register file and *s* = 1 indicates *src* is in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 1. Parentheses, ( ), can be used to set a nonscaled, constant offset. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Halfword addresses must be aligned on halfword (LSB is 0) boundaries.

**Execution**

```
if (cond) src → mem
else nop
```



**Note**—This instruction executes only on the B side (.D2).

**Pipeline**

Pipeline Stage	E1
Read	B14/B15, <i>src</i>
Written	
Unit in use	.D2

**Instruction Type**

Store

**Delay Slots**

0

**See Also**

[STB](#), [STW](#)

## 4.296 STNDW

Store Nonaligned Doubleword to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

### Syntax

#### Register Offset

**STNDW** (.unit) *src*, \*+*baseR*[*offsetR*]

unit = .D1 or .D2

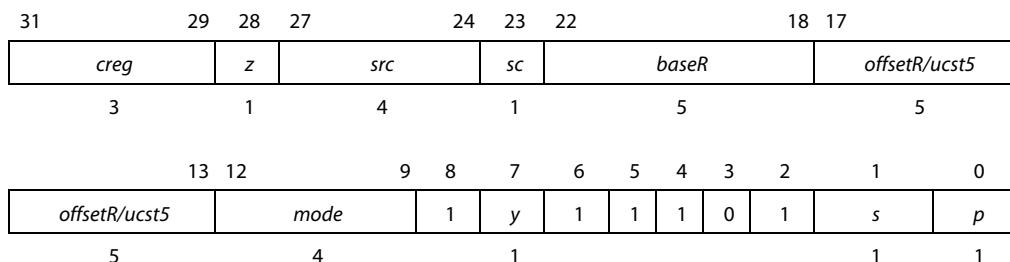
#### Unsigned Constant Offset

**STNDW** (.unit) *src*, \*+*baseR*[*ucst5*]

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4DW	<a href="#">Figure C-9</a>
	DindDW	<a href="#">Figure C-11</a>
	DincDW	<a href="#">Figure C-13</a>
	DdecDW	<a href="#">Figure C-15</a>

### Opcode



Opcode map field used...	For operand type...	Unit
<i>src</i> <i>baseR</i> <i>offsetR</i>	ullong uint uint	.D1, .D2
<i>src</i> <i>baseR</i> <i>offsetR</i>	ullong uint <i>ucst5</i>	.D1, .D2

### Description

Stores a 64-bit quantity to memory from a 64-bit register pair, *src*. [Table 3-11](#) on page 3-28 describes the addressing generator options. The **STNDW** instruction may write a 64-bit value to any byte boundary. Thus alignment to a 64-bit boundary is not required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

Both *offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

The **STNDW** instruction supports both scaled offsets and non-scaled offsets. The *sc* field is used to indicate whether the *offsetR/ucst5* is scaled or not. If *sc* is 1 (scaled), the *offsetR/ucst5* is shifted left 3 bits before adding or subtracting from the *baseR*. If *sc* is 0 (nonscaled), the *offsetR/ucst5* is not shifted before adding to or subtracting from the

*baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or post-decrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

The *src* pair can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* will be loaded from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file. The *r* bit has a value of 1 for the STNDW instruction.



**Note**—No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel, as long as it is not performing a memory access.

#### Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1, and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 3 for doubleword stores.

Parentheses, ( ), can be used to indicate to the assembler that the offset is a nonscaled offset.

For example, **STNDW (.unit) src, \*+baseR (12)** represents an offset of 12 bytes and the assembler writes out the instruction with *offsetC* = 12 and *sc* = 0.

**STNDW (.unit) src, \*+baseR [16]** represents an offset of 16 doublewords, or 128 bytes, and the assembler writes out the instruction with *offsetC* = 16 and *sc* = 1.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

#### Execution

```
if (cond) src → mem
else nop
```

#### Pipeline

Pipeline Stage	E1
Read	<i>baseR, offsetR, src</i>
Written	<i>baseR</i>
Unit in use	.D

**Instruction Type** Store

**Delay Slots** 0

**See Also** [LDNW](#), [LDNDW](#), [STNW](#)

#### Examples

##### Example 1

```
STNDW .D1 A3:A2, *A0++
```

**Before instruction****1 cycle after instruction**

A0	0000 1001h		A0	0000 1009h	
A3	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	60	41	AD	65	00

**Example 2**

STNDW .D1 A3:A2, \*A0++

**Before instruction****1 cycle after instruction**

A0	0000 1003h		A0	0000 100Bh	
A3:A2	A176 3B28h	6041 AD65h	A3:A2	A176 3B28h	6041 AD65h

Byte Memory Address	100B	100A	1009	1008	1007	1006	1005	1004	1003	1002	1001	1000
Data Value Before Store	00	00	00	00	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	60	41	AD	65	00	00	00

## 4.297 STNW

Store Nonaligned Word to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

### Syntax

#### Register Offset

**STNW** (.unit) *src*, \*+*baseR*[*offsetR*]

unit = .D1 or .D2

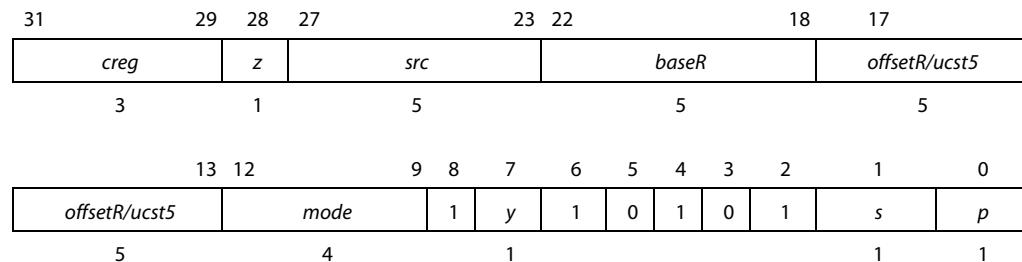
#### Unsigned Constant Offset

**STNW** (.unit) *src*, \*+*baseR*[*ucst5*]

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>

### Opcode



Opcode map field used...	For operand type...	Unit
<i>src</i>	uint	.D1, .D2
<i>baseR</i>	uint	
<i>offsetR</i>	uint	
<i>src</i>	uint	.D1, .D2
<i>baseR</i>	uint	
<i>offsetR</i>	ucst5	

### Description

Stores a 32-bit quantity to memory from a 32-bit register, *src*. [Table 3-11](#) on page 3-28 describes the addressing generator options. The **STNW** instruction may write a 32-bit value to any byte boundary. Thus alignment to a 32-bit boundary is not required. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

Both *offsetR* and *baseR* must be in the same register file, and on the same side, as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

The *offsetR/ucst5* is scaled by a left shift of 2 bits. After scaling, *offsetR/ucst5* is added to, or subtracted from, *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is the address to be accessed from memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12).

The *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* will be loaded from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file. The *r* bit has a value of 1 for the STNW instruction.



**Note**—No other memory access may be issued in parallel with a nonaligned memory access. The other .D unit can be used in parallel as long as it is not performing memory access.

#### Assembler Notes

When no bracketed register or constant is specified, the assembler defaults increments and decrements to 1 and offsets to 0. Loads that do no modification to the *baseR* can use the assembler syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 2 for word stores.

Parentheses, ( ), can be used to tell the assembler that the offset is a non-scaled, constant offset. The assembler right shifts the constant by 2 bits for word stores before using it for the *ucst5* field. After scaling by the STNW instruction, this results in the same constant offset as the assembler source if the least-significant two bits are zeros.

For example, STNW (.unit) *src,\*+baseR* (12) represents an offset of 12 bytes (3 words), and the assembler writes out the instruction with *ucst5* = 3.

STNW (.unit) *src,\*+baseR* [12] represents an offset of 12 words, or 48 bytes, and the assembler writes out the instruction with *ucst5* = 12.

Either brackets or parentheses must be typed around the specified offset if the optional offset parameter is used.

#### Execution

```
if (cond) src → mem
else nop
```

#### Pipeline

Pipeline Stage	E1
Read	<i>baseR, offsetR, src</i>
Written	<i>baseR</i>
Unit in use	.D

#### Instruction Type

Store

#### Delay Slots

0

#### See Also

[LDNW](#), [LDNDW](#), [STNDW](#)

#### Examples

**Example 1**

```
STNW .D1    A3, *A0++
```

**Before instruction****1 cycle after instruction**A0 0000 1001hA0 0000 1005hA3 A176 3B28hA3 A176 3B28h

<b>Byte Memory Address</b>	<b>1007</b>	<b>1006</b>	<b>1005</b>	<b>1004</b>	<b>1003</b>	<b>1002</b>	<b>1001</b>	<b>1000</b>
Data Value Before Store	00	00	00	00	00	00	00	00
Data Value After Store	00	00	00	A1	76	3B	28	00

**Example 2**

STNW .D1 A3, \*A0++

**Before instruction****1 cycle after instruction**A0 0000 1003hA0 0000 1007hA3 A176 3B28hA3 A176 3B28h

<b>Byte Memory Address</b>	<b>1007</b>	<b>1006</b>	<b>1005</b>	<b>1004</b>	<b>1003</b>	<b>1002</b>	<b>1001</b>	<b>1000</b>
Data Value Before Store	00	00	00	00	00	00	00	00
Data Value After Store	00	A1	76	3B	28	00	00	00

## 4.298 STW

Store Word to Memory With a 5-Bit Unsigned Constant Offset or Register Offset

### Syntax

#### Register Offset

**STW** (.unit) *src*, \*+*baseR*[*offsetR*]

unit = .D1 or .D2

#### Unsigned Constant Offset

**STW** (.unit) *src*, \*+*baseR*[*ucst5*]

### Compact Instruction Format

Unit	Opcode Format	Figure
.D	Doff4	<a href="#">Figure C-8</a>
	Dind	<a href="#">Figure C-10</a>
	Dinc	<a href="#">Figure C-12</a>
	Ddec	<a href="#">Figure C-14</a>

### Opcode

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>src</i>		<i>baseR</i>		<i>offsetR/ucst5</i>
3	1		5		5		5
13	12	9	8	7	6	5	4
<i>offsetR/ucst5</i>	<i>mode</i>	0	<i>y</i>	1	1	1	0
5	4	1		1	1	1	1
						1	0
						1	1

### Description

Stores a word to memory from a general-purpose register (*src*). [Table 3-11](#) on page 3-28 describes the addressing generator options. The memory address is formed from a base address register (*baseR*) and an optional offset that is either a register (*offsetR*) or a 5-bit unsigned constant (*ucst5*).

*offsetR* and *baseR* must be in the same register file and on the same side as the .D unit used. The *y* bit in the opcode determines the .D unit and register file used: *y* = 0 selects the .D1 unit and *baseR* and *offsetR* from the A register file, and *y* = 1 selects the .D2 unit and *baseR* and *offsetR* from the B register file.

*offsetR/ucst5* is scaled by a left-shift of 2 bits. After scaling, *offsetR/ucst5* is added to or subtracted from *baseR*. For the preincrement, predecrement, positive offset, and negative offset address generator options, the result of the calculation is the address to be accessed in memory. For postincrement or postdecrement addressing, the value of *baseR* before the addition or subtraction is sent to memory.

The addressing arithmetic that performs the additions and subtractions defaults to linear mode. However, for A4-A7 and for B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see [“Addressing Mode Register \(AMR\)”](#) on page 2-12).

For STW, the entire 32-bits of the *src* register are stored. *src* can be in either register file, regardless of the .D unit or *baseR* or *offsetR* used. The *s* bit determines which file *src* is read from: *s* = 0 indicates *src* will be in the A register file and *s* = 1 indicates *src* will be in the B register file. The *r* bit should be cleared to 0.

Increments and decrements default to 1 and offsets default to zero when no bracketed register or constant is specified. Stores that do no modification to the *baseR* can use the syntax \*R. Square brackets, [ ], indicate that the *ucst5* offset is left-shifted by 2. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **STW (.unit) src, \*+baseR(12)** represents an offset of 12 bytes; whereas, **STW (.unit) src, \*+baseR[12]** represents an offset of 12 words, or 48 bytes. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Word addresses must be aligned on word (two LSBs are 0) boundaries.

**Execution**

```
if (cond) src → mem
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>baseR, offsetR, src</i>
Written	<i>baseR</i>
Unit in use	.D2

**Instruction Type** Store

**Delay Slots** 0

For more information on delay slots for a store, see Chapter 5 “[Pipeline](#)” on page 5-1.

**See Also** [STB, STH](#)

**Examples** [Example 1](#)

STW .D1 A1, \*++A10 [1]

	Before instruction	1 cycle after instruction	3 cycles after instruction
A1	9A327634h	A1	9A327634h
A10	00000100h	A10	00000104h
mem 100h	11111134h	mem 100h	11111134h
mem 104h	00001111h	mem 104h	00001111h
			9A327634h

**Example 2**

STW .D1 A8, \*++A4 [5]

	Before instruction	1 cycle after instruction	3 cycles after instruction
A4	0000 4020h	A4	0000 4034h
			0000 4034h

A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh
mem 4034h	xxxx xxxxh	mem 4034h	xxxx xxxxh	mem 4034h	0123 4567h

### Example 3

STW .D1 A8 , \*++A4 (8)

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>3 cycles after instruction</b>
A4	0000 4020h	A4	0000 4028h	A4	0000 4028h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh
mem 4028h	xxxx xxxxh	mem 4028h	xxxx xxxxh	mem 4028h	0123 4567h

### Example 4

STW .D1 A8 , \*++A4 [A12]

	<b>Before instruction</b>		<b>1 cycle after instruction</b>		<b>3 cycles after instruction</b>
A4	0000 4020h	A4	0000 4038h	A4	0000 4038h
A8	0123 4567h	A8	0123 4567h	A8	0123 4567h
A12	0000 0006h	A12	0000 0006h	A12	0000 0006h
mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh	mem 4020h	xxxx xxxxh
mem 4038h	xxxx xxxxh	mem 4038h	xxxx xxxxh	mem 4038h	0123 4567h

## 4.299 STW

Store Word to Memory With a 15-Bit Unsigned Constant Offset

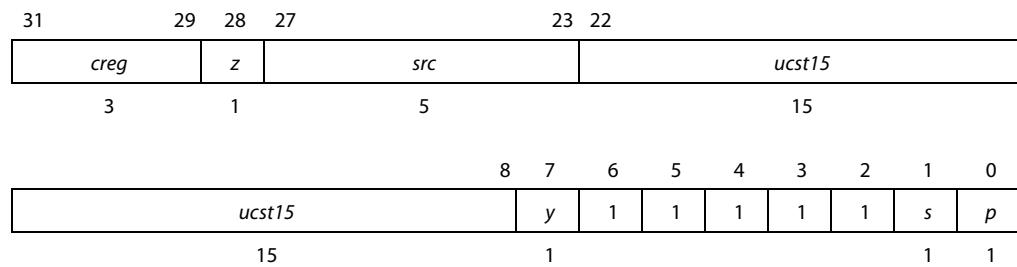
**Syntax** **STW(.unit) src, \*+B14/B15[ucst15]**

unit = .D2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Dstk Dpp	<a href="#">Figure C-16</a> <a href="#">Figure C-21</a>

**Opcode**



**Description**

Stores a word to memory from a general-purpose register (*src*). The memory address is formed from a base address register B14 (*y* = 0) or B15 (*y* = 1) and an offset, which is a 15-bit unsigned constant (*ucst15*). The assembler selects this format only when the constant is larger than five bits in magnitude. This instruction executes only on the .D2 unit.

The offset, *ucst15*, is scaled by a left-shift of 2 bits. After scaling, *ucst15* is added to *baseR*. The result of the calculation is the address that is sent to memory. The addressing arithmetic is always performed in linear mode.

For **STW**, the entire 32-bits of the *src* register are stored. *src* can be in either register file. The *s* bit determines which file *src* is read from: *s* = 0 indicates *src* is in the A register file and *s* = 1 indicates *src* is in the B register file.

Square brackets, [ ], indicate that the *ucst15* offset is left-shifted by 2. Parentheses, ( ), can be used to set a nonscaled, constant offset. For example, **STW (.unit) src, \*+B14/B15(60)** represents an offset of 12 bytes; whereas, **STW (.unit) src, \*+B14/B15[60]** represents an offset of 60 words, or 240 bytes. You must type either brackets or parentheses around the specified offset, if you use the optional offset parameter.

Word addresses must be aligned on word (two LSBs are 0) boundaries.

**Execution**

```
if (cond) src → mem
else nop
```



**Note**—This instruction executes only on the B side (.D2).

**Pipeline**

Pipeline Stage	E1
Read	B14/B15, src
Written	
Unit in use	.D2

**Instruction Type** Store

**Delay Slots** 0

**See Also** [STB](#), [STH](#)

## 4.300 SUB

Subtract Two Signed Integers Without Saturation

**Syntax**    **SUB** (.unit) *src1, src2, dst*

or

**SUB** (.L1 or .L2) *src1, src2, dst\_h:dst\_l*

or

**SUB** (.D1 or .D2) *src2, src1, dst* (if the cross path form is not used)

or

**SUB** (.D1 or .D2) *src1, src2, dst* (if the cross path form is used)

unit = .D1, .D2, .L1, .L2, .S1, .S2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.L	L3	<a href="#">Figure D-4</a>
	Lx1	<a href="#">Figure D-11</a>
.S	S3	<a href="#">Figure F-17</a>
	Sx2op	<a href="#">Figure F-24</a>
	Sx1	<a href="#">Figure F-26</a>
.D	Dx2op	<a href="#">Figure C-17</a>
	Dx1	<a href="#">Figure C-20</a>



**Note**—Subtraction with a signed constant on the .L and .S units allows either the first or the second operand to be the signed 5-bit constant. **SUB** (.unit) *src1, scst5, dst* is encoded as **ADD** (.unit) *-scst5, src2, dst* where the *src1* register is now *src2* and *scst5* is now *-scst5*.

The .D unit, when the cross path form is not used, provides only the second operand as a constant since it is an unsigned 5-bit constant. *ucst5* allows a greater offset for addressing with the .D unit.

**Opcode**

.L unit

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
13	12	11		5	4	3	0
<i>src1</i>	<i>x</i>		<i>op</i>	1	1	0	<i>s</i>
5	1		7				1

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
<i>src1</i>	sint	.L1, .L2	0000111
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	xsint	.L1, .L2	0010111
<i>src2</i>	sint		
<i>dst</i>	sint		
<i>src1</i>	sint	.L1, .L2	0100111
<i>src2</i>	xsint		
<i>dst</i>	slong		
<i>src1</i>	xsint	.L1, .L2	0110111
<i>src2</i>	sint		
<i>dst</i>	slong		
<i>src1</i>	scst5	.L1, .L2	0000110
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.L1, .L2	0100100
<i>src2</i>	slong		
<i>dst</i>	slong		

**Opcode** .S unit

31	29	28	27		23	22		18	17
<i>creg</i>	<i>z</i>	<i>dst</i>			<i>src2</i>			<i>src1</i>	
3	1			5			5		5
13	12	11		6	5	4	3	2	1
<i>src1</i>	<i>x</i>	<i>op</i>			1	0	0	0	<i>s</i>
5	1			6					1
									1

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
<i>src1</i>	sint	.S1, .S2	010111
<i>src2</i>	xsint		
<i>dst</i>	sint		
<i>src1</i>	scst5	.S1, .S2	010110
<i>src2</i>	xsint		
<i>dst</i>	sint		

*src2* - *src1*:

31	30	29	28	27		23	22		18	17
0	0	0	<i>z</i>	<i>dst</i>			<i>src2</i>			<i>src1</i>
			1		5			5		5
13	12	11	10	9	8	7	6	5	4	3
<i>src1</i>	<i>x</i>	1	1	0	1	0	1	1	1	0
5	1									1
										1

Opcode map field used...	For operand type...	Unit
<i>src2</i>	sint	.S1, .S2
<i>src1</i>	sint	
<i>dst</i>	sint	

**Description for .L1, .L2 and .S1, .S2 OpCodes**

**Execution for .L1, .L2 and .S1, .S2 OpCodes**

**Opcode**

.D unit (if the cross path form is not used)

31	29	28	27	23	22	18	17
<i>creg</i>	z		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
	13	12		7	6	5	4
	<i>src1</i>		<i>op</i>	1	0	0	0
	5		6				s
							p
				1	1		

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	010001
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	010011
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description** *src1* is subtracted from *src2*. The result is placed in *dst*.

**Execution** if (cond) *src2* - *src1* → *dst*  
else nop

**Opcode**

.D unit (if the cross path form is used)

31	29	28	27	23	22	18	17
<i>creg</i>	z		<i>dst</i>		<i>src2</i>		<i>src1</i>
3	1		5		5		5
	13	12	11	10	9	8	7
	<i>src1</i>	x	1	0	1	1	0
	5		1				
				0	1	1	0
					0	0	s
						p	
						1	1

Opcode map field used...	For operand type...	Unit
<i>src1</i>	sint	.D1, .D2
<i>src2</i>	xsint	
<i>dst</i>	sint	

**Description** *src2* is subtracted from *src1*. The result is placed in *dst*.

**Execution**

```
if (cond) src1 - src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S, or .D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD](#), [SUBC](#), [SUBU](#), [SSUB](#), [SUB2](#)

**Example** SUB .L1 A1,A2,A3

Before instruction			1 cycle after instruction		
A1	0000325Ah	12,890	A1	0000325Ah	
A2	FFFFF12h	-238	A2	FFFFF12h	
A3	xxxxxxxxh		A3	00003348h	13,128

## 4.301 SUBAB

Subtract Using Byte Addressing Mode

**Syntax** **SUBAB** (.unit) *src2*, *src1*, *dst*

unit = .D1 or .D2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3	1		5			5	
13	12			7	6	5	4
<i>src1</i>			<i>op</i>	1	0	0	0
5			6				
						0	
						<i>s</i>	<i>p</i>
						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	110001
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	110011
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description**

*src1* is subtracted from *src2* using the byte addressing mode specified for *src2*. The subtraction defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “Addressing Mode Register (AMR)” on page 2-12). The result is placed in *dst*.

**Execution**

```
if (cond) src2 -a src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1</i> , <i>src2</i>
Written	<i>dst</i>
Unit in use	.D

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**

[SUB](#), [SUBAH](#), [SUBAW](#)

**Example**

SUBAB .D1 A5,A0,A5



AMR

00030004h

AMR

00030004h

1. BK0 = 3 → size = 16  
A5 in circular addressing mode using BK0

## 4.302 SUBABS4

Subtract With Absolute Value, Four 8-Bit Pairs for Four 8-Bit Results

**Syntax** **SUBABS4** (.unit) *src1*, *src2*, *dst*

unit = .L1 or .L2

**Opcode**

31	29	28	27	23 22					18 17					
<i>creg</i>	<i>z</i>			<i>dst</i>					<i>src2</i>					
3	1			5					5				5	
<i>src1</i>	x	1	0	1	1	0	1	0	1	1	0	s	p	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	u4	.L1, .L2
<i>src2</i>	xu4	
<i>dst</i>	u4	

**Description**

Calculates the absolute value of the differences between the packed 8-bit data contained in the source registers. The values in *src1* and *src2* are treated as unsigned, packed 8-bit quantities. The result is written into *dst* in an unsigned, packed 8-bit format.

For each pair of unsigned 8-bit values in *src1* and *src2*, the absolute value of the difference is calculated. This result is then placed in the corresponding position in *dst*.

- The absolute value of the difference between *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*.
- The absolute value of the difference between *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*.
- The absolute value of the difference between *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*.
- The absolute value of the difference between *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.

The **SUBABS4** instruction aids in motion-estimation algorithms, and other algorithms, that compute the "best match" between two sets of 8-bit quantities.

31	24 23	16 15	8 7	0
ua_3	ua_2	ua_1	ua_0	$\leftarrow src1$

**SUBABS4**

ub_3	ub_2	ub_1	ub_0	$\leftarrow src2$
------	------	------	------	-------------------

=                            =                            =                            =

31	24 23	16 15	8 7	0
abs(ua_3 - ub_3)	abs(ua_2 - ub_2)	abs(ua_1 - ub_1)	abs(ua_0 - ub_0)	$\leftarrow dst$

**Execution**

```
if (cond) {
    abs(ubyte0(src1) - ubyte0(src2)) → ubyte0(dst);
    abs(ubyte1(src1) - ubyte1(src2)) → ubyte1(dst);
    abs(ubyte2(src1) - ubyte2(src2)) → ubyte2(dst);
    abs(ubyte3(src1) - ubyte3(src2)) → ubyte3(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** ABS, SUB, SUB4

**Example** SUBABS4 .L1 A2, A8, A9

**Before instruction**

A2 37 89 F2 3Ah 55 137 242 58  
unsigned

**1 cycle after instruction**

A2 37 89 F2 3Ah

A8 04 B8 49 75h 4 184 73 117  
unsigned

A8 04 B8 49 75h

A9 xxxx xxxxh

A9 33 2F A9 3Bh 51 47 169 59  
unsigned

## 4.303 SUBAH

Subtract Using Halfword Addressing Mode

**Syntax** **SUBAH** (.unit) *src2*, *src1*, *dst*

unit = .D1 or .D2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	<i>src1</i>
3	1		5			5	5
13	12			7	6	5	4
<i>src1</i>			<i>op</i>	1	0	0	0
5			6				
						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	110101
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	110111
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description** *src1* is subtracted from *src2* using the halfword addressing mode specified for *src2*. The subtraction defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12). *src1* is left shifted by 1. The result is placed in *dst*.

**Execution**

```
if (cond) src2 -a src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1</i> , <i>src2</i>
Written	<i>dst</i>
Unit in use	.D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SUB](#), [SUBAB](#), [SUBAW](#)

## 4.304 SUBAW

Subtract Using Word Addressing Mode

**Syntax** **SUBAW (.unit) src2, src1, dst**

unit = .D1 or .D2

**Compact Instruction Format**

Unit	Opcode Format	Figure
.D	Dx5p	<a href="#">Figure C-19</a>

**Opcode**

31	29	28	27		23	22		18	17
<i>creg</i>		<i>z</i>		<i>dst</i>		<i>src2</i>		<i>src1</i>	
3		1		5		5		5	

13	12		7	6	5	4	3	2	1	0
<i>src1</i>		<i>op</i>		1	0	0	0	0	<i>s</i>	<i>p</i>
5		6							1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src2</i>	sint	.D1, .D2	111001
<i>src1</i>	sint		
<i>dst</i>	sint		
<i>src2</i>	sint	.D1, .D2	111011
<i>src1</i>	ucst5		
<i>dst</i>	sint		

**Description**

*src1* is subtracted from *src2* using the word addressing mode specified for *src2*. The subtraction defaults to linear mode. However, if *src2* is one of A4-A7 or B4-B7, the mode can be changed to circular mode by writing the appropriate value to the AMR (see “[Addressing Mode Register \(AMR\)](#)” on page 2-12). *src1* is left shifted by 2. The result is placed in *dst*.

**Execution**

```
if (cond) src2 - a src1 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SUB](#), [SUBAB](#), [SUBAH](#)

**Example** SUBAW .D1 A5,2,A3

**Before instruction<sup>1</sup>****1 cycle after instruction**

A3	xxxxxxxh	A3	00000108h
A5	00000100h	A5	00000100h
AMR	00030004h	AMR	00030004h

1. BK0 = 3 → size = 16  
A5 in circular addressing mode using BK0

## 4.305 SUBC

Subtract Conditionally and Shift—Used for Division

**Syntax** **SUBC (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	dst					src2					src1		
3				z					23	22				18	17	
								5						5		5
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	1	0	0	1	0	1	1	1	1	0	s	p		
		5		1										1	1	

Opcode map field used...	For operand type...	Unit
src1	uint	.L1, .L2
src2	xuint	
dst	uint	

**Description** Subtract *src2* from *src1*. If result is greater than or equal to 0, left shift result by 1, add 1 to it, and place it in *dst*. If result is less than 0, left shift *src1* by 1, and place it in *dst*. This step is commonly used in division.

**Execution**

```
if (cond){
    if ((src1 - src2) ≥ 0), ((src1 - src2) << 1) + 1 → dst
    else (src1 << 1) → dst
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD](#), [SSUB](#), [SUB](#), [SUBDP](#), [SUBSP](#), [SUBU](#), [SUB2](#)

**Examples** **Example 1**

SUBC .L1 A0,A1,A0

Before instruction				1 cycle after instruction			
A0	0000125Ah	4698		A0	0000024B4h	9396	
A1	00001F12h	7954		A1	00001F12h		

### Example 2

SUBC .L1 A0,A1,A0

	Before instruction		1 cycle after instruction	
A0	00021A31h	137,777	A0	000047E5h 18,405
A1	0001F63Fh	128,575	A1	0001F63Fh

## 4.306 SUBDP

Subtract Two Double-Precision Floating-Point Values

**Syntax** **SUBDP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, .S2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3	1		5			5	
<i>src1</i>	<i>x</i>		<i>op</i>		5	4	3
5	1		7		1	1	0

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	dp	.L1, .L2	0011001
<i>src2</i>	xdp		
<i>dst</i>	dp		
<i>src1</i>	xdp	.L1, .L2	0011101
<i>src2</i>	dp		
<i>dst</i>	dp		
<i>src1</i>	dp	.S1, .S2	1110011
<i>src2</i>	xdp		
<i>dst</i>	dp		
<i>src1</i>	dp	.S1, .S2	1110111
<i>src2</i>	xdp		
<i>dst</i>	dp		



**Note**—The assembly syntax allows a cross-path operand to be used for either *src1* or *src2*. The assembler selects between the two opcodes based on which source operand in the assembly instruction requires the cross path. If *src1* requires the cross path, the assembler chooses the second (reverse) form of the instruction syntax and reverses the order of the operands in the encoded instruction.

**Description**

*src2* is subtracted from *src1*. The result is placed in *dst*.



**Note**—

- 1) This instruction takes the rounding mode from and sets the warning bits in the floating-point adder configuration register (FADCR), not the floating-point auxiliary configuration register (FAUCR) as for other .S unit instructions.
- 2) The source specific warning bits set in FADCR are set according to the registers sources in the actual machine instruction and not according to the order of the sources in the assembly form.
- 3) If rounding is performed, the INEX bit is set.
- 4) If one source is SNaN or QNaN, the result is NaN\_out. If either source is SNaN, the INVAL bit is set also.
- 5) If both sources are +infinity or -infinity, the result is NaN\_out and the INVAL bit is set.

- 6) If one source is signed infinity and the other source is anything except NaN or signed infinity of the same sign, the result is signed infinity and the INFO bit is set.
- 7) If overflow occurs, the INEX and OVER bits are set and the results are set as follows (LFPN is the largest floating-point number):

Result Sign	Overflow Output Rounding Mode			
	Nearest Even	Zero	+Infinity	Infinity
+	+infinity	+LFPN	+infinity	+LFPN
-	-infinity	-LFPN	-LFPN	-infinity

- a. If underflow occurs, the INEX and UNDER bits are set and the results are set as follows (SPFN is the smallest floating-point number):

Result Sign	Underflow Output Rounding Mode			
	Nearest Even	Zero	+Infinity	Infinity
+	+0	+0	+SPFN	+0
-	-0	-0	-0	-SPFN

- b. If the sources are equal numbers of the same sign, the result is +0 unless the rounding mode is -infinity, in which case the result is -0.
- c. the sources are both 0 with opposite signs or both denormalized with opposite signs, the sign of the result is the same as the sign of *src1*.
- d. A signed denormalized source is treated as a signed 0 and the DENn bit is set. If the other source is not NaN or signed infinity, the INEX bit is also set.

**Execution**

```
if (cond) src1 - src2 → dst
else nop
```

**Pipeline**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
<b>Read</b>	<i>src1_l</i> , <i>src2_l</i>	<i>src1_h</i> , <i>src2_h</i>					
<b>Written</b>						<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.L or .S	.L or .S					

The low half of the result is written out one cycle earlier than the high half. If *dst* is used as the source for the ADDDP, CMPEQDP, CMPLTDP, CMPGTDP, MPYDP, MPYSPDP, MPYSP2DP, or SUBDP instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type**

ADDDP/SUBDP

**Delay Slots**

6

**Functional Unit Latency**

2

**See Also**[ADDDP](#), [SUB](#), [SUBSP](#), [SUBU](#)**Example**

SUBDP .L1X B1:B0,A3:A2,A5:A4

<b>Before instruction</b>			<b>7 cycles after instruction</b>			
B1:B0	4021 3333h	3333 3333h	B1:B0	4021 3333h	4021 3333h	8.6
A3:A2	C004 0000h	0000 0000h	A3:A2	C004 0000h	0000 0000h	-2.5
A5:A4	xxxx xxxxh	xxxx xxxxh	A5:A4	4026 3333h	3333 3333h	11.1

## 4.307 SUBSP

Subtract Two Single-Precision Floating-Point Values

**Syntax** **SUBSP** (.unit) *src1*, *src2*, *dst*

unit = .L1, .L2, .S1, .S2

**Opcode**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	<i>src1</i>
3	1		5			5	5
		13	12	11		5	4
		<i>src1</i>	<i>x</i>	<i>op</i>		1	1
		5	1	7		0	
						1	1

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	sp	.L1, .L2	0010001
<i>src2</i>	xsp		
<i>dst</i>	sp		
<i>src1</i>	xsp	.L1, .L2	0010101
<i>src2</i>	sp		
<i>dst</i>	sp		
<i>src1</i>	sp	.S1, .S2	1110001
<i>src2</i>	xsp		
<i>dst</i>	sp		
<i>src1</i>	sp	.S1, .S2	1110101
<i>src2</i>	xsp		
<i>dst</i>	sp		<i>src2 - src1</i>



**Note**—The assembly syntax allows a cross-path operand to be used for either *src1* or *src2*. The assembler selects between the two opcodes based on which source operand in the assembly instruction requires the cross path. If *src1* requires the cross path, the assembler chooses the second (reverse) form of the instruction syntax and reverses the order of the operands in the encoded instruction.

**Description**

*src2* is subtracted from *src1*. The result is placed in *dst*.



**Note**—

- 1) This instruction takes the rounding mode from and sets the warning bits in the floating-point adder configuration register (FADCR), not the floating-point auxiliary configuration register (FAUCR) as for other .S unit instructions.
- 2) The source specific warning bits set in FADCR are set according to the registers sources in the actual machine instruction and not according to the order of the sources in the assembly form.
- 3) If rounding is performed, the INEX bit is set.
- 4) If one source is SNaN or QNaN, the result is NaN\_out. If either source is SNaN, the INVAL bit is set also.
- 5) If both sources are +infinity or -infinity, the result is NaN\_out and the INVAL bit is set.

- 6) If one source is signed infinity and the other source is anything except NaN or signed infinity of the same sign, the result is signed infinity and the INFO bit is set.
- 7) If overflow occurs, the INEX and OVER bits are set and the results are set as follows (LFPN is the largest floating-point number):

<b>Overflow Output Rounding Mode</b>				
<b>Result Sign</b>	<b>Nearest Even</b>	<b>Zero</b>	<b>+Infinity</b>	<b>Infinity</b>
+	+infinity	+LFPN	+infinity	+LFPN
-	-infinity	-LFPN	-LFPN	-infinity

- 8) If underflow occurs, the INEX and UNDER bits are set and the results are set as follows (SPFN is the smallest floating-point number):

<b>Underflow Output Rounding Mode</b>				
<b>Result Sign</b>	<b>Nearest Even</b>	<b>Zero</b>	<b>+Infinity</b>	<b>Infinity</b>
+	+0	+0	+SPFN	+0
-	-0	-0	-0	-SPFN

- 9) If the sources are equal numbers of the same sign, the result is +0 unless the rounding mode is -infinity, in which case the result is -0.
- 10) If the sources are both 0 with opposite signs or both denormalized with opposite signs, the sign of the result is the same as the sign of *src1*.
- 11) A signed denormalized source is treated as a signed 0 and the DENN bit is set. If the other source is not NaN or signed infinity, the INEX bit is also set.

**Execution**

```
if (cond) src1 - src2 → dst
else nop
```

**Pipeline**

<b>Pipeline Stage</b>	<b>E1</b>	<b>E2</b>	<b>E3</b>	<b>E4</b>
<b>Read</b>	<i>src1, src2</i>			
<b>Written</b>				
<b>Unit in use</b>	.L or .S			

**Instruction Type** 4-cycle**Delay Slots** 3**Functional Unit Latency** 1**See Also** ADDDP, SUB, SUBDP, SUBU**Example** SUBSP .L1X A2,B1,A3**Before instruction**

A2 4109 999Ah

**4 cycles after instruction**

A2 4109 999Ah 8.6

B1 C020 0000h

B1 C020 0000h -2.5

A3      

xxxxxxxxh
-----------

A3      

4131 999Ah
------------

 11.1

## 4.308 SUBU

Subtract Two Unsigned Integers Without Saturation

**Syntax** **SUBU (.unit) src1, src2, dst**

or

**SUBU (.unit) src1, src2, dst\_h:dst\_l**

unit = .L1 or .L2

**Opcode**

31	29	28	27			23	22			18	17
<i>creg</i>	z			<i>dst</i>				<i>src2</i>			<i>src1</i>
3		1		5				5		5	
13	12	11				5	4	3	2	1	0
<i>src1</i>	x			<i>op</i>		1	1	0	s	p	
5		1		7				1	1		

Opcode map field used...	For operand type...	Unit	Opfield
<i>src1</i>	uint	.L1,.L2	0101111
<i>src2</i>	xuint		
<i>dst</i>	ulong		
<i>src1</i>	xuint	.L1,.L2	0111111
<i>src2</i>	uint		
<i>dst</i>	ulong		

**Description** *src2* is subtracted from *src1*. The result is placed in *dst*.

**Execution** if (cond) *src1* - *src2* → *dst*  
else nop

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADDU](#), [SSUB](#), [SUB](#), [SUBC](#), [SUBDP](#), [SUBSP](#), [SUB2](#)

**Example** SUBU .L1 A1,A2,A5:A4

**Before instruction**

A1	<span style="border: 1px solid black; padding: 2px;">0000325Ah</span>	12,890 <sup>1</sup>	A1	<span style="border: 1px solid black; padding: 2px;">0000325Ah</span>
----	---	---------------------	----	---

**1 cycle after instruction**

A2	<span style="border: 1px solid black; padding: 2px;">FFFFFFFFFF12h</span>	4,294,967,058 <sup>1</sup>	A2	<span style="border: 1px solid black; padding: 2px;">FFFFFFFFFF12h</span>
----	---	----------------------------	----	---

A5:A4	xxxxxxxxh	xxxxxxxxh	A5:A4	000000FFh	00003348h	-4,294,954,168 <sup>2</sup>
-------	-----------	-----------	-------	-----------	-----------	-----------------------------

1. Unsigned 32-bit integer
2. Signed 40-bit (long) integer

## 4.309 SUB2

Subtract Two 16-Bit Integers on Upper and Lower Register Halves

**Syntax** **SUB2 (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2, .D1, .D2

**Opcode** .L unit

31	29	28	27	dst					src2					src1		
3	1			5				5				5				
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	0	0	0	0	1	0	0	1	1	0	s	p		

Opcode map field used...	For operand type...	Unit
src1	i2	.L1, .L2
src2	xi2	
dst	i2	

**Opcode** .S unit

31	29	28	27	dst					src2					src1		
3	1			5				5				5				
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	0	1	0	0	0	1	1	0	0	0	s	p		

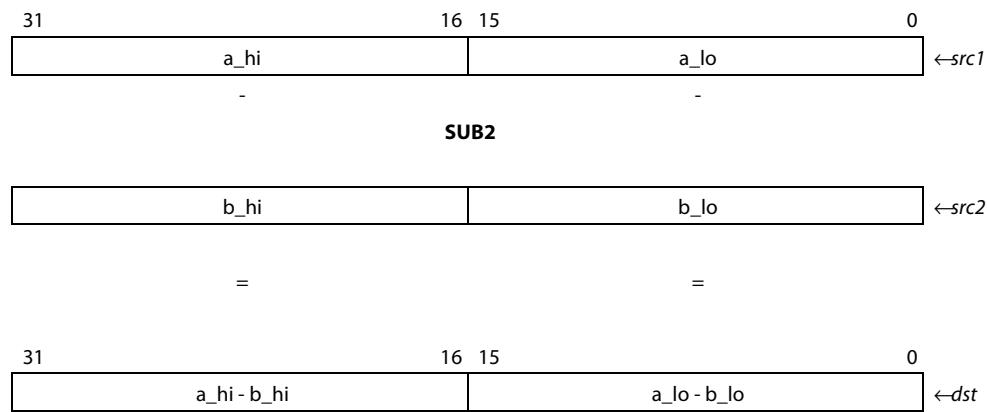
Opcode map field used...	For operand type...	Unit
src1	i2	.S1, .S2
src2	xi2	
dst	i2	

**Opcode** .D unit

31	29	28	27	dst					src2					src1		
3	1			5				5				5				
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	1	0	0	1	0	1	1	1	0	0	s	p		

Opcode map field used...	For operand type...	Unit
<i>src1</i>	i2	.D1, .D2
<i>src2</i>	xi2	
<i>dst</i>	i2	

**Description** The upper and lower halves of *src2* are subtracted from the upper and lower halves of *src1* and the result is placed in *dst*. Any borrow from the lower-half subtraction does not affect the upper-half subtraction. Specifically, the upper-half of *src2* is subtracted from the upper-half of *src1* and placed in the upper-half of *dst*. The lower-half of *src2* is subtracted from the lower-half of *src1* and placed in the lower-half of *dst*.



 **Note**—Unlike the **SUB** instruction, the argument ordering on the .D unit form of .S2 is consistent with the argument ordering for the .L and .S unit forms.

**Execution**

```
if (cond) {
    (lsb16(src1) - lsb16(src2)) → lsb16(dst);
    (msb16(src1) - msb16(src2)) → msb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S, .D

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [ADD2](#), [SUB](#), [SUBU](#), [SUB4](#), [SSUB2](#)

**Examples**

**Example 1**

```
SUB2 .S1 A3, A4, A5
```

**Before instruction****1 cycle after instruction**

A3	<table border="1"><tr><td>1105 6E30h</td></tr></table>	1105 6E30h	4357 28208	A3	<table border="1"><tr><td>1105 6E30h</td></tr></table>	1105 6E30h
1105 6E30h						
1105 6E30h						
A4	<table border="1"><tr><td>1105 6980h</td></tr></table>	1105 6980h	4357 27008	A4	<table border="1"><tr><td>1105 6980h</td></tr></table>	1105 6980h
1105 6980h						
1105 6980h						
A5	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh		A5	<table border="1"><tr><td>0000 04B0h</td></tr></table>	0000 04B0h
xxxx xxxxh						
0000 04B0h						
				0 1200		

**Example 2**

SUB2 .D2 B2, B8, B15

**Before instruction****1 cycle after instruction**

B2	<table border="1"><tr><td>F23A 3789h</td></tr></table>	F23A 3789h	-3526 14217	B2	<table border="1"><tr><td>F23A 3789h</td></tr></table>	F23A 3789h
F23A 3789h						
F23A 3789h						
B8	<table border="1"><tr><td>04B8 6732h</td></tr></table>	04B8 6732h	1208 26418	B8	<table border="1"><tr><td>04B8 6732h</td></tr></table>	04B8 6732h
04B8 6732h						
04B8 6732h						
B15	<table border="1"><tr><td>xxxx xxxxh</td></tr></table>	xxxx xxxxh		B15	<table border="1"><tr><td>ED82 D057h</td></tr></table>	ED82 D057h
xxxx xxxxh						
ED82 D057h						
				-4734 -12201		

**Example 3**

SUB2 .S2X B1,A0,B2

**Before instruction****1 cycle after instruction**

A0	<table border="1"><tr><td>00213271h</td></tr></table>	00213271h	33 <sup>1</sup> 12913 <sup>2</sup>	A0	<table border="1"><tr><td>00213271h</td></tr></table>	00213271h
00213271h						
00213271h						
B1	<table border="1"><tr><td>003A1B48h</td></tr></table>	003A1B48h	58 <sup>1</sup> 6984 <sup>2</sup>	B1	<table border="1"><tr><td>003A1B48h</td></tr></table>	003A1B48h
003A1B48h						
003A1B48h						
B2	<table border="1"><tr><td>xxxxxxxxh</td></tr></table>	xxxxxxxxh		B2	<table border="1"><tr><td>0019E8D7h</td></tr></table>	0019E8D7h
xxxxxxxxh						
0019E8D7h						
				25 <sup>1</sup> -5929 <sup>2</sup>		

1. Signed 16-MSB integer
2. Signed 16-LSB integer

## 4.310 SUB4

Subtract Without Saturation, Four 8-Bit Pairs for Four 8-Bit Results

**Syntax** **SUB4 (.unit) src1, src2, dst**

unit = .L1 or .L2

**Opcode**

31	29	28	27	dst					src2					src1		
3		z													5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
	src1	x	1	1	0	0	1	1	0	1	1	0	s	p		
		5		1									1	1		

Opcode map field used...	For operand type...	Unit
src1	i4	.L1, .L2
src2	xi4	
dst	i4	

**Description**

Performs 2s-complement subtraction between packed 8-bit quantities. The values in *src1* and *src2* are treated as packed 8-bit data and the results are written into *dst* in a packed 8-bit format.

For each pair of 8-bit values in *src1* and *src2*, the difference between the 8-bit value from *src1* and the 8-bit value from *src2* is calculated to produce an 8-bit result. No saturation is performed. The result is placed in the corresponding position in *dst*:

- The difference between *src1* byte0 and *src2* byte0 is placed in byte0 of *dst*.
- The difference between *src1* byte1 and *src2* byte1 is placed in byte1 of *dst*.
- The difference between *src1* byte2 and *src2* byte2 is placed in byte2 of *dst*.
- The difference between *src1* byte3 and *src2* byte3 is placed in byte3 of *dst*.

31	24 23	16 15	8 7	0
a_3	a_2	a_1	a_0	←src1

**SUB4**

b_3	b_2	b_1	b_0	←src2
-----	-----	-----	-----	-------

=                    =                    =                    =

31	24 23	16 15	8 7	0
a_3 - b_3	a_2 - b_2	a_1 - b_1	a_0 - b_0	←dst

**Execution**

```
if (cond) {
    (byte0(src1) - byte0(src2)) → byte0(dst);
    (byte1(src1) - byte1(src2)) → byte1(dst);
    (byte2(src1) - byte2(src2)) → byte2(dst);
    (byte3(src1) - byte3(src2)) → byte3(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src1, src2
Written	dst
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** ADD4, SUB, SUB2

**Example** SUB4 .L1 A2, A8, A9

<b>Before instruction</b>			<b>1 cycle after instruction</b>		
A2	37 89 F2 3Ah	55 137 242 58	A2	37 89 F2 3Ah	
A8	04 B8 49 75h	04 184 73 117	A8	04 B8 49 75h	
A9	xxxx xxxxh		A9	33 D1 A9 C5h	51 -47 169 -59

## 4.311 SWAP2

Swap Bytes in Upper and Lower Register Halves

**Syntax**    **SWAP2 (.unit) src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode**    .L unit

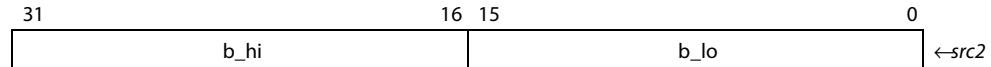
31	29	28	27						23	22						18	17
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					<i>src1</i>	
3		1					5				5					5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
	<i>src1</i>	x	0	0	1	1	0	1	1	1	1	0	s	p			
	5		1										1	1			
<b>Opcode map field used...</b>																	
<b>For operand type...</b>																	
<b>Unit</b>																	
src2																	
dst																	

**Opcode**    .S unit

31	29	28	27						23	22						18	17
<i>creg</i>	<i>z</i>			<i>dst</i>							<i>src2</i>					<i>src1</i>	
3		1					5				5					5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
	<i>src1</i>	x	0	1	0	0	0	0	1	0	0	0	s	p			
	5		1										1	1			
<b>Opcode map field used...</b>																	
<b>For operand type...</b>																	
<b>Unit</b>																	
src2																	
dst																	

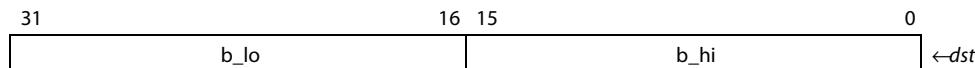
**Description**

The **SWAP2** pseudo-operation takes the lower halfword from *src2* and places it in the upper halfword of *dst*, while the upper halfword from *src2* is placed in the lower halfword of *dst*.



**SWAP2**





The **SWAP2** instruction can be used in conjunction with the **SWAP4** instruction (see [SWAP4](#)) to change the byte ordering (and therefore, the endianess) of 32-bit data.

**Execution**

```
if (cond) {
    msb16(src2) → lsb16(dst);
    lsb16(src2) → msb16(dst)
}
else nop
```

**Pipeline**

Pipeline Stage	E1
Read	src2
Written	dst
Unit in use	.L, .S

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SWAP4](#)

**Examples** [Example 1](#)

SWAP2 .L1 A2,A9

**Before instruction**

A2 3789 F23Ah 14217 -3526

**1 cycle after instruction**

A2 3789 F23Ah  
A9 F23A 3789h -3526 14217

### Example 2

SWAP2 .S2 B2,B12

**Before instruction**

B2 0124 2451h 292 9297

**1 cycle after instruction**

B2 0124 2451h  
B12 2451 0124h 9297 292

4.312 SWAP4

### Swap Byte Pairs in Upper and Lower Register Halves

**Syntax**      **SWAP4** (.unit) *src2*, *dst*

unit = .L1 or .L2

## *Opcode*

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>
<i>src2</i>	xu4	.L1, .L2
<i>dst</i>	u4	

### *Description*

Exchanges pairs of bytes within each halfword of *src2*, placing the result in *dst*. The values in *src2* are treated as unsigned, packed 8-bit values.

Specifically the upper byte in the upper halfword is placed in the lower byte in the upper halfword, while the lower byte of the upper halfword is placed in the upper byte of the upper halfword. Also the upper byte in the lower halfword is placed in the lower byte of the lower halfword, while the lower byte in the lower halfword is placed in the upper byte of the lower halfword.

31	24 23	16 15	8 7	0
ub_3	ub_2	ub_1	ub_0	$\leftarrow src2$

**SWAP4**

↓

31	24 23	16 15	8 7	0
ub_2	ub_3	ub_0	ub_1	←dst

By itself, this instruction changes the ordering of bytes within halfwords. This effectively changes the endianess of 16-bit data packed in 32-bit words. The endianess of full 32-bit quantities can be changed by using the **SWAP4** instruction in conjunction with the **SWAP2** instruction (see [SWAP2](#)).

**Execution**

```
if (cond) {
    ubyte0(src2) → ubyte1(dst);
    ubyte1(src2) → ubyte0(dst);
    ubyte2(src2) → ubyte3(dst);
    ubyte3(src2) → ubyte2(dst)
}
else nop
```

**Pipeline**

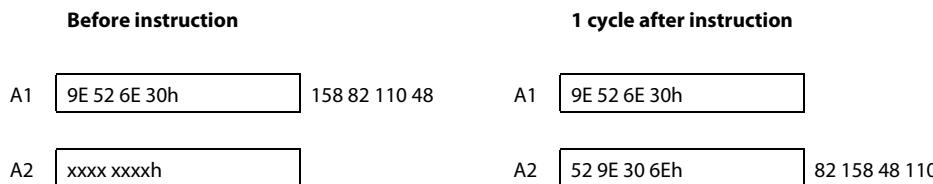
Pipeline Stage	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.L

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SWAP2](#)

**Example** SWAP4 .L1 A1,A2



## 4.313 SWE

Software Exception

**Syntax**    **SWE**

unit = none

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	s	p

1      1

**Description**

Causes an internal exception to be taken. It can be used as a mechanism for User mode programs to request Supervisor mode services. Execution of the **SWE** instruction results in an exception being recognized in the E1 pipeline phase containing the **SWE** instruction. The SXF bit in EFR is set to 1. The HWE bit in NTSR is cleared to 0. If exceptions have been globally enabled, this causes an exception to be recognized before execution of the next execute packet. The address of that next execute packet is placed in NRP.

**Execution**

1 → SXF bit in EFR  
0 → HWE bit in TSR

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also**

[SWENR](#)

## 4.314 SWENR

Software Exception—No Return

**Syntax**

unit = none

**Opcode**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	s	p

1 1

**Description**

Causes an internal exception to be taken. It is intended for use in systems supporting a secure operating mode. It can be used as a mechanism for User mode programs to request Supervisor mode services. It differs from the SWE instruction in four ways:

1. TSR is not copied into NTSR.
2. No return address is placed in NRP (it remains unmodified).
3. The IB bit in TSR is set to 1. This will be observable only in the case where another exception is recognized simultaneously.
4. A branch to REP (restricted entry point register) is forced in the context switch rather than the ISTP-based exception (NMI) vector.

This instruction executes unconditionally.

If another exception (internal or external) is recognized simultaneously with the SWENR-raised exception then the other exception(s) takes priority and normal exception behavior occurs; that is, NTSR and NRP are used and execution is directed to the NMI vector.

**Execution**  
1 → SXF bit in EFR  
0 → HWE bit in TSR

**Instruction Type** Single-cycle

**Delay Slots** 0

**See Also** [SWE](#)

## 4.315 UNPKBU4

Unpack All Unsigned Packed 8-bit to Unsigned Packed 16-bit

**Syntax** **UNPKBU4 (.unit) src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode** Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	1	1	1	1	0	0	1	0	0	0	s	p	

3                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.S1 or .S2	01000

**Opcode** Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.L1 or .L2	01000

**Description** The UNPKBU4 instruction unpacks the unsigned bytes of *xop* into the half-words of *dwdst*.

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0					
								IJKLMNOP	ijklmnop	ABCDEFGHI	ABCDEFGHI									
00000000	IJKLMNOP	00000000	ijklmnop	00000000	ABCDEFGHI	00000000	absdefgh													

v                    v

←op              ←dst

**Execution**

```

byte0(src1) -> byte0(dst_e);
0 -> byte1(dst_e);
byte1(src2) -> byte2(dst_e);
0 -> byte3(dst_e);
byte2(src1) -> byte4(dst_o);
0 -> byte5(dst_o);
byte3(src2) -> byte6(dst_o);
0 -> byte7(dst_o);

```

<b>Instruction Type</b>	Single cycle
<b>Delay Slots</b>	0
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">UNPKHU4</a>
<b>Example</b>	A3 == 0xaabb778d UNPKBU4 .S A3:A2A1 A2 == 0x00aa00bb A1 == 0x0077008d

## 4.316 UNPKH2

Unpack High Signed Packed 16-bit to Packed 32-bit

**Syntax** **UNPKH2 (.unit) src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode** Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	1	1	1	1	0	0	1	0	0	0	s	p	

3                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.S1 or .S2	00111

**Opcode** Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                    5                    5                    5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.L1 or .L2	00111

**Description** The UNPKH2 instruction extracts the 2 signed 16-bit integers in xop and expands each to a 32-bit value.

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0							
aaaaaaaa	abcdefgh	iiiiiiii	ijklmnop	AAAAAAA	ABCDEFGH	IIIIIIII	IJKLMNOP		abcdefgh	ijklmnop	ABCDEFGHI	IJKLMNOP	v	v								← xop
																						← dwdst

```
Execution if(cond) {
    slsb16(src1) -> dst_e
    smsb16(src1) -> dst_o
}
else nop
```

---

<b>Instruction Type</b>	Single cycle
<b>Delay Slots</b>	0
<b>Functional Unit Latency</b>	1
<b>See Also</b>	<a href="#">UNPKLU4</a>
<b>Example</b>	A3 == 0x82c47688 UNPKH2 .S A3,A1:A0 A1 == 0xFFFF82c4 A0 == 0x00007688

## 4.317 UNPKHU2

Unpack High Unsigned Packed 16-bit to Packed 32-bit

**Syntax** **UNPKHU2 (.unit) src2, dst**

unit = .S1, .S2, .L1, or .L2

**Opcode** Opcode for .S Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	1	1	1	1	0	0	1	0	0	0	s	p	

3                            5                            5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.S1 or .S2	00110

**Opcode** Opcode for .L Unit, 1 src

31	29	28	27	23	22	18	17	13	12	11	10	9	8	7	6	5	4	3	2	1	0
creg	z		dst		src2		opfield	x	0	0	1	1	0	1	0	1	1	0	s	p	

3                            5                            5

Opcode map field used...	For operand type...	Unit	Opfield
src2,dst	xop,dwdst	.L1 or .L2	00110

**Description** The UNPKHU2 instruction extracts the 2 unsigned 16-bit integers in xop and expands each to a 32-bit value.

63	56	55	48	47	40	39	32	31	24	23	16	15	8	7	0					
								abcdefgh	ijklmnop	ABCDEF GH	IJKLMNOP					v	v			
00000000	abcdefgh	00000000	ijklmnop	00000000	ABCDEF GH	00000000	IJKLMNOP									← dwdst	← xop			

**Execution**

```

if(cond) {
    ulsb16(src1) -> lsb16(dst_e)
    0 -> msb16(dst_e)
    umsb16(src1) -> lsb16(dst_o)
    0 -> msb16(dst_o)
}
else nop

```

**Instruction Type** Single cycle

**Delay Slots** 0

**Functional Unit Latency** 1

**See Also** [UNPKLU4](#)

**Example**

```
A3 == 0x82c47688
UNPKHU2 .S A3,A1:A0
A1 == 0x000082c4
A0 == 0x00007688
```

```
A3 == 0xfffffffff
UNPKHU2 .S A3,A1:A0
A1 == 0x0000ffff
A0 == 0x0000ffff
```

## 4.318 UNPKHU4

Unpack 16 MSB Into Two Lower 8-Bit Halfwords of Upper and Lower Register Halves

**Syntax** **UNPKHU4** (.unit) *src2*, *dst*

unit = .L1, .L2, .S1, .S2

**Opcode** .L unit

31	29	28	27	dst					src2					18	17	16
3	1			5					5					0	0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	x	0	0	1	1	0	1	0	1	1	0	s	p	
								1						1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xu4	.L1, .L2
<i>dst</i>	u2	

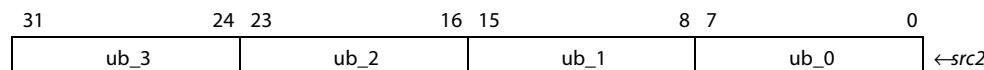
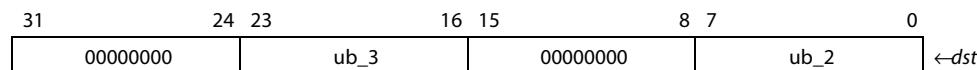
**Opcode** .S unit

31	29	28	27	dst					src2					18	17	16
3	1			5					5					0	0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	x	1	1	1	1	0	0	1	0	0	0	s	p	
								1						1	1	

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xu4	.S1, .S2
<i>dst</i>	u2	

**Description** Moves the two most-significant bytes of *src2* into the two low bytes of the two halfwords of *dst*.

Specifically the upper byte in the upper halfword is placed in the lower byte in the upper halfword, while the lower byte of the upper halfword is placed in the lower byte of the lower halfword. The *src2* bytes are zero-extended when unpacked, filling the two high bytes of the two halfwords of *dst* with zeros.

**UNPKHU4**

**Execution**

```

if (cond) {
    ubyte3(src2) → ubyte2(dst);
    0 → ubyte3(dst);
    ubyte2(src2) → ubyte0(dst);
    0 → ubyte1(dst)
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

**Instruction Type** Single cycle

**Delay Slots** 0

**See Also** [UNPKLU4](#)

**Examples** [Example 1](#)

UNPKHU4 .L1 A1,A2

**Before instruction**

A1 9E 52 6E 30h

A2 xxxx xxxxh

**1 cycle after instruction**

A1 9E 52 6E 30h

A2 00 9E 00 52h

**Example 2**

UNPKHU4 .L2 B17,B18

**Before instruction**

B17 11 05 69 34h

**1 cycle after instruction**

B17 11 05 69 34h

B18 B18

## 4.319 UNPKLU4

Unpack 16 LSB Into Two Lower 8-Bit Halfwords of Upper and Lower Register Halves

**Syntax** **UNPKLU4 (.unit) src2, dst**

unit = .L1, .L2, .S1, .S2

**Opcode** .L unit

31	29	28	27	dst					src2					18	17	16
3	1			5					5					0	0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	x	0	0	1	1	0	1	0	1	1	0	s	p	
								1						1	1	

Opcode map field used...	For operand type...	Unit
src2	xu4	.L1, .L2
dst	u2	

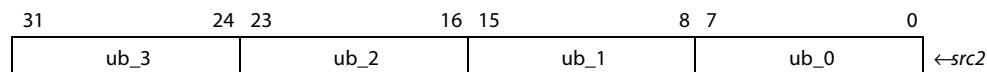
**Opcode** .S unit

31	29	28	27	dst					src2					18	17	16
3	1			5					5					0	0	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	x	1	1	1	1	0	0	1	0	0	0	s	p	
								1						1	1	

Opcode map field used...	For operand type...	Unit
src2	xu4	.S1, .S2
dst	u2	

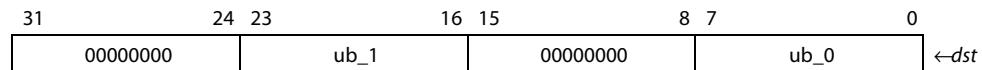
**Description** Moves the two least-significant bytes of *src2* into the two low bytes of the two halfwords of *dst*.

Specifically, the upper byte in the lower halfword is placed in the lower byte in the upper halfword, while the lower byte of the lower halfword is kept in the lower byte of the lower halfword. The *src2* bytes are zero-extended when unpacked, filling the two high bytes of the two halfwords of *dst* with zeros.



### UNPKLU4

↓



**Execution**

```

if (cond) {
    ubyte0(src2) → ubyte0(dst);
    0 → ubyte1(dst);
    ubyte1(src2) → ubyte2(dst);
    0 → ubyte3(dst);
}
else nop

```

**Pipeline**

Pipeline Stage	E1
Read	<i>src2</i>
Written	<i>dst</i>
Unit in use	.L, .S

**Instruction Type** Single cycle

**Delay Slots** 0

**See Also** [UNPKHU4](#)

**Examples** [Example 1](#)

UNPKLU4 .L1 A1,A2

**Before instruction**

A1 

9E 52 6E 30h
--------------

A2 

xxxx xxxxh
------------

**1 cycle after instruction**

A1 

9E 52 6E 30h
--------------

A2 

00 6E 00 30h
--------------

### Example 2

UNPKLU4 .L2 B17,B18

**Before instruction**

B17 

11 05 69 34h
--------------

**1 cycle after instruction**

B17 

11 05 69 34h
--------------

B18

xxxx xxxxh

B18

00 69 00 34h

## 4.320 XOR

Bitwise Exclusive OR

**Syntax**    **XOR (.unit) src1, src2, dst**

unit = .L1, .L2, .S1, .S2, .D1, .D2

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opcode	1	1	0	s	p	

3                                 5                                 5                                 5                                 7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.L1 or .L2	1101111

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opcode	1	1	0	s	p

5                                 5                                 5                                 7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwo1,xdwop2,dwdst	.L1 or .L2	1101111

**Opcode**    Opcode for .L Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opcode	1	1	0	s	p	

3                                 5                                 5                                 5                                 7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	scst5,xop2,dst	.L1 or .L2	1101110

**Opcode**    Opcode for .L Unit, 1/2 src - same as L2S but fixed hdr, bit 23-msb of opcode

31	30	29	28	27	23	22	18	17	13	12	11	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opcode	1	1	0	s	p

5                                 5                                 5                                 7

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	scst5,xdwop2,dwdst	.L1 or .L2	1101110

**Opcode**    Opcode for .S Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opcode	1	0	0	0	s	p	
3			5		5		5			6							

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	op1,xop2,dst	.S1 or .S2	001011

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opcode	1	0	0	0	s	p
					5		5		5			6						

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	dwop1,xdwop2,dwdst	.S1 or .S2	001011

**Opcode**    Opcode for .S Unit, 1/2 src

31	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x		opcode	1	0	0	0	s	p	
3			5		5		5			6							

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	scst5,xop2,dst	.S1 or .S2	001010

**Opcode**    Opcode for .S Unit, 1/2 src, unconditional

31	30	29	28	27	23	22	18	17	13	12	11	6	5	4	3	2	1	0
0	0	0	1		dst		src2		src1	x		opcode	1	0	0	0	s	p
					5		5		5			6						

Opcode map field used...	For operand type...	Unit	Opfield
src1,src2,dst	scst5,xdwop2,dwdst	.S1 or .S2	001010

**Opcode**    Opcode for .D Unit, 2 src

31	29	28	27	23	22	18	17	13	12	11	10	9	6	5	4	3	2	1	0
creg	z		dst		src2		src1	x	1	0		opcode	1	1	0	0	s	p	
3			5		5		5					4							

<b>Opcode map field used...</b>	<b>For operand type...</b>	<b>Unit</b>	<b>Opfield</b>
src1,src2,dst	op1,xop2,dst	.D1 or .D2	1110
src1,src2,dst	scst5,xop2,dst	.D1 or .D2	1111

**Description** The XOR instruction performs the bit-wise XOR between the two source registers and stores the result in the destination register. Note that one can use the constant form of XOR to produce the one's compliment of a number. The XOR instruction may also be used to clear a register.

**Execution**

```
if(cond) src1 XOR src2 -> dst
else nop
```

**Instruction Type** Single cycle

**Delay Slots** 0

**Functional Unit Latency** 1

#### See Also

**Example**

```
A0 == 0x05af0137
XOR .L -1,A0,A15 ; Negate A1 (1's compliment)
A15 == 0xfa50fec8
```

```
A0 == 0xbe10fa31
A1 == 0x00ff00ff
XOR .L A0,A1,A15
A15 == 0xbeefface
```

```
A1 == 0x05af0137
A0 == 0x05af0137
XOR .L -1,A1:A0,A15:A14 ; Negate A1 (1's compliment)
A15 == 0xfa50fec8
A14 == 0xfa50fec8
```

```
A1 == 0xbe10fa31
A0 == 0xbe10fa31
A3 == 0x00ff00ff
A2 == 0x00ff00ff
XOR .L A1:A0,A3:A2,A15:A14
A15 == 0xbeefface
A14 == 0xbeefface
```

## 4.321 XORMPY

Galois Field Multiply With Zero Polynomial

**Syntax** **XORMPY** (.unit) *src1, src2, dst*

unit = .M1 or .M2

**Opcode**

31	30	29	28	27	dst					src2					src1				
0	0	0	1		5					5					5				
						13	12	11	10	9	8	7	6	5	4	3	2	1	0
						<i>src1</i>	x	0	1	1	0	1	1	1	1	0	0	s	p
						5		1								1		1	

Opcode map field used...	For operand type...	Unit
<i>src1</i>	uint	.M1, .M2
<i>src2</i>	xuint	
<i>dst</i>	uint	

**Description**

Performs a Galois field multiply, where *src1* is 32 bits and *src2* is limited to 9 bits. This multiply connects all levels of the gmpy4 together and only extends out by 8 bits. The **XORMPY** instruction is identical to a **GMPY** instruction executed with a zero-value polynomial.

```
uword xormpy(uword src1,uword src2)
{
    // the multiply is always between GF(2^9) and GF(2^32)
    // so no size information is needed

    uint pp;
    uint mask, tpp;
    uint I;
    pp = 0;
    mask = 0x00000100; // multiply by computing
                        // partial products.
    for ( I=0; i<8; I++ ){
        if ( src2 & mask ) pp ^= src1;
        mask >>= 1;
        pp <<= 1;
    }
    if ( src2 & 0x1 ) pp ^= src1;
    return (pp); // leave it asserted left.
}
```

**Execution**

GMPY\_poly = 0  
*(lsb9(src2) gmpy uint(src1)) → uint(dst)*

**Instruction Type**

Four-cycle

**Delay Slots**

3

**See Also** [GMPY4](#), [GMPY](#), [XOR](#)

**Example** XORMPY .M1 A0,A1,A2 GPLYA = FFFFFFFF (ignored)

**Before instruction**A0 12345678hA1 00000126h**1 cycle after instruction**A2 1E654210h

## 4.322 XPND2

Expand Bits to Packed 16-Bit Masks

**Syntax** **XPND2 (.unit) src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst						src2						18	17	16
creg	z			dst						src2						1	1	
3	1			5						5								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	0	1	x	0	0	0	0	1	1	1	1	0	0	s	p			
				1										1	1			

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.M1, .M2
<i>dst</i>	uint	

**Description**

Reads the two least-significant bits of *src2* and expands them into two halfword masks written to *dst*. Bit 1 of *src2* is replicated and placed in the upper halfword of *dst*. Bit 0 of *src2* is replicated and placed in the lower halfword of *dst*. Bits 2 through 31 of *src2* are ignored.

31	24 23	16 15	8 7	0
XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXX10	↔src2

**XPND2**

↓

31	24 23	16 15	8 7	0
11111111	11111111	00000000	00000000	↔dst

The **XPND2** instruction is useful, when combined with the output of the **CMPGT2** or **CMPEQ2** instruction, for generating a mask that corresponds to the individual halfword positions that were compared. That mask may then be used with **ANDN**, **AND**, or **OR** instructions to perform other operations like compositing. This is an example:

```
CMPGT2.S1A3, A4, A5 ; Compare two registers, both upper
; and lower halves.
```

```
XPND2 .M1A5, A2 ; Expand the compare results into
; two 16-bit masks.
```

NOP

```
AND      .D1 A2, A7, A8 ; Apply the mask to a value to create result.
```

Because the **XPND2** instruction only examines the two least-significant bits of *src2*, it is possible to store a large bit mask in a single 32-bit word and expand it using multiple **SHR** and **XPND2** instruction pairs. This can be useful for expanding a packed 1-bit-per-pixel bitmap into full 16-bit pixels in imaging applications.

**Execution**

```
if(src2 & 1) 0xFFFF -> lsb16(dst)
else 0x0000 -> lsb16(dst)
if(src2 & 2) 0xFFFF -> msb16(dst)
else 0x0000 -> msb16(dst)
```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

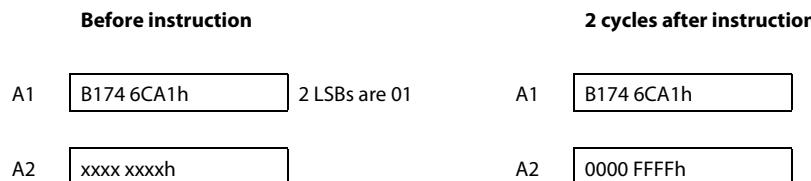
**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [CMPEQ2](#), [CMPGT2](#), [XPND4](#)

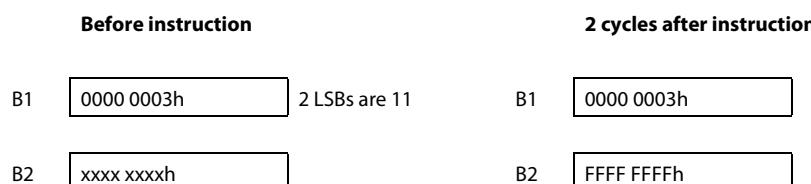
**Examples** [Example 1](#)

XPND2 .M1 A1,A2



[Example 2](#)

XPND2 .M2 B1,B2



## 4.323 XPND4

Expand Bits to Packed 8-Bit Masks

**Syntax** **XPND4 (.unit) src2, dst**

unit = .M1 or .M2

**Opcode**

31	29	28	27	dst						src2						18	17	16
creg	z			dst						src2						1	1	
3	1			5						5								
0	0	0	x	0	0	0	0	1	1	1	1	1	0	0	s	p		
				1											1	1		

Opcode map field used...	For operand type...	Unit
<i>src2</i>	xuint	.M1, .M2
<i>dst</i>	uint	

**Description**

Reads the four least-significant bits of *src2* and expands them into four-byte masks written to *dst*. Bit 0 of *src2* is replicated and placed in the least-significant byte of *dst*. Bit 1 of *src2* is replicated and placed in second least-significant byte of *dst*. Bit 2 of *src2* is replicated and placed in second most-significant byte of *dst*. Bit 3 of *src2* is replicated and placed in most-significant byte of *dst*. Bits 4 through 31 of *src2* are ignored.

31	24 23	16 15	8 7	0
XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXX1001	←src2

**XPND4**

↓

31	24 23	16 15	8 7	0
11111111	00000000	00000000	11111111	←dst

The **XPND4** instruction is useful, when combined with the output of the **CMPGT4** or **CMPEQ4** instruction, for generating a mask that corresponds to the individual byte positions that were compared. That mask may then be used with **ANDN**, **AND**, or **OR** instructions to perform other operations like compositing.

This is an example:

CMPEQ4.S1A3, A4, A5 ; Compare two 32-bit registers all four bytes.

XPND4 .M1A5, A2 ; Expand the compare results into  
; four 8-bit masks.

NOP

AND .D1A2, A7, A8 ; Apply the mask to a value to create result.

Because the **XPND4** instruction only examines the four least-significant bits of *src2*, it is possible to store a large bit mask in a single 32-bit word and expand it using multiple **SHR** and **XPND4** instruction pairs. This can be useful for expanding a packed, 1-bit-per-pixel bitmap into full 8-bit pixels in imaging applications.

**Execution**

```

if(src2 & 1) 0xFF -> byte0(dst)
else 0x00 -> byte0(dst)

if(src2 & 2) 0xFF -> byte1(dst)
else 0x00 -> byte1(dst)

if(src2 & 4) 0xFF -> byte2(dst)
else 0x00 -> byte2(dst)

if(src2 & 8) 0xFF -> byte3(dst_e)
else 0x00 -> byte3(dst_e)

```

**Pipeline**

Pipeline Stage	E1	E2
Read	<i>src2</i>	
Written		<i>dst</i>
Unit in use	.M	

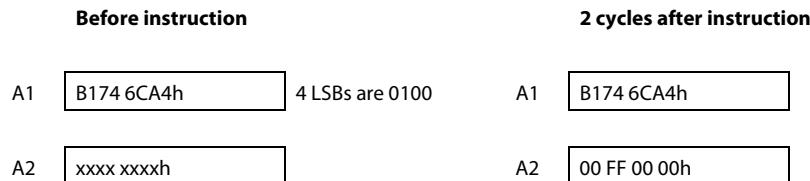
**Instruction Type** Two-cycle

**Delay Slots** 1

**See Also** [CMPEQ4](#), [CMPGTU4](#), [XPND2](#)

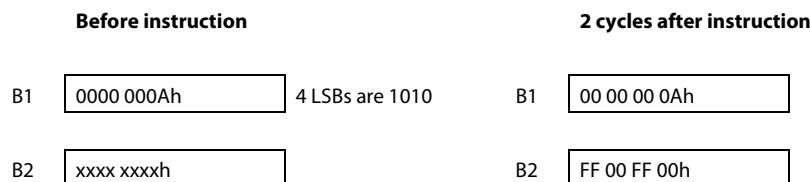
**Examples** **Example 1**

XPND4 .M1 A1,A2



**Example 2**

XPND4 .M2 B1,B2



## 4.324 ZERO

Zero a Register

**Syntax** **ZERO** (.unit) *dst*

or

**ZERO** (.unit) *dst\_o:dst\_e*

unit = .L1, .L2, .D1, .D2, .S1, .S2

**Opcode**

Opcode map field used...	For operand type...	Unit	Opfield
<i>dst</i>	sint	.L1, .L2	0010111
<i>dst</i>	slong	.L1, .L2	0110111
<i>dst</i>	sint	.D1, .D2	010001
<i>dst</i>	sint	.S1, .S2	010111

**Description**

This is a pseudo-operation used to fill the destination register or register pair with 0s.

When the destination is a single register, the assembler uses the **MVK** instruction to load it with zeros: **MVK** (.unit) 0, *dst*

When the destination is a register pair, the assembler uses the **SUB** instruction to subtract a value from itself and store the result in the destination pair.

**Execution**

if (cond) 0 → *dst* else nop

or

if (cond) *src* - *src* → *dst\_o:dst\_e* else nop

**Instruction Type**

Single-cycle

**Delay Slots**

0

**See Also** [MVK](#), [SUB](#)

**Examples** [Example 1](#)

**ZERO .D1 A1**

**Before instruction**

A1	B174 6CA1h
----	------------

**1 cycle after instruction**

A1	0000 0000h
----	------------

**Example 2**

**ZERO .L1 A1:A0**

**Before instruction****1 cycle after instruction**A0    

B174 6CA1h
------------

A0    

0000 0000h
------------

A1    

1234 5678h
------------

A1    

0000 0000h
------------

# Pipeline

The DSP pipeline provides flexibility to simplify programming and improve performance. These two factors provide this flexibility:

1. Control of the pipeline is simplified by eliminating pipeline interlocks.
2. Increased pipelining eliminates traditional architectural bottlenecks in program fetch, data access, and multiply operations. This provides single-cycle throughput.

This chapter starts with a description of the pipeline flow. Highlights are:

- The pipeline can dispatch eight parallel instructions every cycle.
- Parallel instructions proceed simultaneously through each pipeline phase.
- Serial instructions proceed through the pipeline with a fixed relative phase difference between instructions.
- Load and store addresses appear on the CPU boundary during the same pipeline phase, eliminating read-after-write memory conflicts.

All instructions require the same number of pipeline phases for fetch and decode, but require a varying number of execute phases. This chapter contains a description of the number of execution phases for each type of instruction.

Finally, this chapter contains performance considerations for the pipeline. These considerations include the occurrence of fetch packets that contain multiple execute packets, execute packets that contain multicycle NOPs, and memory considerations for the pipeline. For more information about fully optimizing a program and taking full advantage of the pipeline, see the *TMS320C6000 Programmer's Guide* ([SPRU198](#)).

- 5.1 "Pipeline Operation Overview" on page 5-2
- 5.2 "Pipeline Execution of Instruction Types" on page 5-9
- 5.3 "Functional Unit Constraints" on page 5-24
- 5.4 "Performance Considerations" on page 5-43

## 5.1 Pipeline Operation Overview

The pipeline phases are divided into three stages:

- Fetch
- Decode
- Execute

All instructions in the DSP pipeline are shown in [Figure 5-1](#).

**Figure 5-1 Pipeline Stages**

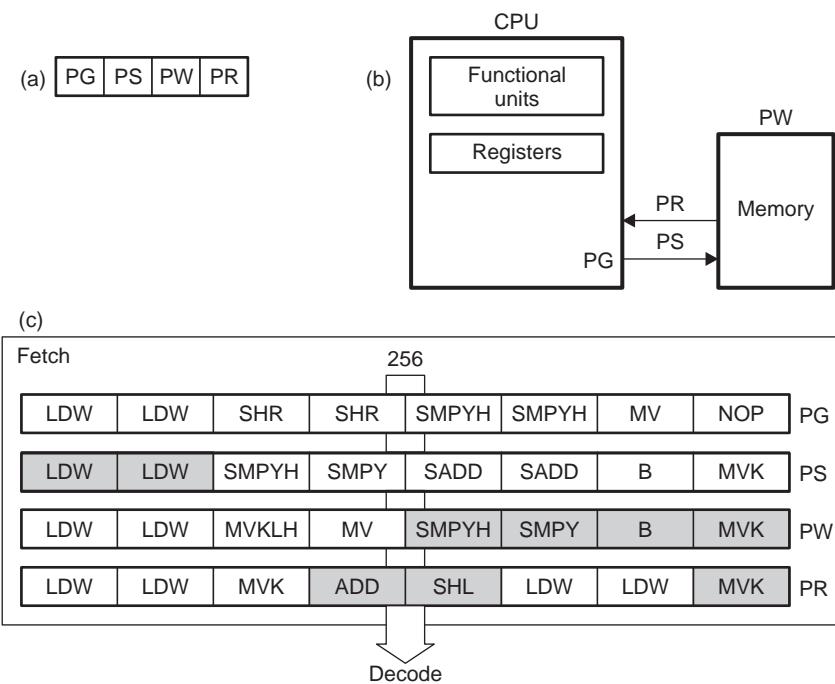


### 5.1.1 Fetch

The fetch phases of the pipeline are:

- **PG:** Program address generate
- **PS:** Program address send
- **PW:** Program access ready wait
- **PR:** Program fetch packet receive

The DSP uses a fetch packet (FP) of eight words. All eight of the words proceed through fetch processing together, through the PG, PS, PW, and PR phases. [Figure 5-2\(a\)](#) shows the fetch phases in sequential order from left to right. [Figure 5-2\(b\)](#) is a functional diagram of the flow of instructions through the fetch phases. During the PG phase, the program address is generated in the CPU. In the PS phase, the program address is sent to memory. In the PW phase, a memory read occurs. Finally, in the PR phase, the fetch packet is received at the CPU. [Figure 5-2\(c\)](#) shows fetch packets flowing through the phases of the fetch stage of the pipeline. In [Figure 5-2\(c\)](#), the first fetch packet (in PR) is made up of four execute packets, and the second and third fetch packets (in PW and PS) contain two execute packets each. The last fetch packet (in PG) contains a single execute packet of eight instructions.

**Figure 5-2 Fetch Phases of the Pipeline**

### 5.1.2 Decode

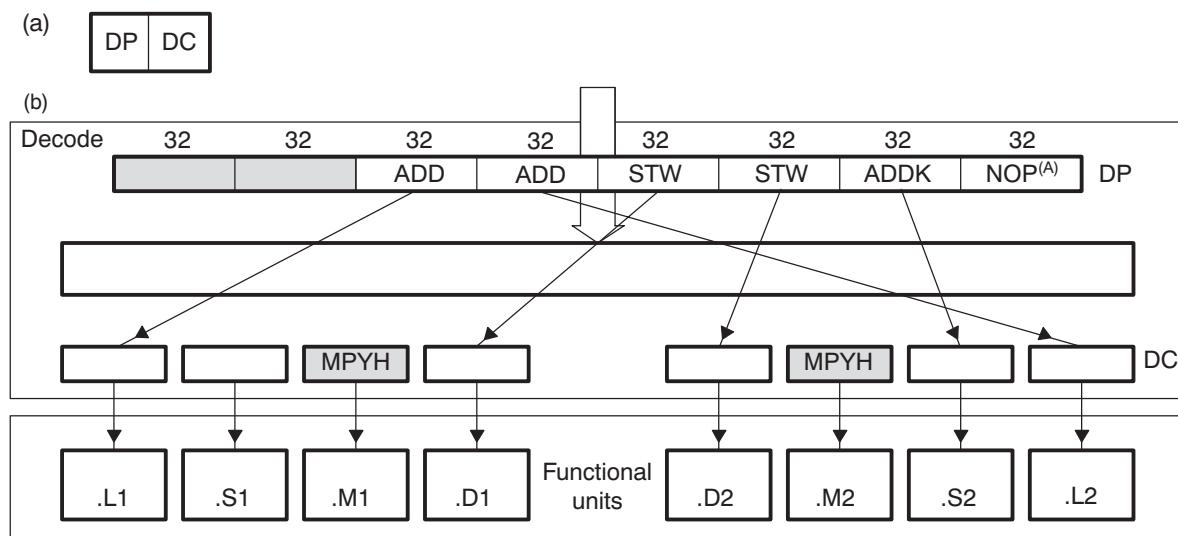
The decode phases of the pipeline are:

- **DP:** Instruction dispatch
- **DC:** Instruction decode

In the DP phase of the pipeline, the fetch packets are split into execute packets. Execute packets consist of one instruction or from two to eight parallel instructions. During the DP phase, the instructions in an execute packet are assigned to the appropriate functional units. In the DC phase, the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units.

**Figure 5-3(a)** shows the decode phases in sequential order from left to right. **Figure 5-3(b)** shows a fetch packet that contains two execute packets as they are processed through the decode stage of the pipeline. The last six instructions of the fetch packet (FP) are parallel and form an execute packet (EP). This EP is in the dispatch phase (DP) of the decode stage. The arrows indicate each instruction's assigned functional unit for execution during the same cycle. The **NOP** instruction in the eighth slot of the FP is not dispatched to a functional unit because there is no execution associated with it.

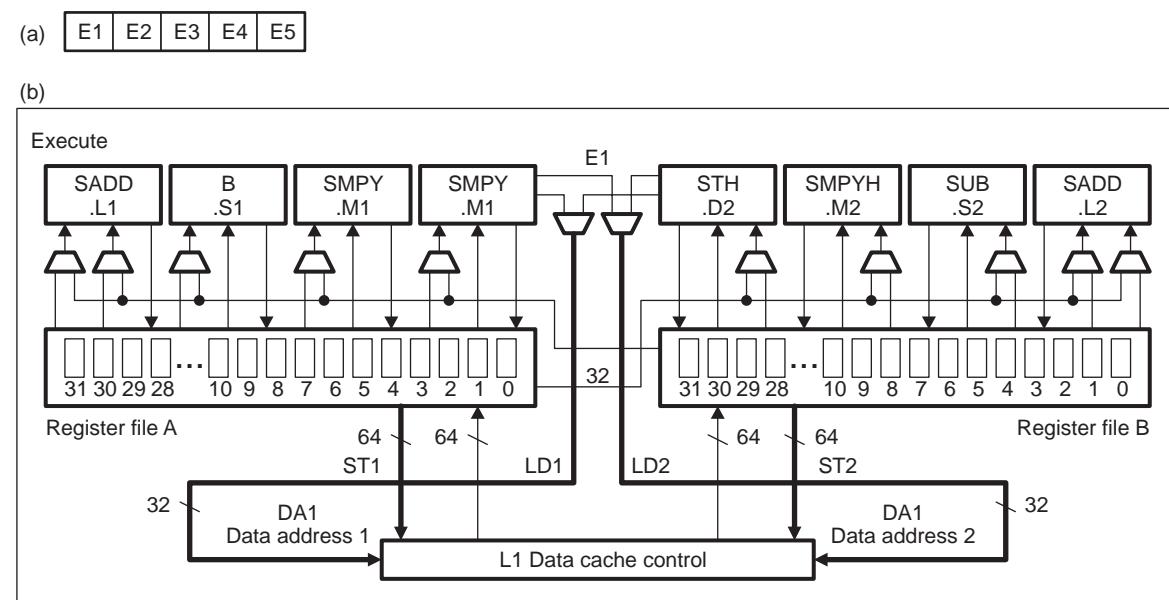
The first two slots of the fetch packet (shaded below) represent an execute packet of two parallel instructions that were dispatched on the previous cycle. This execute packet contains two **MPY** instructions that are now in decode (DC) one cycle before execution. There are no instructions decoded for the .L, .S, and .D functional units for the situation illustrated.

**Figure 5-3 Decode Phases of the Pipeline**


(A) NOP is not dispatched to a functional unit.

### 5.1.3 Execute

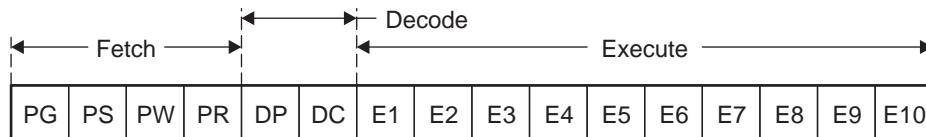
The execute portion of the pipeline is subdivided into five phases (E1-E5). Different types of instructions require different numbers of these phases to complete their execution. These phases of the pipeline play an important role in your understanding of the device state at CPU cycle boundaries. The execution of different types of instructions in the pipeline is described in “[Pipeline Execution of Instruction Types](#)” on page 5-9. [Figure 5-4\(a\)](#) shows the execute phases of the pipeline in sequential order from left to right. [Figure 5-3\(b\)](#) shows the portion of the functional block diagram in which execution occurs.

**Figure 5-4 Execute Phases of the Pipeline**


### 5.1.4 Pipeline Operation Summary

[Figure 5-5](#) shows all the phases in each stage of the pipeline in sequential order, from left to right.

**Figure 5-5 Pipeline Phases**



[Figure 5-6](#) shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion through each phase of the pipeline.

For example, examine cycle 7 in [Figure 5-6](#). When the instructions from FPn reach E1, the instructions in the execute packet from FP n + 1 are being decoded. FP n + 2 is in dispatch while FPs n + 3, n + 4, n + 5, and n + 6 are each in one of four phases of program fetch. See “[Performance Considerations](#)” on page 5-43 for additional detail on code flowing through the pipeline. [Table 5-1](#) on page 5-5 summarizes the pipeline phases and what happens in each phase.

**Figure 5-6 Pipeline Operation: One Execute Packet per Fetch Packet**

Fetch Packet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	
n+1		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
n+2			PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8	E9
n+3				PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7	E8
n+4					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6	E7
n+5						PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
n+6							PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
n+7								PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+8									PG	PS	PW	PR	DP	DC	E1	E2	E3
n+9										PG	PS	PW	PR	DP	DC	E1	E2
n+10											PG	PS	PW	PR	DP	DC	E1

**Table 5-1 Operations Occurring During Pipeline Phases (Part 1 of 2)**

Stage	Phase	Symbol	During This Phase
Program fetch	Program address generate	PG	The address of the fetch packet is determined.
	Program address send	PS	The address of the fetch packet is sent to memory.
	Program wait	PW	A program memory access is performed.
	Program data receive	PR	The fetch packet is at the CPU boundary.
Program decode	Dispatch	DP	The next execute packet in the fetch packet is determined and sent to the appropriate functional units to be decoded.
	Decode	DC	Instructions are decoded in functional units.

**Table 5-1 Operations Occurring During Pipeline Phases (Part 2 of 2)**

<b>Stage</b>	<b>Phase</b>	<b>Symbol</b>	<b>During This Phase</b>
Execute	Execute 1	E1	<p>For all instruction types, the conditions for the instructions are evaluated and operands are read.</p> <p>For load and store instructions, address generation is performed and address modifications are written to a register file.<sup>1</sup></p> <p>For branch instructions, branch fetch packet in PG phase is affected.<sup>1</sup></p> <p>For single-cycle instructions, results are written to a register file.<sup>1</sup></p> <p>For DP compare, ADDDP/SUBDP, and MPYDP instructions, the lower 32-bits of the sources are read. For all other instructions, the sources are read.<sup>1</sup></p> <p>For MPYSPDP instruction, the <i>src1</i> and the lower 32 bits of <i>src2</i> are read.<sup>1</sup></p> <p>For 2-cycle DP instructions, the lower 32 bits of the result are written to a register file.<sup>1</sup></p>
	Execute 2	E2	<p>For load instructions, the address is sent to memory. For store instructions, the address and data are sent to memory.<sup>1</sup></p> <p>Single-cycle instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.<sup>1</sup></p> <p>For multiply unit, nonmultiply instructions, results are written to a register file.<sup>2</sup></p> <p>For multiply, 2-cycle DP, and DP compare instructions, results are written to a register file.<sup>1</sup></p> <p>For DP compare and ADDDP/SUBDP instructions, the upper 32 bits of the source are read.<sup>1</sup></p> <p>For MPYDP instruction, the lower 32 bits of <i>src1</i> and the upper 32 bits of <i>src2</i> are read.<sup>1</sup></p> <p>For MPYI and MPYID instructions, the sources are read.<sup>1</sup></p> <p>For MPYSPDP instruction, the <i>src1</i> and the upper 32 bits of <i>src2</i> are read.<sup>1</sup></p>
	Execute 3	E3	<p>Data memory accesses are performed. Any multiply instructions that saturate results set the SAT bit in the control status register (CSR) if saturation occurs.<sup>1</sup></p> <p>For MPYDP instruction, the upper 32 bits of <i>src1</i> and the lower 32 bits of <i>src2</i> are read.<sup>1</sup></p> <p>For MPYI and MPYID instructions, the sources are read.<sup>1</sup></p>
	Execute 4	E4	<p>For load instructions, data is brought to the CPU boundary.<sup>1</sup></p> <p>For multiply extensions, results are written to a register file.<sup>3</sup></p> <p>For MPYI and MPYID instructions, the sources are read.<sup>1</sup></p> <p>For MPYDP instruction, the upper 32 bits of the sources are read.<sup>1</sup></p> <p>For MPYI and MPYID instructions, the sources are read.<sup>1</sup></p> <p>For 4-cycle instructions, results are written to a register file.<sup>1</sup></p> <p>For INTDP and MPYSP2DP instructions, the lower 32 bits of the result are written to a register file.<sup>1</sup></p>
	Execute 5	E5	<p>For load instructions, data is written into a register.<sup>1</sup></p> <p>For INTDP and MPYSP2DP instructions, the upper 32 bits of the result are written to a register file.<sup>1</sup></p>
	Execute 6	E6	For ADDDP/SUBDP and MPYSPDP instructions, the lower 32 bits of the result are written to a register file. <sup>1</sup>
	Execute 7	E7	For ADDDP/SUBDP and MPYSPDP instructions, the upper 32 bits of the result are written to a register file. <sup>1</sup>
	Execute 8	E8	Nothing is read or written.
	Execute 9	E9	<p>For MPYI instruction, the result is written to a register file.<sup>1</sup></p> <p>For MPYDP and MPYID instructions, the lower 32 bits of the result are written to a register file.<sup>1</sup></p>
	Execute 10	E10	For MPYDP and MPYID instructions, the upper 32 bits of the result are written to a register file. <sup>1</sup>

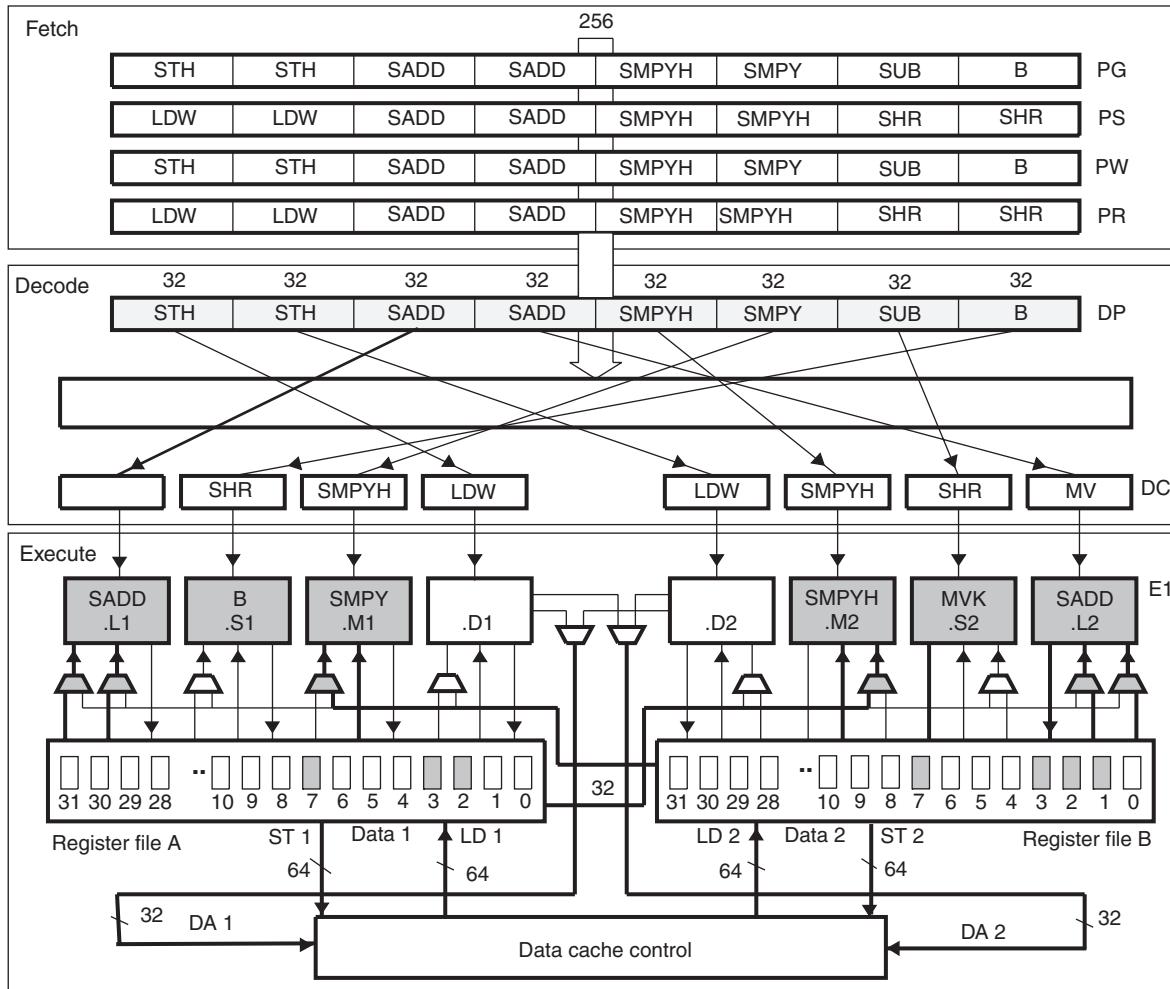
1. This assumes that the conditions for the instructions are evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

2. Multiply unit, nonmultiply instructions are **AVG2**, **AVG4**, **BITC4**, **BITR**, **DEAL**, **ROT**, **SHFL**, **SSHVL**, and **SSHVR**.

3. Multiply extensions include **MPY2**, **MPY4**, **DOTPx2**, **DOTPU4**, **MPYHlx**, **MPYLlx**, and **MVD**.

[Figure 5-7](#) shows a functional block diagram of the pipeline stages. The pipeline operation is based on CPU cycles. A CPU cycle is the period during which a particular execute packet is in a particular pipeline phase. CPU cycle boundaries always occur at clock cycle boundaries.

**Figure 5-7 Pipeline Phases Block Diagram**



As code flows through the pipeline phases, it is processed by different parts of the DSP. [Figure 5-7](#) shows a full pipeline with a fetch packet in every phase of fetch. One execute packet of eight instructions is being dispatched at the same time that a 7-instruction execute packet is in decode. The arrows between DP and DC correspond to the functional units identified in the code in [Example 5-1](#).

In the DC phase portion of [Figure 5-7](#), one box is empty because a **NOP** was the eighth instruction in the fetch packet in DC and no functional unit is needed for a **NOP**. Finally, [Figure 5-7](#) shows six functional units processing code during the same cycle of the pipeline.

Registers used by the instructions in E1 are shaded in [Figure 5-7](#). The multiplexers used for the input operands to the functional units are also shaded in the figure. The bold crosspaths are used by the **MPY** instructions.

Most DSP instructions are single-cycle instructions, which means they have only one execution phase (E1). A small number of instructions require more than one execute phase. The types of instructions, each of which require different numbers of execute phases, are described in 5.2 “[Pipeline Execution of Instruction Types](#)”.

**Example 5-1 Execute Packet in [Figure 5-7](#)**

```

SADD.L1 A2,A7,A2; E1 Phase
    ||| SADD.L2 B2,B7,B2
    ||| SMPYH.M2XB3,A3,B2
    ||| SMPYH.M1XB3,A3,A2
    ||| B .S1 LOOP1
    ||| MVK .S2 117,B1
    ||| LDW .D2 *B4++,B3; DC Phase
    ||| LDW .D1 *A4++,A3
    ||| MV .L2XA1,B0
    ||| SMPYH.M1A2,A2,A0
    ||| SMPYH.M2B2,B2,B10
    ||| SHR .S1 A2,16,A5
    ||| SHR .S2 B2,16,B5
LOOP1:
    ||| STH .D1 A5,*A8++[2]; DP, PW, and PG Phases
    ||| STH .D2 B5,*B8++[2]
    ||| SADD.L1 A2,A7,A2
    ||| SADD.L2 B2,B7,B2
    ||| SMPYH.M2XB3,A3,B2
    ||| SMPYH.M1XB3,A3,A2
    ||| [B1] B.S1LOOP1
    ||| [B1] SUB.S2B1,1,B1
    ||| LDW .D2 *B4++,B3: PR and PS Phases
    ||| LDW .D1 *A4++,A3
    ||| SADD.L1 A0,A1,A1
    ||| SADD.L2 B10,B0,B0
    ||| SMPYH.M1A2,A2,A0
    ||| SMPYH.M2B2,B2,B10
    ||| SHR .S1 A2,16,A5
    ||| SHR .S2 B2,16,B5

```

**End of Example 5-1**

## 5.2 Pipeline Execution of Instruction Types

The pipeline operation of the C674x DSP instructions can be categorized into fourteen instruction types. Thirteen of these (**NOP** is not included) are shown in [Table 5-2](#), [Table 5-3](#), [Table 5-4](#), and [Table 5-5](#), which is a mapping of operations occurring in each execution phase for the different instruction types. The delay slots and functional unit latency associated with each instruction type are listed in the bottom row. See section [3.8.13 “Constraints on Floating-Point Instructions”](#) on page 3-23 for any instruction constraints.

The execution of instructions is defined in terms of delay slots. A delay slot is a CPU cycle that occurs after the first execution phase (E1) of an instruction. Results from instructions with delay slots are not available until the end of the last delay slot. For example, a multiply instruction has one delay slot, which means that one CPU cycle elapses before the results of the multiply are available for use by a subsequent instruction. However, results are available from other instructions finishing execution during the same CPU cycle in which the multiply is in a delay slot.

**Table 5-2 Execution Stage Length Description for Each Instruction Type - Part A**

		Instruction Type						
Execution Phase <sup>1,2</sup>	Single Cycle		16 × 16 Single Multiply/.M Unit Nonmultiply		Store	Multiply Extensions	Load	Branch
	E1	E2	E3	E4	E5			
E1	Compute result and write to register	Read operands and start computations	Compute address	Reads operands and start computations	Compute address	Target code in PG <sup>3</sup>		
E2		Compute result and write to register	Send address and data to memory			Send address to memory		
E3			Access memory			Access memory		
E4				Write results to register		Send data back to CPU		
E5						Write data into register		
<b>Delay slots</b>	0	1	0 <sup>4</sup>	3	4 <sup>4</sup>	5 <sup>3</sup>		
<b>Functional unit latency</b>	1	1	1	1	1	1		

1. This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.
2. **NOP** is not shown and has no operation in any of the execution phases.
3. See section [5.2.6](#) for more information on branches.
4. See section [5.2.3](#) and section [5.2.5](#) for more information on execution and delay slots for stores and loads.

## 5.2 Pipeline Execution of Instruction Types

## Chapter 5—Pipeline

**Table 5-3** Execution Stage Length Description for Each Instruction Type - Part B

Execution Phase <sup>1,2</sup>	Instruction Type				
	1-Cycle DP	3-Cycle	4-Cycle	INSTDP	DP Compare
E1	Compute the lower results and write to register	Read sources and start computation	Read sources and start computation	Read sources and start computation	Read lower sources and start computation
E2	Compute the upper results and write to register	Continue computation	Continue computation	Continue computation	Read upper sources, finish computation, and write results to register
E3		Complete computation and write results to register	Continue computation	Continue computation	
E4			Complete computation and write results to register	Continue computation and write lower results to register	
E5				Complete computation and write upper results to register	
<b>Delay slots</b>	1	2	3	4	1
<b>Functional unit latency</b>	1	1	1	1	1

1. This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.
2. **NOP** is not shown and has no operation in any of the execution phases.

**Table 5-4** Execution Stage Length Description for Each Instruction Type - Part C

Execution Phase <sup>1,2</sup>	Instruction Type			
	ADDDP/SUBDP	MPYI	MPYID	MPYDP
E1	Read lower sources and start computation	Read sources and start computation	Read sources and start computation	Read lower sources and start computation
E2	Read upper sources and continue computation	Read sources and continue computation	Read sources and continue computation	Read lower <i>src1</i> and upper <i>src2</i> and continue computation
E3	Continue computation	Read sources and continue computation	Read sources and continue computation	Read lower <i>src2</i> and upper <i>src1</i> and continue computation
E4	Continue computation	Read sources and continue computation	Read sources and continue computation	Read upper sources and continue computation
E5	Continue computation	Continue computation	Continue computation	Continue computation
E6	Compute the lower results and write to register	Continue computation	Continue computation	Continue computation
E7	Compute the upper results and write to register	Continue computation	Continue computation	Continue computation
E8		Continue computation	Continue computation	Continue computation
E9		Complete computation and write results to register	Continue computation and write lower results to register	Continue computation and write lower results to register
E10			Complete computation and write upper results to register	Complete computation and write upper results to register
<b>Delay slots</b>	6	8	9	9
<b>Functional unit latency</b>	2	4	4	4

1. This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.
2. **NOP** is not shown and has no operation in any of the execution phases.

**Table 5-5** Execution Stage Length Description for Each Instruction Type - Part D

Execution Phase <sup>12</sup>	Instruction Type	
	MPYSPDP	MPYSP2DP
E1	Read <i>src1</i> and lower <i>src2</i> and start computation	Read sources and start computation
E2	Read <i>src1</i> and upper <i>src2</i> and continue computation	Continue computation
E3	Continue computation	Continue computation
E4	Continue computation	Continue computation and write lower results to register
E5	Continue computation	Complete computation and write upper results to register
E6	Continue computation and write lower results to register	
E7	Complete computation and write upper results to register	
<b>Delay slots</b>	6	4
<b>Functional unit latency</b>	3	2

1. This table assumes that the condition for each instruction is evaluated as true. If the condition is evaluated as false, the instruction does not write any results or have any pipeline operation after E1.

2. **NOP** is not shown and has no operation in any of the execution phases.

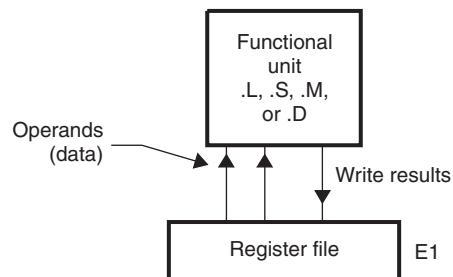
### 5.2.1 Single-Cycle Instructions

Single-cycle instructions complete execution during the E1 phase of the pipeline ([Table 5-6](#)). [Figure 5-8](#) shows the fetch, decode, and execute phases of the pipeline that the single-cycle instructions use.

[Figure 5-9](#) shows the single-cycle execution diagram. The operands are read, the operation is performed, and the results are written to a register, all during E1. Single-cycle instructions have no delay slots.

**Table 5-6** Single-Cycle Instruction Execution

Pipeline Stage	E1
Read	<i>src1, src2</i>
Written	<i>dst</i>
Unit in use	.L, .S, .M, or .D

**Figure 5-8** Single-Cycle Instruction Phases**Figure 5-9** Single-Cycle Instruction Execution Block Diagram

### 5.2.2 Two-Cycle Instructions and .M Unit Nonmultiply Operations

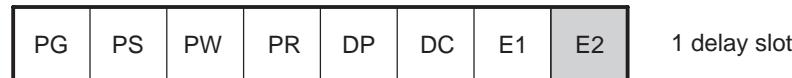
Two-cycle or multiply instructions use both the E1 and E2 phases of the pipeline to complete their operations (Table 5-7). Figure 5-10 shows the fetch, decode, and execute phases of the pipeline that the two-cycle instructions use.

Figure 5-11 shows the operations occurring in the pipeline for a multiply instruction. In the E1 phase, the operands are read and the multiply begins. In the E2 phase, the multiply finishes, and the result is written to the destination register. Multiply instructions have one delay slot. Figure 5-11 also applies to the other .M unit nonmultiply operations.

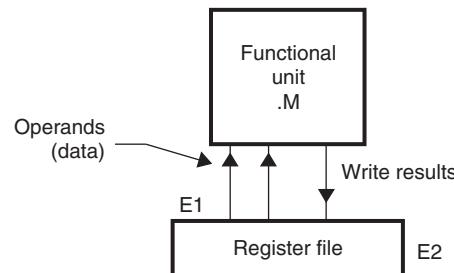
**Table 5-7     Multiply Instruction Execution**

Pipeline Stage	E1	E2
Read	<i>src1, src2</i>	
Written		<i>dst</i>
Unit in use	.M	

**Figure 5-10    Two-Cycle Instruction Phases**



**Figure 5-11    Single 16 × 16 Multiply Instruction Execution Block Diagram**



### 5.2.3 Store Instructions

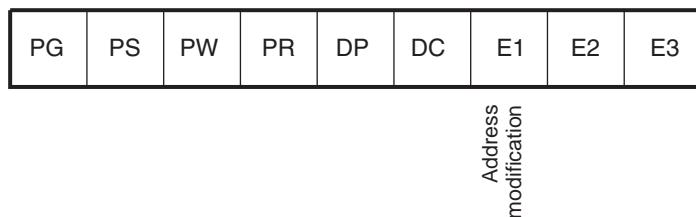
Store instructions require phases E1 through E3 of the pipeline to complete their operations (Table 5-8). Figure 5-12 shows the fetch, decode, and execute phases of the pipeline that the store instructions use.

Figure 5-13 shows the operations occurring in the pipeline phases for a store instruction. In the E1 phase, the address of the data to be stored is computed. In the E2 phase, the data and destination addresses are sent to data memory. In the E3 phase, a memory write is performed. The address modification is performed in the E1 stage of the pipeline. Even though stores finish their execution in the E3 phase of the pipeline, they have no delay slots. There is additional explanation of why stores have zero delay slots in Section 5.2.5 .

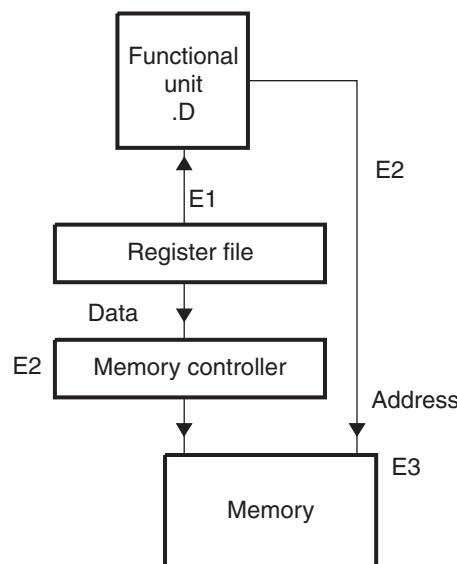
**Table 5-8      Store Instruction Execution**

Pipeline Stage	E1	E2	E3
Read	<i>baseR, offsetR, src</i>		
Written		<i>baseR</i>	
Unit in use		<i>.D2</i>	

**Figure 5-12    Store Instruction Phases**



**Figure 5-13    Store Instruction Execution Block Diagram**



When you perform a load and a store to the same memory location, these rules apply ( $i = \text{cycle}$ ):

- When a load is executed before a store, the old value is loaded and the new value is stored.

$$\begin{array}{cc} i & \text{LDW} \\ i+1 & \text{STW} \end{array}$$

- When a store is executed before a load, the new value is stored and the new value is loaded.

$$\begin{array}{cc} i & \text{STW} \\ i+1 & \text{LDW} \end{array}$$

- When the instructions are executed in parallel, the old value is loaded first and then the new value is stored, but both occur in the same phase.

$$\begin{array}{cc} i & \text{STW} \\ i & \parallel \text{LDW} \end{array}$$

### 5.2.4 Extended Multiply Instructions

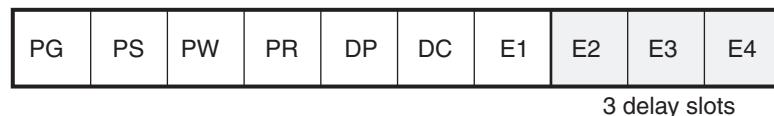
The extended multiply instructions use phases E1 through E4 to complete their operations (Table 5-9). Figure 5-14 shows the fetch, decode, and execute phases of the pipeline that the extended multiply instructions.

Figure 5-15 shows the operations occurring in the pipeline for the multiply extensions. In the E1 phase, the operands are read and the multiplies begin. In the E4 phase, the multiplies finish, and the results are written to the destination register. Extended multiply instructions have three delay slots.

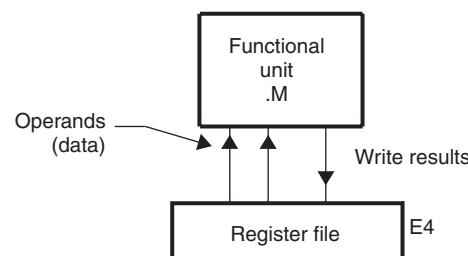
**Table 5-9 Extended Multiply Instruction Execution**

Pipeline Stage	E1	E2	E3	E4
Read	<i>src1, src2</i>			
Written				<i>dst</i>
Unit in use		<i>.M</i>		

**Figure 5-14 Extended Multiply Instruction Phases**



**Figure 5-15 Extended Multiply Instruction Execution Block Diagram**



### 5.2.5 Load Instructions

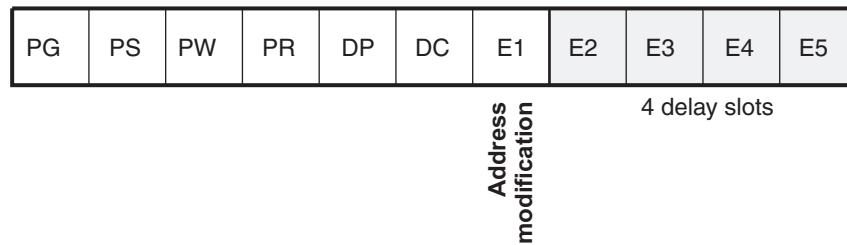
Data loads require all five, E1 through E5, of the pipeline execute phases to complete their operations (Table 5-10). Figure 5-16 shows the fetch, decode, and execute phases of the pipeline that the load instructions use.

Figure 5-17 shows the operations occurring in the pipeline phases for a load. In the E1 phase, the data address pointer is modified in its register. In the E2 phase, the data address is sent to data memory. In the E3 phase, a memory read at that address is performed.

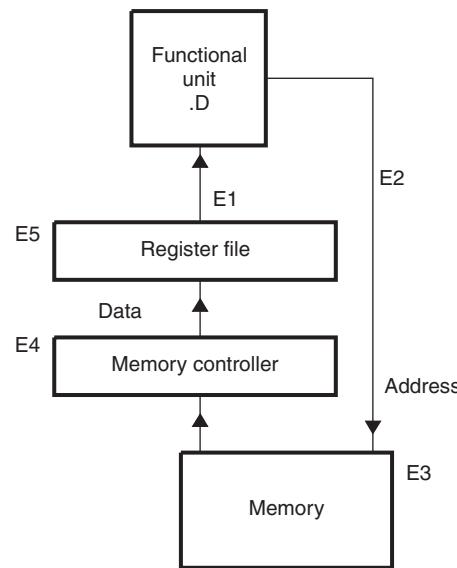
**Table 5-10 Load Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5
Read	<i>baseR, offsetR, src</i>				
Written		<i>baseR</i>			<i>dst</i>
Unit in use			<i>.D</i>		

**Figure 5-16 Load Instruction Phases**



**Figure 5-17 Load Instruction Execution Block Diagram**



In the E4 stage of a load, the data is received at the CPU core boundary. Finally, in the E5 phase, the data is loaded into a register. Because data is not written to the register until E5, load instructions have four delay slots. Because pointer results are written to the register in E1, there are no delay slots associated with the address modification.

In the following code, pointer results are written to the A4 register in the first execute phase of the pipeline and data is written to the A3 register in the fifth execute phase.

```
LDW .D1 *A4++, A3
```

Because a store takes three execute phases to write a value to memory and a load takes three execute phases to read from memory, a load following a store accesses the value placed in memory by that store in the cycle after the store is completed. This is why the store is considered to have zero delay slots.

### 5.2.6 Branch Instructions

Although branch instructions take one execute phase, there are five delay slots between the execution of the branch and execution of the target code ([Table 5-11](#)). [Figure 5-18](#) shows the pipeline phases used by the branch instruction and branch target code. The delay slots are shaded.

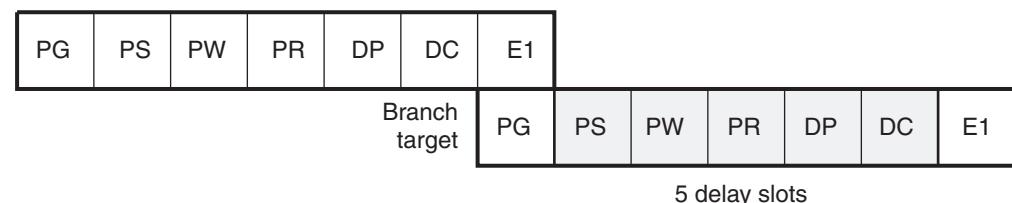
[Figure 5-19](#) shows a branch instruction execution block diagram. If a branch is in the E1 phase of the pipeline (in the .S2 unit in [Figure 5-19](#)), its branch target is in the fetch packet that is in PG during that same cycle (shaded in the figure). Because the branch target has to wait until it reaches the E1 phase to begin execution, the branch takes five delay slots before the branch target code executes.

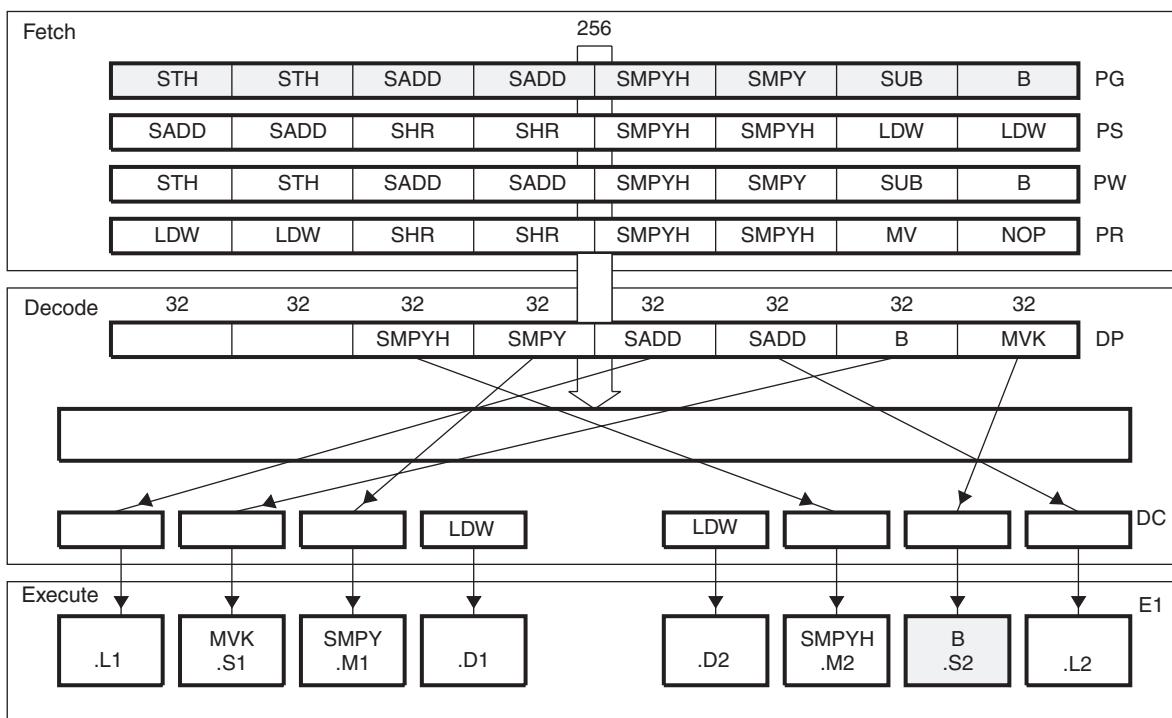
On the DSP, a stall is inserted if a branch is taken to an execute packet that spans fetch packets to give time to fetch the second packet. Normally the assembler compensates for this by preventing branch targets from spanning fetch packets. The one case in which this cannot be done is in the case that an interrupt or exception occurred and the return target is a fetch packet spanning execute packet.

**Table 5-11 Branch Instruction Execution**

Pipeline Stage	Target Instruction						
	E1	PS	PW	PR	DP	DC	E1
Read	<i>src2</i>						
Written							
Branch taken							✓
Unit in use	.S2						

**Figure 5-18 Branch Instruction Phases**



**Figure 5-19 Branch Instruction Execution Block Diagram**

### 5.2.7 Two-Cycle DP Instructions

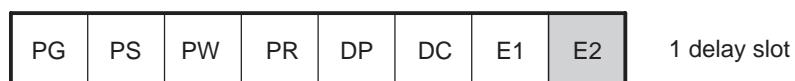
Two-cycle DP instructions use both the E1 and E2 phases of the pipeline to complete their operations (see [Table 5-12](#)). The following instructions are two-cycle DP instructions:

- ABSDP
- RCPDP
- RSQDP
- SPD

The lower and upper 32 bits of the DP source are read on E1 using the *src1* and *src2* ports, respectively. The lower 32 bits of the DP source are written on E1 and the upper 32 bits of the DP source are written on E2. The two-cycle DP instructions are executed on the .S units. The status is written to the FAUCR on E1. [Figure 5-20](#) shows the fetch, decode, and execute phases of the pipeline that the two-cycle DP instructions use.

**Table 5-12 Two-Cycle DP Instruction Execution**

Pipeline Stage	E1	E2
Read	<i>src2_l,</i> <i>src2_h</i>	
Written	<i>dst_l</i>	<i>dst_h</i>
Unit in use	.S	

**Figure 5-20 Two-Cycle DP Instruction Phases**

### 5.2.8 Three-Cycle Instructions

Three-cycle instructions use the E1 through E3 phases of the pipeline to complete their operations (see [Table 5-13](#)). The following instructions are three-cycle instructions:

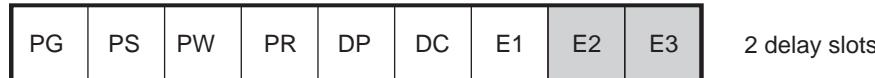
- DADDSP
- DINTHSP
- DINTHSP
- DSPINT
- DSPINTH
- DSUBSP
- FADDDP
- FADDSP
- FSUBDP
- FSUBSP

The sources are read on E1 and the results are written on E3. The three-cycle instructions are executed on the .L and .S units. The status is written to the floating point adder configuration register (FADCR) on E3. [Figure 5-21](#) shows the fetch, decode and execute phases of the pipeline that the three-cycle instructions use.

**Table 5-13     Three-Cycle Instruction Execution**

Pipeline Stage	E1	E2	E3
Read	src1, src2		
Written			dst
Unit in use	.L or .S		

**Figure 5-21     Three-Cycle Instruction Phases**



### 5.2.9 Four-Cycle Instructions

Four-cycle instructions use the E1 through E4 phases of the pipeline to complete their operations (see [Table 5-14](#)). The following instructions are four-cycle instructions:

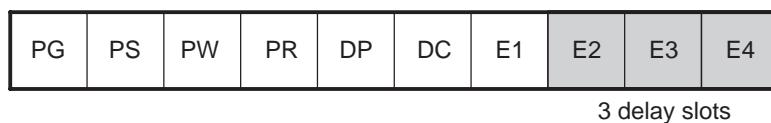
- ADDSP
- DPINT
- DPSP
- DPTRUNC
- INTSP
- MPYSP
- SPINT
- S PTRUNC
- SUBSP

The sources are read on E1 and the results are written on E4. The four-cycle instructions are executed on the .L or .M units. The status is written to the floating-point multiplier configuration register (FMCR) or the floating-point adder configuration register (FADCR) on E4. [Figure 5-22](#) shows the fetch, decode, and execute phases of the pipeline that the four-cycle instructions use.

**Table 5-14 Four-Cycle Instruction Execution**

Pipeline Stage	E1	E2	E3	E4
<b>Read</b>	<i>src1, src2</i>			
<b>Written</b>				<i>dst</i>
<b>Unit in use</b>	.L or .M			

**Figure 5-22 Four-Cycle Instruction Phases**



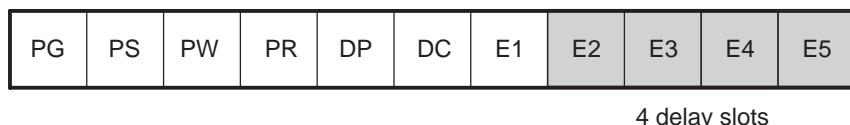
### 5.2.10 INTDP Instruction

The INTDP instruction uses the E1 through E5 phases of the pipeline to complete its operations (see [Table 5-15](#)). *src2* is read on E1, the lower 32 bits of the result are written on E4, and the upper 32 bits of the result are written on E5. The INTDP instruction is executed on the .L unit. The status is written to the floating-point adder configuration register (FADCR) on E4. [Figure 5-23](#) shows the fetch, decode, and execute phases of the pipeline that the INTDP instruction uses.

**Table 5-15 INTDP Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5
<b>Read</b>	<i>src2</i>				
<b>Written</b>				<i>dst_L</i>	<i>dst_h</i>
<b>Unit in use</b>	.L				

**Figure 5-23 INTDP Instruction Phases**



### 5.2.11 Double-Precision (DP) Compare Instructions

The double-precision (DP) compare instructions use the E1 and E2 phases of the pipeline to complete their operations (see [Table 5-16](#)). The lower 32 bits of the sources are read on E1, the upper 32 bits of the sources are read on E2, and the results are written on E2. The following instructions are DP compare instructions:

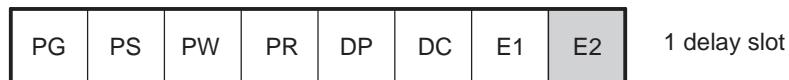
- CMPEQDP
- CMPLTDP
- CMPGTDP

The DP compare instructions are executed on the .S unit. The functional unit latency for DP compare instructions is 2. The status is written to the floating-point auxiliary register (FAUCR) on E2. [Figure 5-24](#) shows the fetch, decode, and execute phases of the pipeline that the DP compare instruction uses.

**Table 5-16 DP Compare Instruction Execution**

Pipeline Stage	E1	E2
<b>Read</b>	<i>src1_l, src2_l</i>	<i>src1_h, src2_h</i>
<b>Written</b>		<i>dst</i>
<b>Unit in use</b>	.S	.S

**Figure 5-24 DP Compare Instruction Phases**



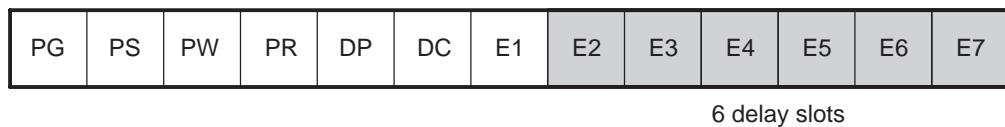
### 5.2.12 ADDDP/SUBDP Instructions

The ADDDP/SUBDP instructions use the E1 through E7 phases of the pipeline to complete their operations (see [Table 5-17](#)). The lower 32 bits of the result are written on E6, and the upper 32 bits of the result are written on E7. The ADDDP/SUBDP instructions are executed on the .L or .S units. The functional unit latency for ADDDP/SUBDP instructions is 2. The status is written to the floating-point adder configuration register (FADCR) on E6. [Figure 5-25](#) shows the fetch, decode, and execute phases of the pipeline that the ADDDP/SUBDP instructions use.

**Table 5-17 ADDDP/SUBDP Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
<b>Read</b>	<i>src1_l,</i> <i>src2_l</i>	<i>src1_h,</i> <i>src2_h</i>					
<b>Written</b>						<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.L or .S	.L or .S					

**Figure 5-25 ADDDP/SUBDP Instruction Phases**



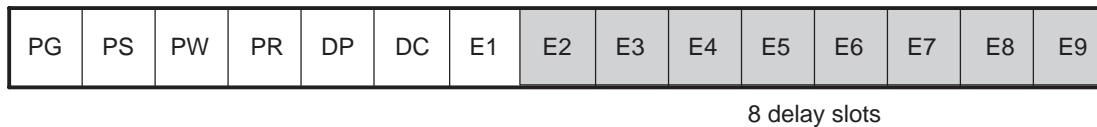
### 5.2.13 MPYI Instruction

The **MPYI** instruction uses the E1 through E9 phases of the pipeline to complete its operations (see [Table 5-18](#)). The sources are read on cycles E1 through E4 and the result is written on E9. The **MPYI** instruction is executed on the .M unit. The functional unit latency for the **MPYI** instruction is 4. [Figure 5-26](#) shows the fetch, decode, and execute phases of the pipeline that the **MPYI** instruction uses.

**Table 5-18 MPYI Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9
<b>Read</b>	<i>src1, src2</i>	<i>src1, src2</i>	<i>src1, src2</i>	<i>src1, src2</i>					
<b>Written</b>									<i>dst</i>
<b>Unit in use</b>	.M	.M	.M	.M					

**Figure 5-26 MPYI Instruction Phases**



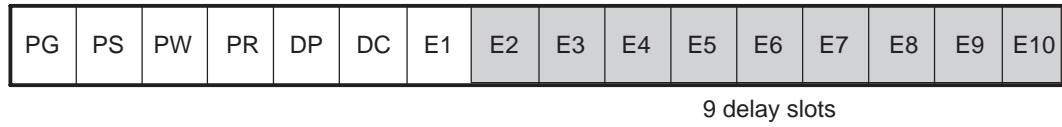
### 5.2.14 MPYID Instruction

The **MPYID** instruction uses the E1 through E10 phases of the pipeline to complete its operations (see [Table 5-19](#)). The sources are read on cycles E1 through E4, the lower 32 bits of the result are written on E9, and the upper 32 bits of the result are written on E10. The **MPYID** instruction is executed on the .M unit. The functional unit latency for the **MPYID** instruction is 4. [Figure 5-27](#) shows the fetch, decode, and execute phases of the pipeline that the **MPYID** instruction uses.

**Table 5-19 MPYID Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
<b>Read</b>	<i>src1, src2</i>	<i>src1, src2</i>	<i>src1, src2</i>	<i>src1, src2</i>						
<b>Written</b>									<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M	.M	.M	.M						

**Figure 5-27 MPYID Instruction Phases**



### 5.2.15 MPYDP Instruction

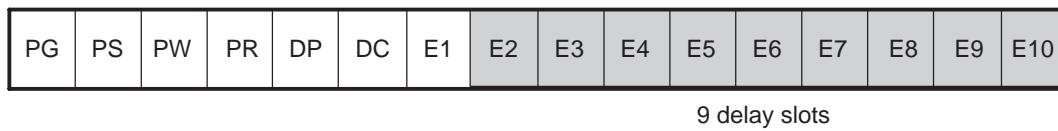
The **MPYDP** instruction uses the E1 through E10 phases of the pipeline to complete its operations (see [Table 5-20](#)). The lower 32 bits of *src1* are read on E1 and E2, and the upper 32 bits of *src1* are read on E3 and E4. The lower 32 bits of *src2* are read on E1 and E3, and the upper 32 bits of *src2* are read on E2 and E4. The lower 32 bits of the result are written on E9, and the upper 32 bits of the result are written on E10. The **MPYDP**

instruction is executed on the .M unit. The functional unit latency for the MPYDP instruction is 4. The status is written to the floating-point multiplier configuration register (FMCR) on E9. [Figure 5-28](#) shows the fetch, decode, and execute phases of the pipeline that the MPYDP instruction uses.

**Table 5-20 MPYDP Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
<b>Read</b>	<i>src1_l</i> , <i>src2_l</i>	<i>src1_l</i> , <i>src2_h</i>	<i>src1_h</i> , <i>src2_l</i>	<i>src1_h</i> , <i>src2_h</i>						
<b>Written</b>									<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M	.M	.M	.M						

**Figure 5-28 MPYDP Instruction Phases**



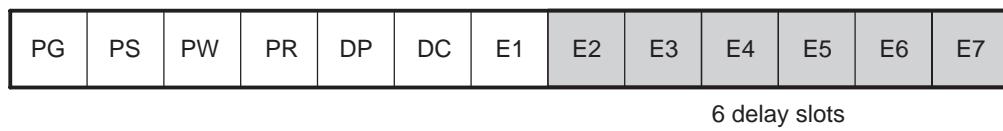
### 5.2.16 MPYSPDP Instruction

The MPYSPDP instruction uses the E1 through E7 phases of the pipeline to complete its operations (see [Table 5-21](#)). *src1* is read on E1 and E2. The lower 32 bits of *src2* are read on E1, and the upper 32 bits of *src2* are read on E2. The lower 32 bits of the result are written on E6, and the upper 32 bits of the result are written on E7. The MPYSPDP instruction is executed on the .M unit. The functional unit latency for the MPYSPDP instruction is 3. [Figure 5-29](#) shows the fetch, decode, and execute phases of the pipeline that the MPYSPDP instruction uses.

**Table 5-21 MPYSPDP Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5	E6	E7
<b>Read</b>	<i>src1, src2_l</i>	<i>src1, src2_h</i>					
<b>Written</b>						<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M	.M					

**Figure 5-29 MPYSPDP Instruction Phases**



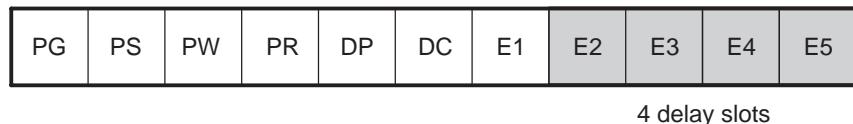
### 5.2.17 MPYSP2DP Instruction

The **MPYSP2DP** instruction uses the E1 through E5 phases of the pipeline to complete its operations (see [Table 5-22](#)). *src1* and *src2* are read on E1. The lower 32 bits of the result are written on E4, and the upper 32 bits of the result are written on E5. The **MPYSP2DP** instruction is executed on the .M unit. The functional unit latency for the **MPYSP2DP** instruction is 2. [Figure 5-30](#) shows the fetch, decode, and execute phases of the pipeline that the **MPYSP2DP** instruction uses.

**Table 5-22 MPYSP2DP Instruction Execution**

Pipeline Stage	E1	E2	E3	E4	E5
<b>Read</b>	<i>src1, src2</i>				
<b>Written</b>				<i>dst_l</i>	<i>dst_h</i>
<b>Unit in use</b>	.M				

**Figure 5-30 MPYSP2DP Instruction Phases**



## 5.3 Functional Unit Constraints

If you want to optimize the instruction pipeline, consider the instructions that are executed on each unit. Sources and destinations are read and written differently for each instruction. If you analyze these differences, you can make further optimization improvements by considering what happens during the execution phases of instructions that use the same functional unit in each execution packet.

The following sections provide information about what happens during each execute phase of the instructions within a category for each of the functional units.

### 5.3.1 .S-Unit Constraints

**Table 5-23** shows the instruction constraints for single-cycle instructions executing on the .S unit.

**Table 5-23 Single-Cycle .S-Unit Instruction Constraints**

Instruction Execution		
Cycle	1	2
Single-cycle	RW	
Instruction Type	Subsequent Same-Unit Instruction Executable	
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
Branch		✓
Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable	
Single-cycle		✓
Load		✓
Store		✓
INTDP		✓
ADDDP/SUBDP		✓
16 × 16 multiply		✓
4-cycle		✓
MPYI		✓
MPYID		✓
MPYDP		✓

**LEGEND:** Shaded column = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction; ✓ = Next instruction can enter E1 during cycle

**Table 5-24** shows the instruction constraints for DP compare instructions executing on the .S unit.

**Table 5-24 DP Compare .S-Unit Instruction Constraints**

		Instruction Execution		
Cycle		1	2	3
DP compare		R	RW	
Instruction Type		Subsequent Same-Unit Instruction Executable		
Single-cycle		Xrw ✓		
DP compare		Xr ✓		
2-cycle DP		Xrw ✓		
ADDDP/SUBDP		Xr ✓		
ADDSP/SUBSP		Xr ✓		
Branch <sup>1</sup>		Xr ✓		
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable		
Single-cycle		Xr ✓		
Load		Xr ✓		
Store		Xr ✓		
INTDP		Xr ✓		
ADDDP/SUBDP		Xr ✓		
16 × 16 multiply		Xr ✓		
4-cycle		Xr ✓		
MPYI		Xr ✓		
MPYID		Xr ✓		
MPYDP		Xr ✓		

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xrw = Next instruction cannot enter E1 during cycle-read/decode/write constraint

1. The branch on register instruction is the only branch instruction that reads a general-purpose register

**Table 5-25** shows the instruction constraints for 2-cycle DP instructions executing on the .S unit.

**Table 5-25 2-Cycle DP .S-Unit Instruction Constraints**

Instruction Execution			
Cycle	1	2	3
2-cycle	RW	W	
Subsequent Same-Unit Instruction Executable			
Single-cycle	Xw	✓	
DP compare	✓	✓	
2-cycle DP	Xw	✓	
Branch	✓	✓	
Same Side, Different Unit, Both Using Cross Path Executable			
Single cycle	✓	✓	
Load	✓	✓	
Store	✓	✓	
INTDP	✓	✓	
ADDDP/SUBDP	✓	✓	
16 × 16 multiply	✓	✓	
4-cycle	✓	✓	
MPYI	✓	✓	
MPYID	✓	✓	
MPYDP	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xw = Next instruction cannot enter E1 during cycle-write constraint

**Table 5-26** shows the instruction constraints for ADDSP/SUBSP instructions executing on the .S unit.

**Table 5-26 ADDSP/SUBSP .S-Unit Instruction Constraints**

Instruction Execution				
Cycle	1	2	3	4
ADDSP/SUBSP	R		W	
Subsequent Same-Unit Instruction Executable				
Single-cycle	✓	✓	Xw	
2-cycle DP	✓	Xw	Xw	
DP compare	✓	Xw	✓	
ADDDP/SUBDP	✓	✓	✓	
ADDSP/SUBSP	✓	✓	✓	
Branch	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xw = Next instruction cannot enter E1 during cycle-write constraint

**Table 5-27** shows the instruction constraints for ADDDP/SUBDP instructions executing on the .S unit.

**Table 5-27 ADDDP/SUBDP .S-Unit Instruction Constraints**

		Instruction Execution						
Cycle		1	2	3	4	5	6	7
ADDDP/SUBDP		R	R			W	W	
Instruction Type		Subsequent Same-Unit Instruction Executable						
Single-cycle		Xr	✓	✓	✓	Xw	Xw	
2-cycle DP		Xr	✓	✓	Xw	Xw	Xw	
DP compare		Xr	✓	✓	Xw	Xw	✓	
ADDDP/SUBDP		Xr	✓	✓	✓	✓	✓	
ADDSP/SUBSP		Xr	Xw	Xw	✓	✓	✓	
Branch		Xr	✓	✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint

**Table 5-28** shows the instruction constraints for branch instructions executing on the .S unit.

**Table 5-28 Branch .S-Unit Instruction Constraints**

		Instruction Execution						
Cycle		1	2	3	4	5	6	7
Branch <sup>1</sup>		R						
Instruction Type		Subsequent Same-Unit Instruction Executable						
Single-cycle		✓	✓	✓	✓	✓	✓	✓
DP compare		✓	✓	✓	✓	✓	✓	✓
2-cycle DP		✓	✓	✓	✓	✓	✓	✓
Branch		✓	✓	✓	✓	✓	✓	✓
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable						
Single-cycle		✓	✓	✓	✓	✓	✓	✓
Load		✓	✓	✓	✓	✓	✓	✓
Store		✓	✓	✓	✓	✓	✓	✓
INTDP		✓	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP		✓	✓	✓	✓	✓	✓	✓
16 × 16 multiply		✓	✓	✓	✓	✓	✓	✓
4-cycle		✓	✓	✓	✓	✓	✓	✓
MPYI		✓	✓	✓	✓	✓	✓	✓
MPYID		✓	✓	✓	✓	✓	✓	✓
MPYDP		✓	✓	✓	✓	✓	✓	✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; ✓ = Next instruction can enter E1 during cycle

1. The branch on register instruction is the only branch instruction that reads a general-purpose register

### 5.3.2 .M-Unit Constraints

**Table 5-29** shows the instruction constraints for  $16 \times 16$  multiply instructions executing on the .M unit.

**Table 5-29     $16 \times 16$  Multiply .M-Unit Instruction Constraints**

Instruction Execution			
Cycle	1	2	3
$16 \times 16$ multiply	R	W	
Instruction Type	Subsequent Same-Unit Instruction Executable		
16 × 16 multiply		✓	✓
4-cycle		✓	✓
MPYI		✓	✓
MPYID		✓	✓
MPYDP		✓	✓
Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable		
Single-cycle		✓	✓
Load		✓	✓
Store		✓	✓
DP compare		✓	✓
2-cycle DP		✓	✓
Branch		✓	✓
4-cycle		✓	✓
INTDP		✓	✓
ADDDP/SUBDP		✓	✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle

**Table 5-30** shows the instruction constraints for 4-cycle instructions executing on the .M unit.

**Table 5-30 4-Cycle .M-Unit Instruction Constraints**

		Instruction Execution				
Cycle		1	2	3	4	5
4-cycle		R			W	
<b>Instruction Type</b>		<b>Subsequent Same-Unit Instruction Executable</b>				
16 × 16 multiply		✓	Xw	✓	✓	
4-cycle		✓	✓	✓	✓	
MPYI		✓	✓	✓	✓	
MPYID		✓	✓	✓	✓	
MPYDP		✓	✓	✓	✓	
<b>Instruction Type</b>		<b>Same Side, Different Unit, Both Using Cross Path Executable</b>				
Single-cycle		✓	✓	✓	✓	
Load		✓	✓	✓	✓	
Store		✓	✓	✓	✓	
DP compare		✓	✓	✓	✓	
2-cycle DP		✓	✓	✓	✓	
Branch		✓	✓	✓	✓	
4-cycle		✓	✓	✓	✓	
INTDP		✓	✓	✓	✓	
ADDDP/SUBDP		✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xw = Next instruction cannot enter E1 during cycle-write constraint

**Table 5-31** shows the instruction constraints for MPYI instructions executing on the .M unit.

**Table 5-31 MPYI .M-Unit Instruction Constraints**

		Instruction Execution									
Cycle		1	2	3	4	5	6	7	8	9	10
MPYI		R	R	R	R						W
Instruction Type		Subsequent Same-Unit Instruction Executable									
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	✓	✓	
4-cycle		Xr	Xr	Xr	Xu	Xw	Xu	✓	✓	✓	
MPYI		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
MPYID		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
MPYDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	
MPYSPDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	
MPYSP2DP		Xr	Xr	Xr	Xw	Xw	Xu	✓	✓	✓	
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable									
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
Load		✓	✓	✓	✓	✓	✓	✓	✓	✓	
Store		✓	✓	✓	✓	✓	✓	✓	✓	✓	
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint; Xu = Next instruction cannot enter E1 during cycle-other resource conflict

**Table 5-32** shows the instruction constraints for MPYID instructions executing on the .M unit.

**Table 5-32 MPYID .M-Unit Instruction Constraints**

		Instruction Execution										
Cycle		1	2	3	4	5	6	7	8	9	10	11
MPYID		R	R	R	R					W	W	
Instruction Type		Subsequent Same-Unit Instruction Executable										
16 × 16 multiply		Xr	Xr	Xr	✓	✓	✓	Xw	Xw	✓	✓	
4-cycle		Xr	Xr	Xr	Xu	Xw	Xw	✓	✓	✓	✓	
MPYI		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
MPYID		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
MPYDP		Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓	
MPYSPDP		Xr	Xr	Xr	Xw	Xu	Xu	✓	✓	✓	✓	
MPYSP2DP		Xr	Xr	Xr	Xw	Xw	Xw	✓	✓	✓	✓	
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable										
Single-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
Load		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Store		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
DP compare		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
2-cycle DP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
Branch		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
4-cycle		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
INTDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	
ADDDP/SUBDP		Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint; Xu = Next instruction cannot enter E1 during cycle-other resource conflict

**Table 5-33** shows the instruction constraints for MPYDP instructions executing on the .M unit.

**Table 5-33 MPYDP .M-Unit Instruction Constraints**

		Instruction Execution										
Cycle		1	2	3	4	5	6	7	8	9	10	11
MPYDP	R	R	R	R						W	W	
Instruction Type		Subsequent Same-Unit Instruction Executable										
16 × 16 multiply	Xr	Xr	Xr	✓	✓	✓	Xw	Xw	✓	✓		
4-cycle	Xr	Xr	Xr	Xu	Xw	Xw	✓	✓	✓	✓		
MPYI	Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓		
MPYID	Xr	Xr	Xr	Xu	Xu	Xu	✓	✓	✓	✓		
MPYDP	Xr	Xr	Xr	n	n	n	✓	✓	✓	✓		
MPYSPDP	Xr	Xr	Xr	Xw	Xu	Xu	✓	✓	✓	✓		
MPYSP2DP	Xr	Xr	Xr	Xw	Xw	Xw	✓	✓	✓	✓		
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable										
Single-cycle	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		
Load	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Store	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
DP compare	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		
2-cycle DP	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		
Branch	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		
4-cycle	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		
INTDP	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		
ADDDP/SUBDP	Xr	Xr	Xr	✓	✓	✓	✓	✓	✓	✓		

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint; Xu = Next instruction cannot enter E1 during cycle-other resource conflict

**Table 5-34** shows the instruction constraints for MPYSP instructions executing on the .M unit.

**Table 5-34 MPYSP .M-Unit Instruction Constraints**

		Instruction Execution			
Cycle		1	2	3	4
MPYSP		R		W	
Instruction Type	Subsequent Same-Unit Instruction Executable				
MPYSPDP					✓ ✓ ✓
MPYSP2DP					✓ ✓ ✓
Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable				
Single-cycle					✓ ✓ ✓
Load					✓ ✓ ✓
Store					✓ ✓ ✓
DP compare					✓ ✓ ✓
2-cycle DP					✓ ✓ ✓
Branch					✓ ✓ ✓
4-cycle					✓ ✓ ✓
INTDP					✓ ✓ ✓
ADDDP/SUBDP					✓ ✓ ✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle

**Table 5-35** shows the instruction constraints for MPYSPDP instructions executing on the .M unit.

**Table 5-35 MPYSPDP .M-Unit Instruction Constraints**

		Instruction Execution						
Cycle		1	2	3	4	5	6	7
MPYSPDP	R	R				W	W	
Instruction Type		Subsequent Same-Unit Instruction Executable						
16 × 16 multiply	Xr	✓	✓	Xw	Xw	✓		
MPYDP	Xr	Xu	Xu	✓	✓	✓	✓	
MPYI	Xr	Xu	Xu	✓	✓	✓	✓	
MPYID	Xr	Xu	Xu	✓	✓	✓	✓	
MPYSP	Xr	Xw	Xw	✓	✓	✓	✓	
MPYSPDP	Xr	Xu	✓	✓	✓	✓	✓	
MPYSP2DP	Xr	Xw	Xw	✓	✓	✓	✓	
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable						
Single-cycle	Xr	✓	✓	✓	✓	✓	✓	
Load	Xr	✓	✓	✓	✓	✓	✓	
Store	Xr	✓	✓	✓	✓	✓	✓	
DP compare	Xr	✓	✓	✓	✓	✓	✓	
2-cycle DP	Xr	✓	✓	✓	✓	✓	✓	
Branch	Xr	✓	✓	✓	✓	✓	✓	
4-cycle	Xr	✓	✓	✓	✓	✓	✓	
INTDP	Xr	✓	✓	✓	✓	✓	✓	
ADDDP/SUBDP	Xr	✓	✓	✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint; Xu = Next instruction cannot enter E1 during cycle-other resource conflict

**Table 5-36** shows the instruction constraints for **MPYSP2DP** instructions executing on the .M unit.

**Table 5-36 MPYSP2DP .M-Unit Instruction Constraints**

		Instruction Execution				
Cycle		1	2	3	4	5
MPYSP2DP		R	R	W	W	
Instruction Type		Subsequent Same-Unit Instruction Executable				
16 × 16 multiply		✓	Xw	Xw	✓	
MPYDP		Xu	✓	✓	✓	
MPYI		Xu	✓	✓	✓	
MPYID		Xu	✓	✓	✓	
MPYSP		Xw	✓	✓	✓	
MPYSPDP		Xu	✓	✓	✓	
MPYSP2DP		Xw	✓	✓	✓	
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable				
Single-cycle		Xr	✓	✓	✓	
Load		Xr	✓	✓	✓	
Store		Xr	✓	✓	✓	
DP compare		Xr	✓	✓	✓	
2-cycle DP		Xr	✓	✓	✓	
Branch		Xr	✓	✓	✓	
4-cycle		Xr	✓	✓	✓	
INTDP		Xr	✓	✓	✓	
ADDDP/SUBDP		Xr	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint; Xu = Next instruction cannot enter E1 during cycle-other resource conflict

### 5.3.3 L-Unit Constraints

**Table 5-37** shows the instruction constraints for single-cycle instructions executing on the .L unit.

**Table 5-37 Single-Cycle .L-Unit Instruction Constraints**

Instruction Execution		
Cycle	1	2
Single-cycle	RW	
Instruction Type	Subsequent Same-Unit Instruction Executable	
Single-cycle		✓
4-cycle		✓
INTDP		✓
ADDDP/SUBDP		✓
Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable	
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
4-cycle		✓
Load		✓
Store		✓
Branch		✓
16 × 16 multiply		✓
MPYI		✓
MPYID		✓
MPYDP		✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle

**Table 5-38** shows the instruction constraints for 4-cycle instructions executing on the .L unit.

**Table 5-38 4-Cycle .L-Unit Instruction Constraints**

		Instruction Execution				
Cycle		1	2	3	4	5
4-cycle		R			W	
<b>Instruction Type</b>		<b>Subsequent Same-Unit Instruction Executable</b>				
Single-cycle		✓	✓	Xw	✓	
4-cycle		✓	✓	✓	✓	
INTDP		✓	✓	✓	✓	
ADDDP/SUBDP		✓	✓	✓	✓	
<b>Instruction Type</b>		<b>Same Side, Different Unit, Both Using Cross Path Executable</b>				
Single-cycle		✓	✓	✓	✓	
DP compare		✓	✓	✓	✓	
2-cycle DP		✓	✓	✓	✓	
4-cycle		✓	✓	✓	✓	
Load		✓	✓	✓	✓	
Store		✓	✓	✓	✓	
Branch		✓	✓	✓	✓	
16 × 16 multiply		✓	✓	✓	✓	
MPYI		✓	✓	✓	✓	
MPYID		✓	✓	✓	✓	
MPYDP		✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle; Xw = Next instruction cannot enter E1 during cycle-write constraint

**Table 5-39** shows the instruction constraints for INTDP instructions executing on the .L unit.

**Table 5-39    INTDP .L-Unit Instruction Constraints**

		Instruction Execution					
Cycle		1	2	3	4	5	6
INTDP	R			W	W		
Instruction Type		Subsequent Same-Unit Instruction Executable					
Single-cycle		✓	✓	Xw	Xw	✓	
4-cycle		Xw	✓	✓	✓	✓	
INTDP		Xw	✓	✓	✓	✓	
ADDDP/SUBDP		✓	✓	✓	✓	✓	
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable					
Single-cycle		✓	✓	✓	✓	✓	
DP compare		✓	✓	✓	✓	✓	
2-cycle DP		✓	✓	✓	✓	✓	
4-cycle		✓	✓	✓	✓	✓	
Load		✓	✓	✓	✓	✓	
Store		✓	✓	✓	✓	✓	
Branch		✓	✓	✓	✓	✓	
16 × 16 multiply		✓	✓	✓	✓	✓	
MPYI		✓	✓	✓	✓	✓	
MPYID		✓	✓	✓	✓	✓	
MPYDP		✓	✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle; Xw = Next instruction cannot enter E1 during cycle-write constraint

**Table 5-40** shows the instruction constraints for ADDDP/SUBDP instructions executing on the .L unit.

**Table 5-40 ADDDP/SUBDP .L-Unit Instruction Constraints**

		Instruction Execution							
Cycle		1	2	3	4	5	6	7	8
ADDDP/SUBDP		R	R				W	W	
Instruction Type	Subsequent Same-Unit Instruction Executable								
Single-cycle	Xr	✓	✓	✓	✓		Xw	Xw	✓
4-cycle	Xr	Xw	Xw	✓	✓	✓	✓	✓	✓
INTDP	Xrw	Xw	Xw	✓	✓	✓	✓	✓	✓
ADDDP/SUBDP	Xr	✓	✓	✓	✓	✓	✓	✓	✓
Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable								
Single-cycle	Xr	✓	✓	✓	✓	✓	✓	✓	✓
DP compare	Xr	✓	✓	✓	✓	✓	✓	✓	✓
2-cycle DP	Xr	✓	✓	✓	✓	✓	✓	✓	✓
4-cycle	Xr	✓	✓	✓	✓	✓	✓	✓	✓
Load	✓	✓	✓	✓	✓	✓	✓	✓	✓
Store	✓	✓	✓	✓	✓	✓	✓	✓	✓
Branch	Xr	✓	✓	✓	✓	✓	✓	✓	✓
16 × 16 multiply	Xr	✓	✓	✓	✓	✓	✓	✓	✓
MPYI	Xr	✓	✓	✓	✓	✓	✓	✓	✓
MPYID	Xr	✓	✓	✓	✓	✓	✓	✓	✓
MPYDP	Xr	✓	✓	✓	✓	✓	✓	✓	✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;

✓ = Next instruction can enter E1 during cycle; Xr = Next instruction cannot enter E1 during cycle-read/decode constraint; Xw = Next instruction cannot enter E1 during cycle-write constraint; Xrw = Next instruction cannot enter E1 during cycle-read/decode/write constraint

### 5.3.4 D-Unit Instruction Constraints

Table 5-41 shows the instruction constraints for load instructions executing on the .D unit.

**Table 5-41 Load .D-Unit Instruction Constraints**

		Instruction Execution						
		1	2	3	4	5	6	
Instruction Type	Cycle							
	Load	RW						
	Store	W						
Instruction Type		Subsequent Same-Unit Instruction Executable						
Single-cycle		✓	✓	✓	✓	✓	✓	
Load			✓	✓	✓	✓	✓	
Store			✓	✓	✓	✓	✓	
Instruction Type		Same Side, Different Unit, Both Using Cross Path Executable						
16 × 16 multiply		✓	✓	✓	✓	✓	✓	
MPYI			✓	✓	✓	✓	✓	
MPYID			✓	✓	✓	✓	✓	
MPYDP			✓	✓	✓	✓	✓	
Single-cycle			✓	✓	✓	✓	✓	
DP compare			✓	✓	✓	✓	✓	
2-cycle DP			✓	✓	✓	✓	✓	
Branch			✓	✓	✓	✓	✓	
4-cycle			✓	✓	✓	✓	✓	
INTDP			✓	✓	✓	✓	✓	
ADDDP/SUBDP			✓	✓	✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle

**Table 5-42** shows the instruction constraints for store instructions executing on the .D unit.

**Table 5-42 Store .D-Unit Instruction Constraints**

		Instruction Execution				
Cycle		1	2	3	4	
Store		RW				
<b>Instruction Type</b>		<b>Subsequent Same-Unit Instruction Executable</b>				
Single-cycle			✓	✓	✓	
Load			✓	✓	✓	
Store			✓	✓	✓	
<b>Instruction Type</b>		<b>Same Side, Different Unit, Both Using Cross Path Executable</b>				
16 × 16 multiply			✓	✓	✓	
MPYI			✓	✓	✓	
MPYID			✓	✓	✓	
MPYDP			✓	✓	✓	
Single-cycle			✓	✓	✓	
DP compare			✓	✓	✓	
2-cycle DP			✓	✓	✓	
Branch			✓	✓	✓	
4-cycle			✓	✓	✓	
INTDP			✓	✓	✓	
ADDDP/SUBDP			✓	✓	✓	

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle

**Table 5-43** shows the instruction constraints for single-cycle instructions executing on the .D unit.

**Table 5-43 Single-Cycle .D-Unit Instruction Constraints**

Instruction Execution		
Cycle	1	2
Single-cycle	RW	
Instruction Type	Subsequent Same-Unit Instruction Executable	
Single-cycle		✓
Load		✓
Store		✓
Instruction Type	Same Side, Different Unit, Both Using Cross Path Executable	
16 × 16 multiply		✓
MPYI		✓
MPYID		✓
MPYDP		✓
Single-cycle		✓
DP compare		✓
2-cycle DP		✓
Branch		✓
4-cycle		✓
INTDP		✓
ADDDP/SUBDP		✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle

**Table 5-44** shows the instruction constraints for LDDW instructions executing on the .D unit.

**Table 5-44 LDDW Instruction With Long Write Instruction Constraints**

Instruction Execution						
Cycle	1	2	3	4	5	6
LDDW	RW				W	
Instruction Type	Subsequent Same-Unit Instruction Executable					
Instruction with long result		✓	✓	✓	Xw	✓

**LEGEND:** Shaded text = E1 phase of the single-cycle instruction; R = Sources read for the instruction; W = Destinations written for the instruction;  
 ✓ = Next instruction can enter E1 during cycle; Xw = Next instruction cannot enter E1 during cycle-write constraint

## 5.4 Performance Considerations

The DSP pipeline is most effective when it is kept as full as the algorithms in the program allow it to be. It is useful to consider some situations that can affect pipeline performance.

A fetch packet (FP) is a grouping of eight instructions. Each FP can be split into from one to eight execute packets (EPs). Each EP contains instructions that execute in parallel. Each instruction executes in an independent functional unit. The effect on the pipeline of combinations of EPs that include varying numbers of parallel instructions, or just a single instruction that executes serially with other code, is considered here.

In general, the number of execute packets in a single FP defines the flow of instructions through the pipeline. Another defining factor is the instruction types in the EP. Each type of instruction has a fixed number of execute cycles that determines when this instruction's operations are complete. Section 5.4.2 “[Multicycle NOPs](#)” on page 5-44 covers the effect of including a multicycle NOP in an individual EP.

Finally, the effect of the memory system on the operation of the pipeline is considered. The access of program and data memory is discussed, along with memory stalls.

### 5.4.1 Pipeline Operation With Multiple Execute Packets in a Fetch Packet

Referring to [Figure 5-6](#) on page 5-5, pipeline operation is shown with eight instructions in every fetch packet. [Figure 5-31](#), however, shows the pipeline operation with a fetch packet that contains multiple execute packets. Code for [Figure 5-31](#) might have this layout:

```

    instruction A ;EP kFPn
    ||      instruction B ;
    instruction C ;EP k + 1FPn
    ||      instruction D
    ||      instruction E
    instruction F ;EP k + 2FPn
    ||      instruction G
    ||      instruction H
    instruction I ;EP k + 3FPn + 1
    ||      instruction J
    ||      instruction K
    ||      instruction L
    ||      instruction M
    ||      instruction N
    ||      instruction O
    ||      instruction P
... continuing with EPs k+4 through k+8, which have eight instructions in parallel,
like k+3.

```

**Figure 5-31 Pipeline Operation: Fetch Packets With Different Numbers of Execute Packets**

Fetch Packet (FP)	Execute Packet (EP)	Clock Cycle													
		1	2	3	4	5	6	7	8	9	10	11	12	13	
n	k	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5			
n	k+1						DP	DC	E1	E2	E3	E4	E5		
n	k+2						DP	DC	E1	E2	E3	E4	E5		
n+1	k+3					PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
n+2	k+4					PG	PS	PW	Pipeline	PR	DP	DC	E1	E2	E3
n+3	k+5					PG	PS	stall	PW	PR	DP	DC	E1	E2	
n+4	k+6					PG		PS	PW	PR	DP	DC	E1		
n+5	k+7						PG	PS	PW	PR	DP	DC			
n+6	k+8							PG	PS	PW	PR	DP			

In Figure 5-31, fetch packet n, which contains three execute packets, is shown followed by six fetch packets (n + 1 through n + 6), each with one execute packet (containing eight parallel instructions). The first fetch packet (n) goes through the program fetch phases during cycles 1–4. During these cycles, a program fetch phase is started for each of the fetch packets that follow.

In cycle 5, the program dispatch (DP) phase, the CPU scans the p-bits and detects that there are three execute packets (k through k + 2) in fetch packet n. This forces the pipeline to stall, which allows the DP phase to start for execute packets k + 1 and k + 2 in cycles 6 and 7. Once execute packet k + 2 is ready to move on to the DC phase (cycle 8), the pipeline stall is released.

The fetch packets n + 1 through n + 4 were all stalled so the CPU could have time to perform the DP phase for each of the three execute packets (k through k + 2) in fetch packet n. Fetch packet n + 5 was also stalled in cycles 6 and 7: it was not allowed to enter the PG phase until after the pipeline stall was released in cycle 8. The pipeline continues operation as shown with fetch packets n + 5 and n + 6 until another fetch packet containing multiple execution packets enters the DP phase, or an interrupt occurs.

### 5.4.2 Multicycle NOPs

The **NOP** instruction has an optional operand, *count*, that allows you to issue a single instruction for multicycle NOPs. A **NOP 2**, for example, fills in extra delay slots for the instructions in its execute packet and for all previous execute packets. If a **NOP 2** is in parallel with an **MPY** instruction, the **MPY** result is available for use by instructions in the next execute packet.

Figure 5-32 shows how a multicycle NOP drives the execution of other instructions in the same execute packet. Figure 5-32(a) shows a **NOP** in an execute packet (in parallel) with other code. The results of the **LD**, **ADD**, and **MPY** is available during the proper cycle for each instruction. Hence, **NOP** has no effect on the execute packet.

Figure 5-32(b) shows the replacement of the single-cycle NOP with a multicycle NOP (NOP5) in the same execute packet. The NOP5 causes no operation to perform other than the operations from the instructions inside its execute packet. The results of the LD, ADD, and MPY cannot be used by any other instructions until the NOP5 period has completed.

**Figure 5-32 Multicycle NOP in an Execute Packet**

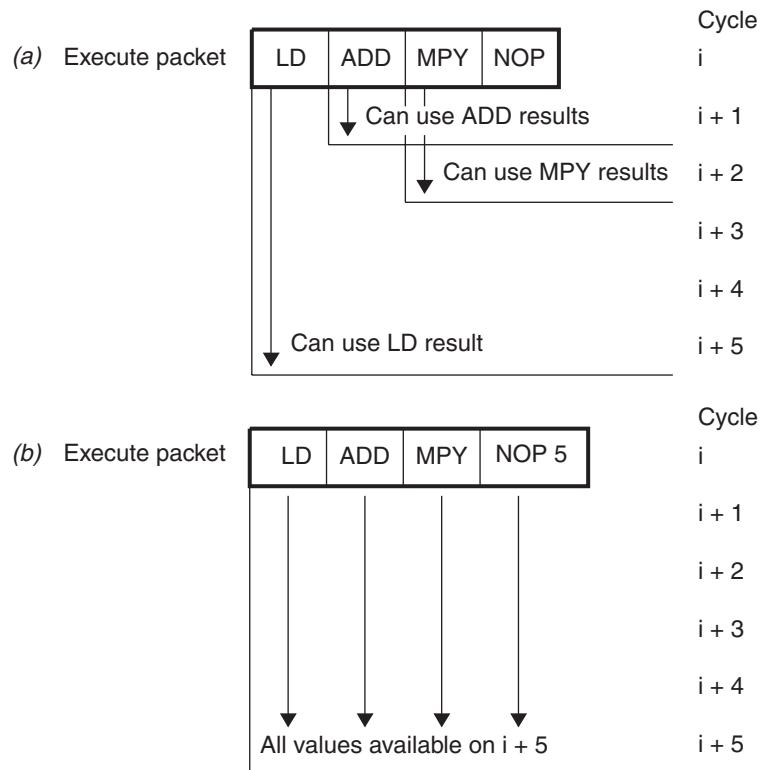


Figure 5-33 shows how a multicycle **NOP** can be affected by a branch. If the delay slots of a branch finish while a multicycle **NOP** is still dispatching **NOPs** into the pipeline, the branch overrides the multicycle **NOP** and the branch target begins execution five delay slots after the branch was issued.

**Figure 5-33 Branching and Multicycle NOPs**

Cycle #		Pipeline Phase	
		Branch	Target
1	EP1	B . . .	E1 PG
2	EP2	EP without branch	(A) PS
3	EP3	EP without branch	(A) PW
4	EP4	EP without branch	(A) PR
5	EP5	EP without branch	(A) DP
6	EP6	LD MPY ADD NOP5	(A) DC
7	Branch EP7	Branch will execute here	E1
8			
9			
10			
11	Normal EP7	See Figure 4-21 (b)	

(A) Delay slots of the branch

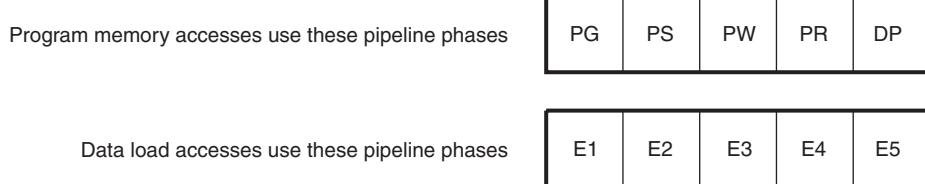
In one case, execute packet 1 (EP1) does not have a branch. The **NOP 5** in EP6 forces the CPU to wait until cycle 11 to execute EP7.

In the other case, EP1 does have a branch. The delay slots of the branch coincide with cycles 2 through 6. Once the target code reaches E1 in cycle 7, it executes.

### 5.4.3 Memory Considerations

The DSP to access memory at a high speed. These phases are shown in Figure 5-34.

**Figure 5-34 Pipeline Phases Used During Memory Accesses**



To understand the memory accesses, compare data loads and instruction fetches/dispatches. The comparison is valid because data loads and program fetches operate on internal memories of the same speed on the DSP and perform the same types of operations (listed in [Table 5-45](#)) to accommodate those memories. [Table 5-45](#) shows the operation of program fetches pipeline versus the operation of a data load.

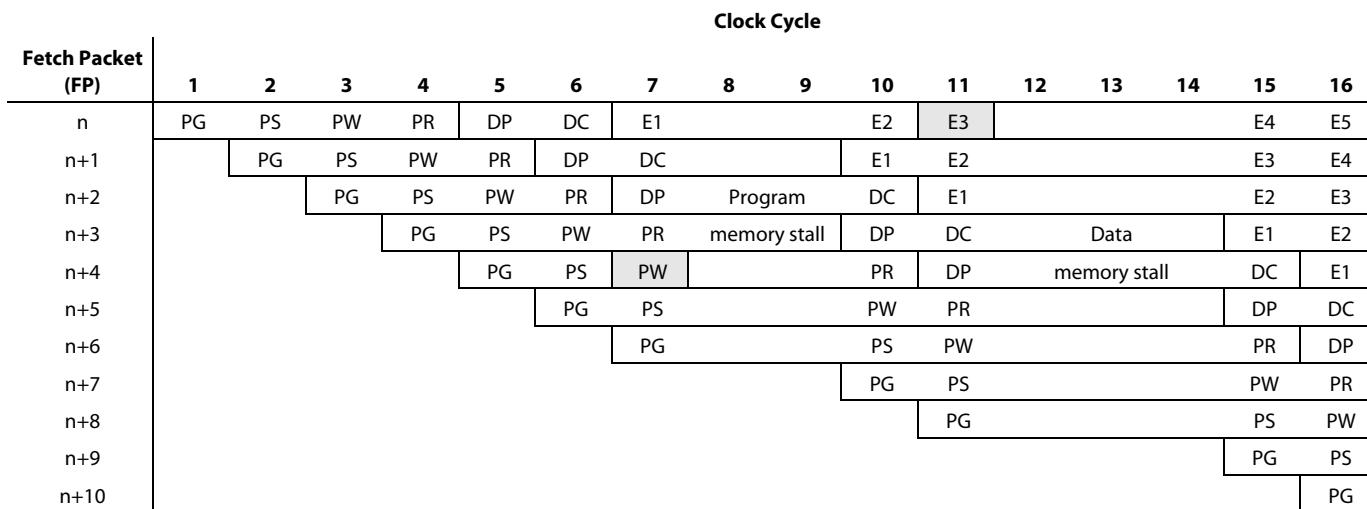
**Table 5-45 Program Memory Accesses Versus Data Load Accesses**

Operation	Program Memory Access Phase	Data Load Access Phase
Compute address	PG	E1
Send address to memory	PS	E2
Memory read/write	PW	E3
Program memory: receive fetch packet at CPU boundary	PR	E4
Data load: receive data at CPU boundary		
Program memory: send instruction to functional units	DP	E5
Data load: send data to register		

Depending on the type of memory and the time required to complete an access, the pipeline may stall to ensure proper coordination of data and instructions.

A memory stall occurs when memory is not ready to respond to an access from the CPU. This access occurs during the PW phase for a program memory access and during the E3 phase for a data memory access. The memory stall causes all of the pipeline phases to lengthen beyond a single clock cycle, causing execution to take additional clock cycles to finish. The results of the program execution are identical whether a stall occurs or not. [Figure 5-35](#) illustrates this point.

**Figure 5-35 Program and Data Memory Stalls**





# Interrupts

This chapter describes CPU interrupts, including reset and the nonmaskable interrupt (NMI). It details the related CPU control registers and their functions in controlling interrupts. It also describes interrupt processing, the method the CPU uses to detect automatically the presence of interrupts and divert program execution flow to your interrupt service code. Finally, this chapter describes the programming implications of interrupts.

- 6.1 ["Overview" on page 6-2](#)
- 6.2 ["Globally Enabling and Disabling Interrupts" on page 6-10](#)
- 6.3 ["Individual Interrupt Control" on page 6-13](#)
- 6.4 ["Interrupt Detection and Processing" on page 6-16](#)
- 6.4 ["Interrupt Detection and Processing" on page 6-16](#)
- 6.5 ["Performance Considerations" on page 6-25](#)
- 6.6 ["Programming Considerations" on page 6-26](#)

## 6.1 Overview

Typically, DSPs work in an environment that contains multiple external asynchronous events. These events require tasks to be performed by the DSP when they occur. An interrupt is an event that stops the current process in the CPU so that the CPU can attend to the task needing completion because of the event. These interrupt sources can be on chip or off chip, such as timers, analog-to-digital converters, or other peripherals.

Servicing an interrupt involves saving the context of the current process, completing the interrupt task, restoring the registers and the process context, and resuming the original process. There are eight registers that control servicing interrupts.

An appropriate transition on an interrupt pin sets the pending status of the interrupt within the interrupt flag register (IFR). If the interrupt is properly enabled, the CPU begins processing the interrupt and redirecting program flow to the interrupt service routine.

### 6.1.1 Types of Interrupts and Signals Used

There are four types of interrupts on the CPU.

- Reset
- Maskable
- Nonmaskable
- Exception

 **Note**—The nonmaskable interrupt (NMI) is not supported on all C6000 devices, see the device-specific data manual for more information.

These first three types are differentiated by their priorities, as shown in [Table 6-1](#). The reset interrupt has the highest priority and corresponds to the RESET signal. The nonmaskable interrupt (NMI) has the second highest priority and corresponds to the NMI signal. The lowest priority interrupts are interrupts 4-15 corresponding to the INT4-INT15 signals. RESET, NMI, and some of the INT4-INT15 signals are mapped to pins on C6000 devices. Some of the INT4-INT15 interrupt signals are used by internal peripherals and some may be unavailable or can be used under software control. Check your device-specific data sheet to see your interrupt specifications.

The CPU supports exceptions as another type of interrupt. When exceptions are enabled, the NMI input behaves as an exception. This chapter does not deal in depth with exceptions, as it assumes for discussion of NMI as an interrupt that they are disabled. Chapter 7 “[CPU Exceptions](#)” on page 7-1 discusses exceptions including NMI behavior as an exception.



**CAUTION—Code Compatibility** The CPU using NMI as an interrupt is assured only when exceptions are not enabled. Any additional or modified code requiring the use of NMI as an exception to ensure correct behavior will likely require changes to the pre-existing code to adjust for the additional functionality added by enabling exceptions.

**Table 6-1** Interrupt Priorities

Priority	Interrupt Name	Interrupt Type
Highest	Reset	Reset
	NMI	Nonmaskable
	INT4	Maskable
	INT5	Maskable
	INT6	Maskable
	INT7	Maskable
	INT8	Maskable
	INT9	Maskable
	INT10	Maskable
	INT11	Maskable
	INT12	Maskable
	INT13	Maskable
	INT14	Maskable
Lowest	INT15	Maskable

### 6.1.1.1 Reset (RESET)

Reset is the highest priority interrupt and is used to halt the CPU and return it to a known state. The reset interrupt is unique in a number of ways:

- $\overline{\text{RESET}}$  is an active-low signal. All other interrupts are active-high signals.
- $\overline{\text{RESET}}$  must be held low for 10 clock cycles before it goes high again to reinitialize the CPU properly.
- The instruction execution in progress is aborted and all registers are returned to their default states.
- The reset interrupt service fetch packet must be located at a specific address which is specific to the specific device. See the device data sheet for more information.
- $\overline{\text{RESET}}$  is not affected by branches.

### 6.1.1.2 Nonmaskable Interrupt (NMI)



**Note**—The nonmaskable interrupt (NMI) is not supported on all C6000 devices, see your device-specific data manual for more information.

NMI is the second-highest priority interrupt and is generally used to alert the CPU of a serious hardware problem such as imminent power failure.

For NMI processing to occur, the nonmaskable interrupt enable (NMIE) bit in the interrupt enable register (IER) must be set to 1. If NMIE is set to 1, the only condition that can prevent NMI processing is if the NMI occurs during the delay slots of a branch (whether the branch is taken or not).

NMIE is cleared to 0 at reset to prevent interruption of the reset. It is cleared at the occurrence of an NMI to prevent another NMI from being processed. You cannot manually clear NMIE, but you can set NMIE to allow nested NMIs. While NMI is cleared, all maskable interrupts (INT4-INT15) are disabled.

On the CPU, if an NMI is recognized within an SPLOOP operation, the behavior is the same as for an NMI with exceptions enabled. The SPLOOP operation terminates immediately (loop does not wind down as it does in case of an interrupt). The SPLX bit in the NMI/exception task state register (NTSR) is set for status purposes. The NMI service routine must look at this as one of the factors on whether a return to the interrupted code is possible. If the SPLX bit in NTSR is set, then a return to the interrupted code results in incorrect operation. See Section [8.13](#) on page 8-36 for more information.

#### 6.1.1.3 Maskable Interrupts (INT4-INT15)

The CPUs have 12 interrupts that are maskable. These have lower priority than the NMI and reset interrupts as well as all exceptions. These interrupts can be associated with external devices, on-chip peripherals, software control, or not be available.

Assuming that a maskable interrupt does not occur during the delay slots of a branch (this includes conditional branches that do not complete execution due to a false condition), the following conditions must be met to process a maskable interrupt:

- The global interrupt enable bit (GIE) bit in the control status register (CSR) is set to 1.
- The NMIE bit in the interrupt enable register (IER) is set to 1.
- The corresponding interrupt enable (IE) bit in the IER is set to 1.
- The corresponding interrupt occurs, which sets the corresponding bit in the interrupt flags register (IFR) to 1 and there are no higher priority interrupt flag (IF) bits set in the IFR.

#### 6.1.1.4 Interrupt Service Table (IST)

When the CPU begins processing an interrupt, it references the interrupt service table (IST). The IST is a table of fetch packets that contain code for servicing the interrupts. The IST consists of 16 consecutive fetch packets. Each interrupt service fetch packet (ISFP) contains up to 14 instructions (either 8 32-bit instructions in a nonheader-based fetch packet or up to 14 instructions in a compact header-based fetch packet). A simple interrupt service routine may fit in an individual fetch packet.

The addresses and contents of the IST are shown in [Figure 6-1](#). Because each fetch packet contains eight 32-bit instruction words (or 32 bytes), each address in the table is incremented by 32 bytes (20h) from the one adjacent to it.

**Figure 6-1      Interrupt Service Table**

xxxx 000h	RESET ISFP
xxxx 020h	NMI ISFP
xxxx 040h	Reserved
xxxx 060h	Reserved
xxxx 080h	INT4 ISFP
xxxx 0A0h	INT5 ISFP
xxxx 0C0h	INT6 ISFP
xxxx 0E0h	INT7 ISFP
xxxx 100h	INT8 ISFP
xxxx 120h	INT9 ISFP
xxxx 140h	INT10 ISFP
xxxx 160h	INT11 ISFP
xxxx 180h	INT12 ISFP
xxxx 1A0h	INT13 ISFP
xxxx 1C0h	INT14 ISFP
xxxx 1E0h	INT15 ISFP

Program memory

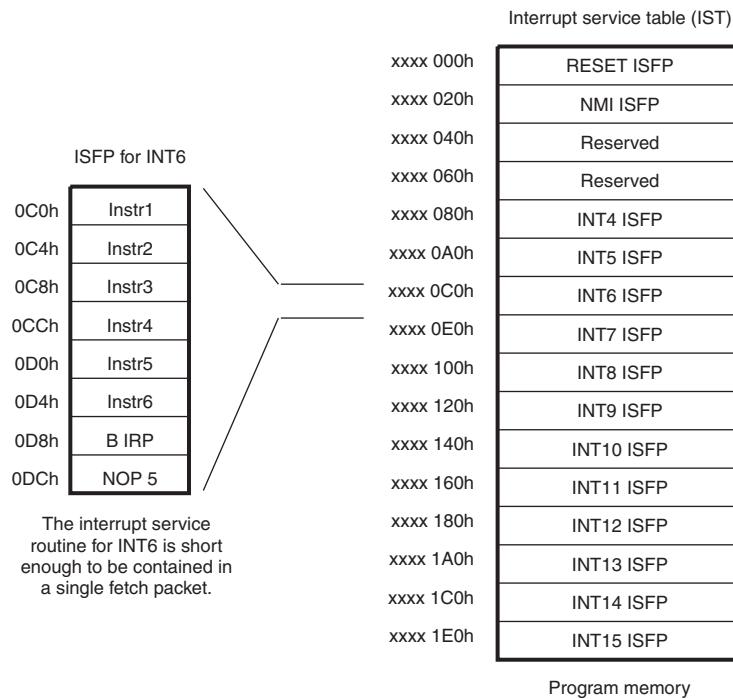
#### 6.1.1.5 Interrupt Service Fetch Packet (ISFP)

An ISFP is a fetch packet used to service an interrupt. [Figure 6-2](#) shows an ISFP that contains an interrupt service routine small enough to fit in a single fetch packet (FP). To branch back to the main program, the FP contains a branch to the interrupt return pointer instruction (**B IRP**). This is followed by a **NOP 5** instruction to allow the branch target to reach the execution stage of the pipeline.



**Note**—The ISFP should be exactly 8 words long. To prevent the compiler from using compact instructions (see section 3.10 “[Compact Instructions on the CPU](#)” on page 3-29), the interrupt service table should be preceded by a `.nocmp` directive. See the *TMS320C6000 Assembly Language Tools User’s Guide* ([SPRU186](#)).

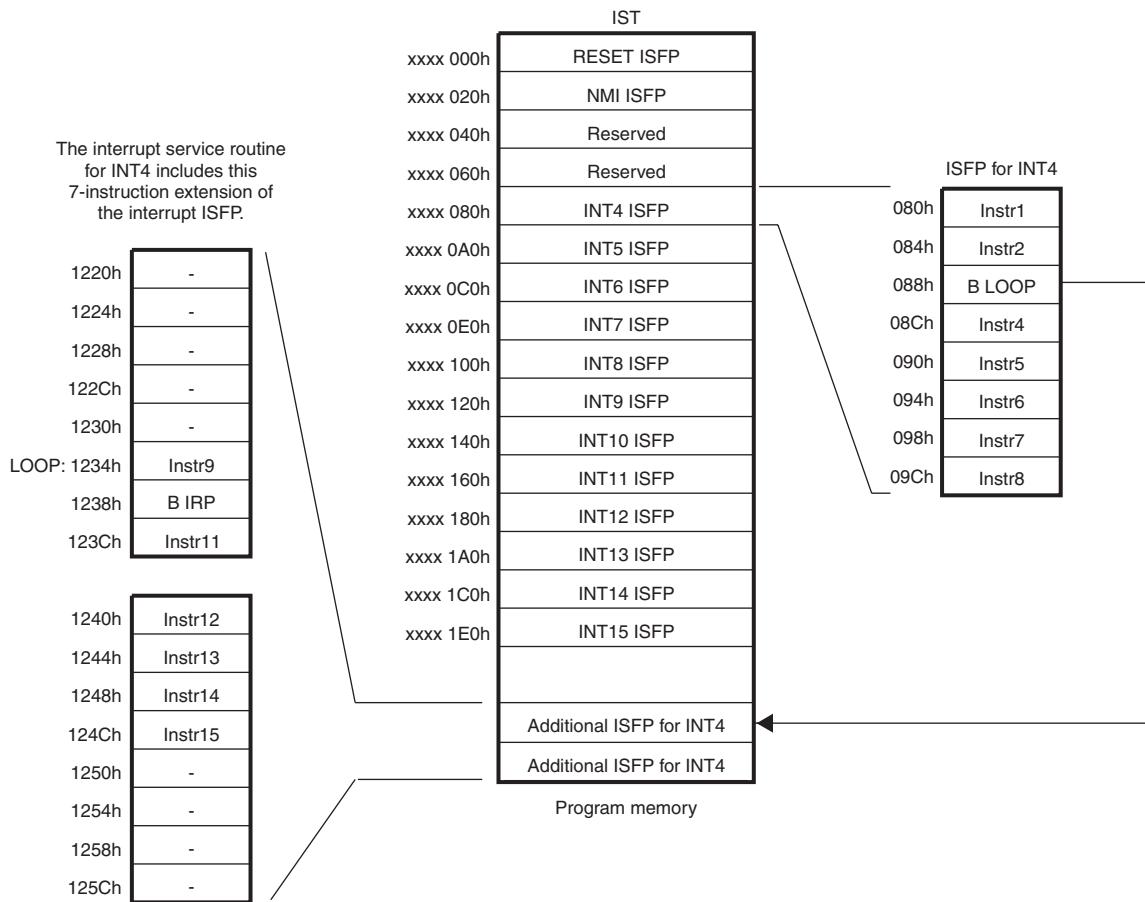
If the **NOP 5** was not in the routine, the CPU would execute the next five execute packets (some of which are likely to be associated with the next ISFP) because of the delay slots associated with the **B IRP** instruction. See section 5.2.6 “[Branch Instructions](#)” for more information.

**Figure 6-2      Interrupt Service Fetch Packet**


If the interrupt service routine for an interrupt is too large to fit in a single fetch packet, a branch to the location of additional interrupt service routine code is required.

**Figure 6-3** shows that the interrupt service routine for INT4 was too large for a single fetch packet, and a branch to memory location 1234h is required to complete the interrupt service routine.

**Note**—The instruction **B LOOP** branches into the middle of a fetch packet and processes code starting at address 1234h. The CPU ignores code from address 1220h–1230h, even if it is in parallel to code at address 1234h.

**Figure 6-3** Interrupt Service Table With Branch to Additional Interrupt Service Code Located Outside the IST


### 6.1.1.6 Interrupt Service Table Pointer (ISTP)

The reset fetch packet must be located at the default location (see device data manual for more information), but the rest of the IST can be at any program memory location that is on a 256-word boundary (that is, any 1K byte boundary). The location of the IST is determined by the interrupt service table base (ISTB) field of the interrupt service table pointer register (ISTP). The ISTP is shown in [Figure 2-11](#) and described in [Table 2-15](#) on page 2-22. [Example 6-1](#) shows the relationship of the ISTB to the table location.

Because the HPEINT field in ISTP gives the value of the highest priority interrupt that is both pending and enabled, the whole of ISTP gives the address of the highest priority interrupt that is both pending and enabled

#### Example 6-1 Relocation of Interrupt Service Table

*(a) Relocating the IST to 800h*

- 1) Copy IST, located between 0h and 200h, to the memory location between 800h and A00h.
- 2) Write 800h to ISTP: MVK 800h, A2  
MVC A2, ISTP

$$\text{ISTP} = 800h = 1000\ 0000\ 0000b$$

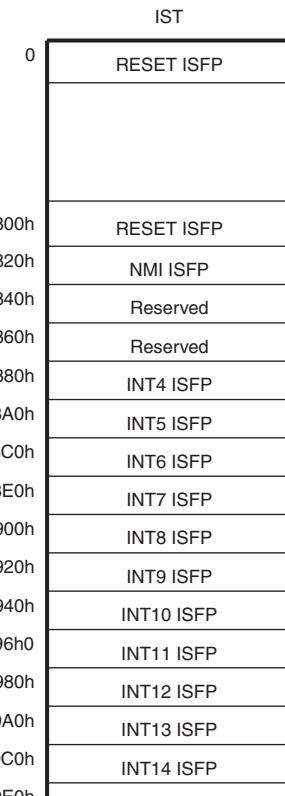
*(b) How the ISTP directs the CPU to the appropriate ISFP in the relocated IST*

Assume: IFR = BBC0h = 1011 1011 1100 0000b  
IER = 1230h = 0001 0010 0011 0001b

2 enabled interrupts pending: INT9 and INT12

The 1s in IFR indicate pending interrupts; the 1s in IER indicate the interrupts that are enabled. INT9 has a higher priority than INT12, so HPEINT is encoded with the value for INT9, 01001b.

HPEINT corresponds to bits 9-5 of the ISTP:  
ISTP = 1001 0010 0000b = 920h = address of INT9



Program memory

**End of Example 6-1**

## 6.1.2 Summary of Interrupt Control Registers

Table 6-2 lists the interrupt control registers on the CPU.

**Table 6-2      Interrupt Control Registers**

Acronym	Register Name	Description	Section
CSR	Control status register	Allows you to globally set or disable interrupts	2.8.4 "Control Status Register (CSR)" on page 2-15
ICR	Interrupt clear register	Allows you to clear flags in the IFR manually	2.8.6 "Interrupt Clear Register (ICR)" on page 2-18
IER	Interrupt enable register	Allows you to enable interrupts	2.8.7 "Interrupt Enable Register (IER)" on page 2-19
IFR	Interrupt flag register	Shows the status of interrupts	2.8.8 "Interrupt Flag Register (IFR)" on page 2-20
IRP	Interrupt return pointer register	Contains the return address used on return from a maskable interrupt. This return is accomplished via the <b>B IRP</b> instruction.	2.8.9 "Interrupt Return Pointer Register (IRP)" on page 2-20
ISR	Interrupt set register	Allows you to set flags in the IFR manually	2.8.10 "Interrupt Set Register (ISR)" on page 2-21
ISTP	Interrupt service table pointer register	Pointer to the beginning of the interrupt service table	2.8.11 "Interrupt Service Table Pointer Register (ISTP)" on page 2-22
ITSR	Interrupt task state register	Interrupted (non-NMI) machine state.	2.9.8 "Interrupt Task State Register (ITSR)" on page 2-28
NRP	Nonmaskable interrupt return pointer register	Contains the return address used on return from a nonmaskable interrupt. This return is accomplished via the <b>B NRP</b> instruction.	2.8.12 "Nonmaskable Interrupt (NMI) Return Pointer Register (NRP)" on page 2-23
NTSR	Nonmaskable interrupt task state register	Interrupted (NMI) machine state.	2.9.9 "NMI/Exception Task State Register (NTSR)" on page 2-29
TSR	Task state register	Allows you to globally set or disable interrupts. Contains status of current machine state.	2.9.14 "Task State Register (TSR)" on page 2-33

## 6.2 Globally Enabling and Disabling Interrupts

The control status register (CSR) contains two fields that control interrupts: GIE and PGIE, as shown in [Figure 2-3](#) and described in [Table 2-9](#) on page 2-15.

On the CPU, there is one physical GIE bit that is mapped to bit 0 of both CSR and TSR. Similarly, there is one physical PGIE bit. It is mapped as CSR.PGIE (bit 1) and ITSR.GIE (bit 0). Modification to either of these bits is reflected in both of the mappings. In the following discussion, references to the GIE bit in CSR also refer to the GIE bit in TSR, and references to the PGIE bit in CSR also refer to the GIE bit in ITSR.

The global interrupt enable (GIE) allows you to enable or disable all maskable interrupts by controlling the value of a single bit. GIE is bit 0 of both the control status register (CSR) and the task state register (TSR).

- GIE = 1 enables the maskable interrupts so that they are processed.
- GIE = 0 disables the maskable interrupts so that they are not processed.

The CPU detects interrupts in parallel with instruction execution. As a result, the CPU may begin interrupt processing in the same cycle that an **MVC** instruction writes 0 to GIE to disable interrupts. The PGIE bit (bit 1 of CSR) records the value of GIE after the CPU begins interrupt processing, recording whether the program was in the process of disabling interrupts.

During maskable interrupt processing, the CPU finishes executing the current execute packet. The CPU then copies the current value of GIE to PGIE, overwriting the previous value of PGIE. The CPU then clears GIE to prevent another maskable interrupt from occurring before the handler saves the machine’s state. ([Section 6.6.2](#) on page 6-27 discusses nesting interrupts.)

When the interrupt handler returns to the interrupted code with the **B IRP** instruction, the CPU copies PGIE back to GIE. When the interrupted code resumes, GIE reflects the last value written by the interrupted code.

Because interrupt detection occurs in parallel with CPU execution, the CPU can take an interrupt in the cycle immediately following an **MVC** instruction that clears GIE. The behavior of PGIE and the **B IRP** instruction ensures, however, that interrupts do not occur after subsequent execute packets. Consider the code in [Example 6-2](#).

### Example 6-2    Interrupts Versus Writes to GIE

```
-----  
;Assume GIE = 1  
MVC    CSR,B0      ; (1) Get CSR  
AND    -2,B0,B0;  (2) Get ready to clear GIE  
MVC    B0,CSR     ; (3) Clear GIE  
ADD    A0,A1,A2;  (4)  
ADD    A3,A4,A5;  (5)
```

**End of Example 6-2**

In [Example 6-2](#), the CPU may service an interrupt between instructions 1 and 2, between instructions 2 and 3, or between instructions 3 and 4. The CPU will not service an interrupt between instructions 4 and 5.

If the CPU services an interrupt between instructions 1 and 2 or between instructions 2 and 3, the PGIE bit will hold the value 1 when arriving at the interrupt service routine. If the CPU services an interrupt between instructions 3 and 4, the PGIE bit will hold the value 0. Thus, when the interrupt service routine resumes the interrupted code, it will resume with GIE set as the interrupted code intended.

On the CPU, programs must directly manipulate the GIE bit in CSR to disable and enable interrupts. [Example 6-3](#) and [Example 6-4](#) show code examples for disabling and enabling maskable interrupts globally, respectively.

#### **Example 6-3 Code Sequence to Disable Maskable Interrupts Globally**

```
MVC CSR,B0 ; get CSR
    AND -2,B0,B0; get ready to clear GIE
    MVC B0,CSR ; clear GIE
```

**End of Example 6-3**

#### **Example 6-4 Code Sequence to Enable Maskable Interrupts Globally**

```
MVC CSR,B0 ; get CSR
    OR 1,B0,B0 ; get ready to set GIE
    MVC B0,CSR ; set GIE
```

**End of Example 6-4**

The CPU copies TSR to ITSR, thereby, saving the old value of GIE. It then clears TSR.GIE. (ITSR.GIE is physically the same bit as CSR.PGIE and TSR.GIE is physically the same bit as CSR.GIE.) When returning from an interrupt with the **B IRP** instruction, the CPU restores the TSR state by copying ITSR back to TSR.

The CPU provides two new instructions that allow for simpler and safer manipulation of the GIE bit.

- The **DINT** instruction disables interrupts by:
  - Copies the value of CSR.GIE (and TSR.GIE) to TSR.SGIE
  - Clears CSR.GIE and TSR.GIE to 0 (disabling interrupts immediately)
 The CPU will not service an interrupt between the execute packet containing **DINT** and the execute packet that follows it.
- The **RINT** instruction restores interrupts to the previous state by:
  - Copies the value of TSR.SGIE to CSR.GIE (and TSR.GIE)
  - Clears TSR.SGIE to 0

If SGIE bit in TSR when **RINT** executes, interrupts are enabled immediately and the CPU may service an interrupt in the cycle immediately following the execute packet containing **RINT**.

**Example 6-5** illustrates the use and timing of the **DINT** instruction in disabling maskable interrupts globally and **Example 6-6** shows how to enable maskable interrupts globally using the complementary **RINT** instruction.

#### **Example 6-5 Code Sequence with Disable Global Interrupt Enable**

```
;Assume GIE = 1
ADD B0,1,B0; Interrupt possible between ADD and DINT
DINT        ; No interrupt between DINT and SUB
SUB B0,1,B0;
```

**End of Example 6-5**

#### **Example 6-6 Code Sequence with Restore Global Interrupt Enable**

```
;Assume SGIE == 1, GIE = 0
ADD B0,1,B0; No Interrupt between ADD and RINTRINT; Interrupt possible between RINT
and SUB
SUB B0,1,B0;
```

**End of Example 6-6**

**Example 6-7** shows a code fragment in which a load/modify/store is executed with interrupts disabled so that the register cannot be modified by an interrupt between the read and write operation. Since the **DINT** instruction saves the CSR.GIE bit to the TSR.SGIE bit and the **RINT** instruction copies the TSR.SGIE bit back to the CSR.GIE bit, if interrupts were disabled before the **DINT** instruction, they will still be disabled after the **RINT** instruction. If they were enabled before the **DINT** instruction, they will be enabled after the **RINT** instruction.

#### **Example 6-7 Code Sequence with Disable Reenable Interrupt Sequence**

```
DINT ; Disable interrupts
LDW *B0,B1      ; Load data
NOP 3           ; Wait for data to reach register
OR B1,1,B1     ; Set bit in word
STW B1,*B0      ; Store modified data back to original location
RINT           ; Re-enable interrupts
```

**End of Example 6-7**

 **Note**—The use of **DINT** and **RINT** instructions in a nested manner, like the following code:

```
DINT
DINT
RINT
RINT
```

leaves interrupts disabled after the second **RINT** instruction. The successive use of the **DINT** instruction leaves the TSR.SGIE bit cleared to 0, so the **RINT** instructions copy zero to the GIE bit.

## 6.3 Individual Interrupt Control

Servicing interrupts effectively requires individual control of all three types of interrupts: reset, nonmaskable, and maskable. Enabling and disabling individual interrupts is done with the interrupt enable register (IER). The status of pending interrupts is stored in the interrupt flag register (IFR). Manual interrupt processing can be accomplished through the use of the interrupt set register (ISR) and interrupt clear register (ICR). The interrupt return pointers restore context after servicing nonmaskable and maskable interrupts.

### 6.3.1 Enabling and Disabling Interrupts

You can enable and disable individual interrupts by setting and clearing bits in the IER that correspond to the individual interrupts. An interrupt can trigger interrupt processing only if the corresponding bit in the IER is set. Bit 0, corresponding to reset, is not writable and is always read as 1, so the reset interrupt is always enabled. You cannot disable the reset interrupt. Bits IE4-IE15 can be written as 1 or 0, enabling or disabling the associated interrupt, respectively. The IER is shown in [Figure 2-7](#) and described in [Table 2-12](#) on page 2-19.

When NMIE = 0, all nonreset interrupts are disabled, preventing interruption of an NMI. The NMIE bit is cleared at reset to prevent any interruption of process or initialization until you enable NMI. After reset, you must set the NMIE bit to enable the NMI and to allow INT15-INT4 to be enabled by the GIE bit in CSR and the corresponding IER bit. You cannot manually clear the NMIE bit; the NMIE bit is unaffected by a write of 0. The NMIE bit is also cleared by the occurrence of an NMI. If cleared, the NMIE bit is set only by completing a **B NRP** instruction or by a write of 1 to the NMIE bit. [Example 6-8](#) and [Example 6-9](#) show code for enabling and disabling individual interrupts, respectively.

#### Example 6-8 Code Sequence to Enable an Individual Interrupt (INT9)

```
MVK    200h,B1; set bit 9
      MVC    IER,B0 ; get IER
      OR     B1,B0,B0; get ready to set IE9
      MVC    B0,IER ; set bit 9 in IER
```

**End of Example 6-8**

#### Example 6-9 Code Sequence to Disable an Individual Interrupt (INT9)

```
MVK    FDFFh,B1 ; clear bit 9
      MVC    IER,B0
      AND    B1,B0,B0; get ready to clear IE9
      MVC    B0,IER ; clear bit 9 in IER
```

**End of Example 6-9**

### 6.3.2 Status of Interrupts

The interrupt flag register (IFR) contains the status of INT4-INT15 and NMI. Each interrupt's corresponding bit in IFR is set to 1 when that interrupt occurs; otherwise, the bits have a value of 0. If you want to check the status of interrupts, use the **MVC** instruction to read IFR. The IFR is shown in [Figure 2-8](#) and described in [Table 2-13](#) on page 2-20.

### 6.3.3 Setting and Clearing Interrupts

The interrupt set register (ISR) and the interrupt clear register (ICR) allow you to set or clear maskable interrupts manually in IFR. Writing a 1 to IS4-IS15 in ISR causes the corresponding interrupt flag to be set in IFR. Similarly, writing a 1 to a bit in ICR causes the corresponding interrupt flag to be cleared. Writing a 0 to any bit of either ISR or ICR has no effect. Incoming interrupts have priority and override any write to ICR. You cannot set or clear any bit in ISR or ICR to affect NMI or reset. The ISR is shown in [Figure 2-10](#) and described in [Table 2-14](#) on page 2-21. The ICR is shown in [Figure 2-6](#) and described in [Table 2-11](#) on page 2-18.



**Note**—Any write to the ISR or ICR (by the MVC instruction) effectively has one delay slot because the results cannot be read (by the MVC instruction) in IFR until two cycles after the write to ISR or ICR.

Any write to ICR is ignored by a simultaneous write to the same bit in ISR.

[Example 6-10](#) and [Example 6-11](#) show code examples to set and clear individual interrupts.

#### Example 6-10 Code to Set an Individual Interrupt (INT6) and Read the Flag Register

```
MVK      40h, B3
MVC      B3, ISR
NOP
MVC      IFR, B4
```

**End of Example 6-10**

#### Example 6-11 Code to Clear an Individual Interrupt (INT6) and Read the Flag Register

```
MVK      40h, B3
MVC      B3, ICR
NOP
MVC      IFR, B4
```

**End of Example 6-11**

### 6.3.4 Returning From Interrupt Servicing

Returning control from interrupts is handled differently for all three types of interrupts: reset, nonmaskable, and maskable.

#### 6.3.4.1 CPU State After RESET

After RESET, the control registers and bits contain the following values:

- AMR, ISR, ICR, and IFR = 0
- ISTP = Default value varies by device (See data manual for correct value)
- IER = 1
- IRP and NRP = undefined
- CSR bits 15-0
  - = 100h in little-endian mode
  - = 000h in big-endian mode
- TSR = 0
- ITSR = 0
- NTSR = 0

The program execution begins at the address specified by the ISTB field in ISTP.

### 6.3.4.2 Returning From Nonmaskable Interrupts

The NMI return pointer register (NRP), shown in [Figure 2-12](#) on page 2-23, contains the return pointer that directs the CPU to the proper location to continue program execution after NMI processing. A branch using the address in NRP (**B NRP**) in your interrupt service routine returns to the program flow when NMI servicing is complete. [Example 6-12](#) shows how to return from an NMI.

The NTSR register will be copied back into the TSR register during the transfer of control out of the interrupt.

#### **Example 6-12 Code to Return From NMI**

```
-----  
B   NRP      ; return, sets NMIE  
NOP 5       ; delay slots
```

**End of Example 6-12**

### 6.3.4.3 Returning From Maskable Interrupts

The interrupt return pointer register (IRP), shown in [Figure 2-9](#) on page 2-21, contains the return pointer that directs the CPU to the proper location to continue program execution after processing a maskable interrupt. A branch using the address in IRP (**B IRP**) in your interrupt service routine returns to the program flow when interrupt servicing is complete. [Example 6-13](#) shows how to return from a maskable interrupt.

The ITSR will be copied back into the TSR during the transfer of control out of the interrupt.

#### **Example 6-13 Code to Return from a Maskable Interrupt**

```
-----  
B       IRP ; return, moves PGIE to GIE  
NOP     5   ; delay slots
```

**End of Example 6-13**

## 6.4 Interrupt Detection and Processing

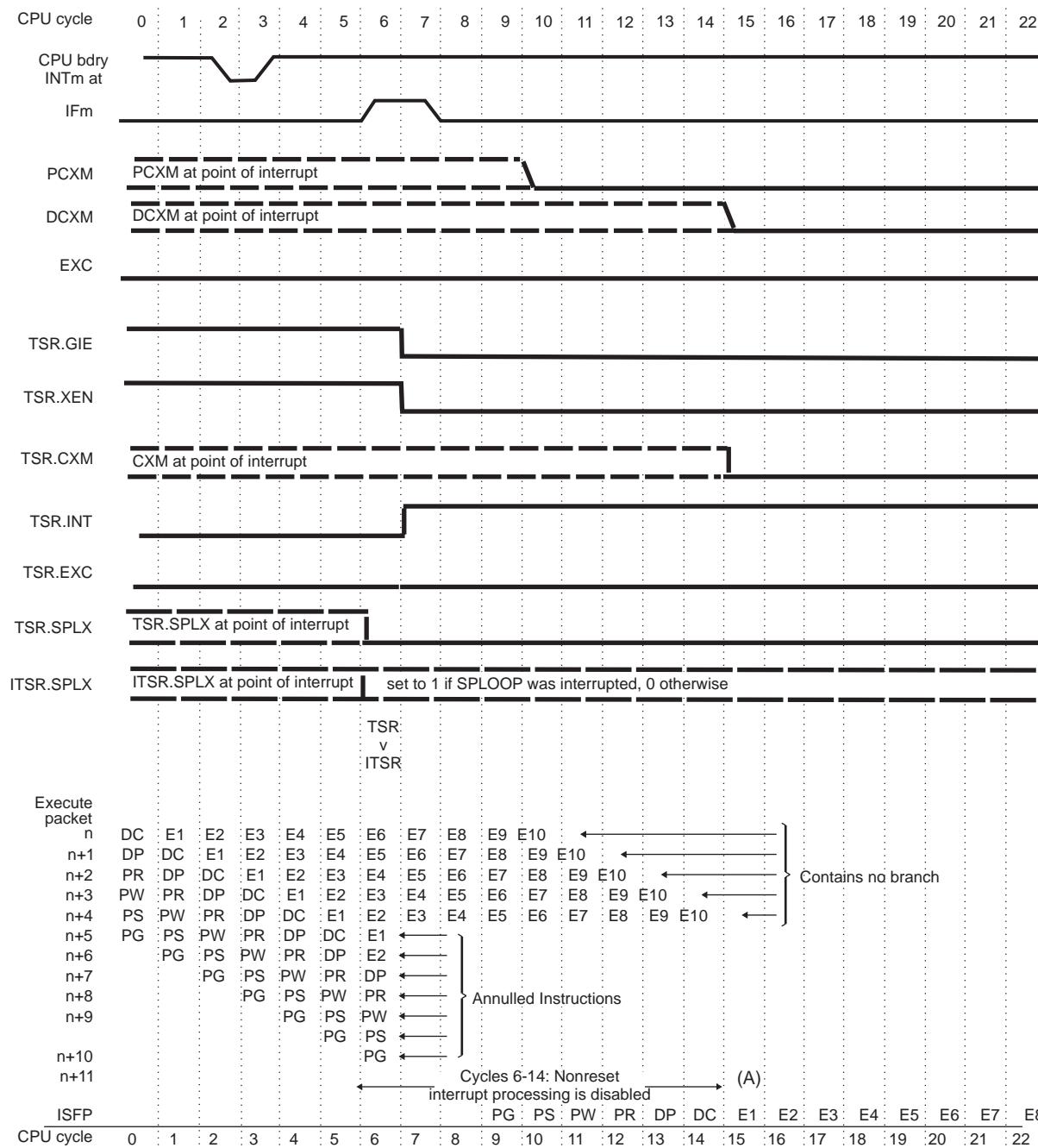
This section describes interrupts on the CPU.

When an interrupt occurs, it sets a flag in the interrupt flag register (IFR). Depending on certain conditions, the interrupt may or may not be processed. This section discusses the mechanics of setting the flag bit, the conditions for processing an interrupt, and the order of operation for detecting and processing an interrupt. The similarities and differences between reset and nonreset interrupts are also discussed.

### 6.4.1 Setting the Nonreset Interrupt Flag

Figure 6-4 shows the processing of a nonreset interrupt (INTm) for the CPU. The flag (IFm) for INTm in IFR is set following the low-to-high transition of the INTm signal on the CPU boundary. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle. Once there is a low-to-high transition on an interrupt pin, the interrupt is detected as pending inside the CPU. When the interrupt signal has been detected (cycle 4). Two clock cycles after detection, the interrupt's corresponding flag bit in IFR is set (cycle 6).

**Figure 6-4 Nonreset Interrupt Detection and Processing: Pipeline Operation**



(A) After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

In [Figure 6-4](#), IFm is set during CPU cycle 6. You could attempt to clear bit IFm by using an **MVC** instruction to write a 1 to bit m of ICR in execute packet n + 3 (during CPU cycle 4). However, in this case, the automated write by the interrupt detection logic takes precedence and IFm remains set.

[Figure 6-4](#) assumes INTm is the highest priority pending interrupt and is enabled by the GIE and NMIE bits, as necessary. If it is not the highest priority pending interrupt, IFm remains set until either you clear it by writing a 1 to bit m of ICR, or the processing of INTm occurs.

#### 6.4.1.1 Detection of Missed Interrupts

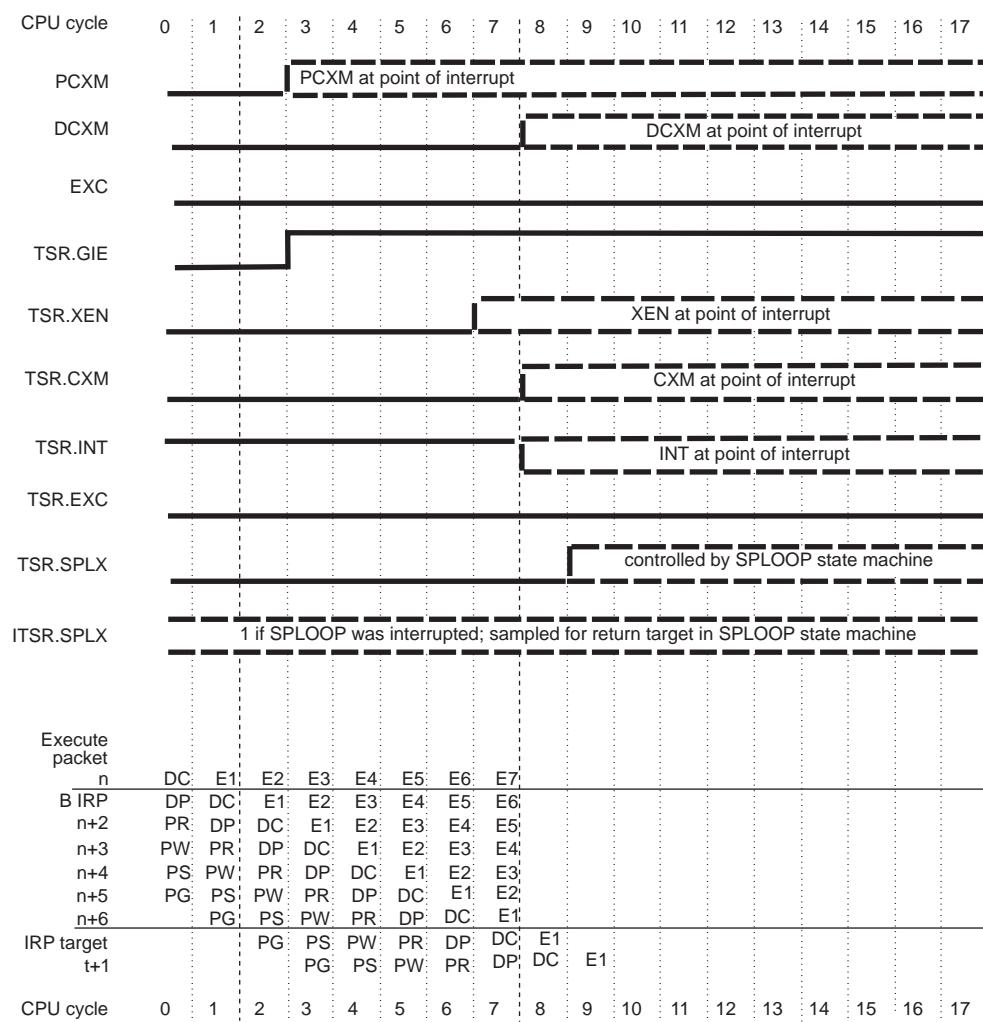
Each INTm input has a corresponding output that indicates if a low-to-high transition occurred on the input while the pending flag for that input had not yet been cleared. These outputs may be used by the interrupt controller to create an exception back to the CPU to notify the user of the missed interrupt. See the device-specific data manual to verify the device supports this feature.

#### 6.4.2 Conditions for Processing a Nonreset Interrupt

In clock cycle 4 of [Figure 6-4](#), a nonreset interrupt in need of processing is detected. For this interrupt to be processed, the following conditions must be valid on the same clock cycle and are evaluated every clock cycle:

- IFm is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the interrupt logic.)
- There is not a higher priority IFm bit set in IFR.
- The corresponding bit in IER is set (IEm = 1).
- GIE = 1
- NMIE = 1
- The five previous execute packets (n through n + 4) do not contain a branch (even if the branch is not taken) and are not in the delay slots of a branch.
- The two previous execute packets and the current execute packet (n + 3 through n + 5) do not contain an **SPLOOP**, **SPLOOPD**, or **SPLOOPW** instruction.
- If an SPLOOP is active, then the conditions set forth in 8.13.1 “[Interrupting the Loop Buffer](#)” apply.

Any pending interrupt will be taken as soon as pending branches are completed.

**Figure 6-5** Return from Interrupt Execution and Processing: Pipeline Operation

### 6.4.3 Saving TSR Context in Nonreset Interrupt Processing

When control is transferred to the interrupt processing sequence, the context needed to return from the ISR is saved in the interrupt task state register (ITSR). The task state register (TSR) is set for the default interrupt processing context. [Table 6-3](#) shows the behavior for each bit in TSR. [Figure 6-4](#) shows the timing of the changes to the TSR bits as well as the CPU outputs used in interrupt processing.

The current execution mode is held in a piped series of register bits allowing a change in the mode to progress from the PS phase through the E1 phase. Fetches from program memory use the PS-valid register which is only loaded at the start of a transfer of control. This value is an output on the program memory interface and is shown in the timing diagram as PCXM. As the target execute packet progresses through the pipeline, the new mode is registered for that stage. Each stage uses its registered version of the execution mode. The field in TSR is the E1-valid version of CXM. It always indicates the execution mode for the instructions executing in E1. The mode is used in the data memory interface, and is registered for all load/store instructions when they execute in E1. This is shown in the timing diagram as DCXM. Note that neither PCXM nor DCXM is visible in any register to you.

**Table 6-3 TSR Field Behavior When an Interrupt is Taken**

Bit	Field	Action
0	GIE	Saved to GIE bit in ITSR (will be 1). Cleared to 0.
1	SGIE	Saved to SGIE bit in ITSR. Cleared to 0.
2	GEE	Saved to GEE bit in ITSR. Unchanged.
3	XEN	Saved to XEN bit in ITSR. Cleared to 0.
7-6	CXM	Saved to CXM bits in ITSR. Cleared to 0 (Supervisor mode).
9	INT	Saved to INT bit in ITSR. Set to 1.
10	EXC	Saved to EXC bit in ITSR. Cleared to 0.
14	SPLX	SPLX is set in the TSR by the SPLOOP buffer whenever it is in operation. Upon interrupt, if the SPLOOP buffer is operating (thus SPLX = 1), then ITSR.SPLX will be set to 1, and the TSR.SPLX bit will be cleared to 0 after the SPLOOP buffer winds down and the interrupt vector is taken. See “Task State Register (TSR), Interrupt Task State Register (ITSR), and NMI/Exception Task State Register (NTSR)” on page 8-4 for more information on SPLOOP.
15	IB	Saved to IB bit in ITSR (will be 0). Set by CPU control logic.

#### 6.4.4 Actions Taken During Nonreset Interrupt Processing

During CPU cycles 6-14 of [Figure 6-4](#), the following interrupt processing actions occur:

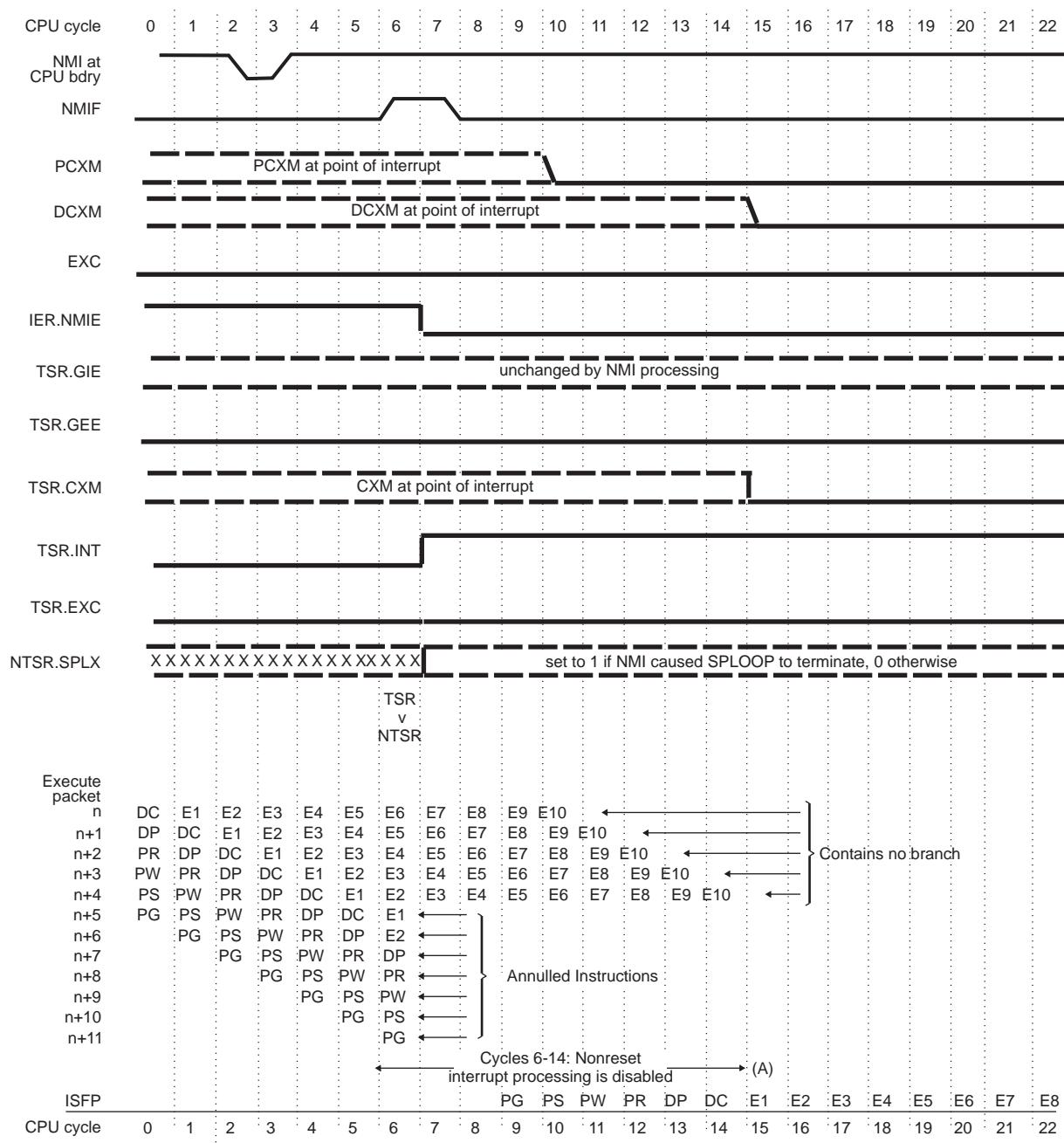
- Processing of subsequent nonreset interrupts is disabled.
- The PGIE bit is set to the value of the GIE bit and then the GIE bit is cleared. TSR context is saved into ITSR, and TSR is set to default interrupt processing values as shown in [Table 6-3](#). Explicit (MVC) writes to the TSR are completed before the TSR is saved to the ITSR.
- The next execute packets (from n + 5 on) are annulled. If an execute packet is annulled during a particular pipeline stage, it does not modify any CPU state. Annulling also forces an instruction to be annulled in future pipeline stages.
- The address of the first annulled execute packet (n + 5) is loaded in IRP.
- A branch to the address formed from ISTP (the pointer to the ISFP for INTm) is forced into the E1 phase of the pipeline during cycle 9.
- IFm is cleared during cycle 8.

#### 6.4.5 Conditions for Processing a Nonmaskable Interrupt

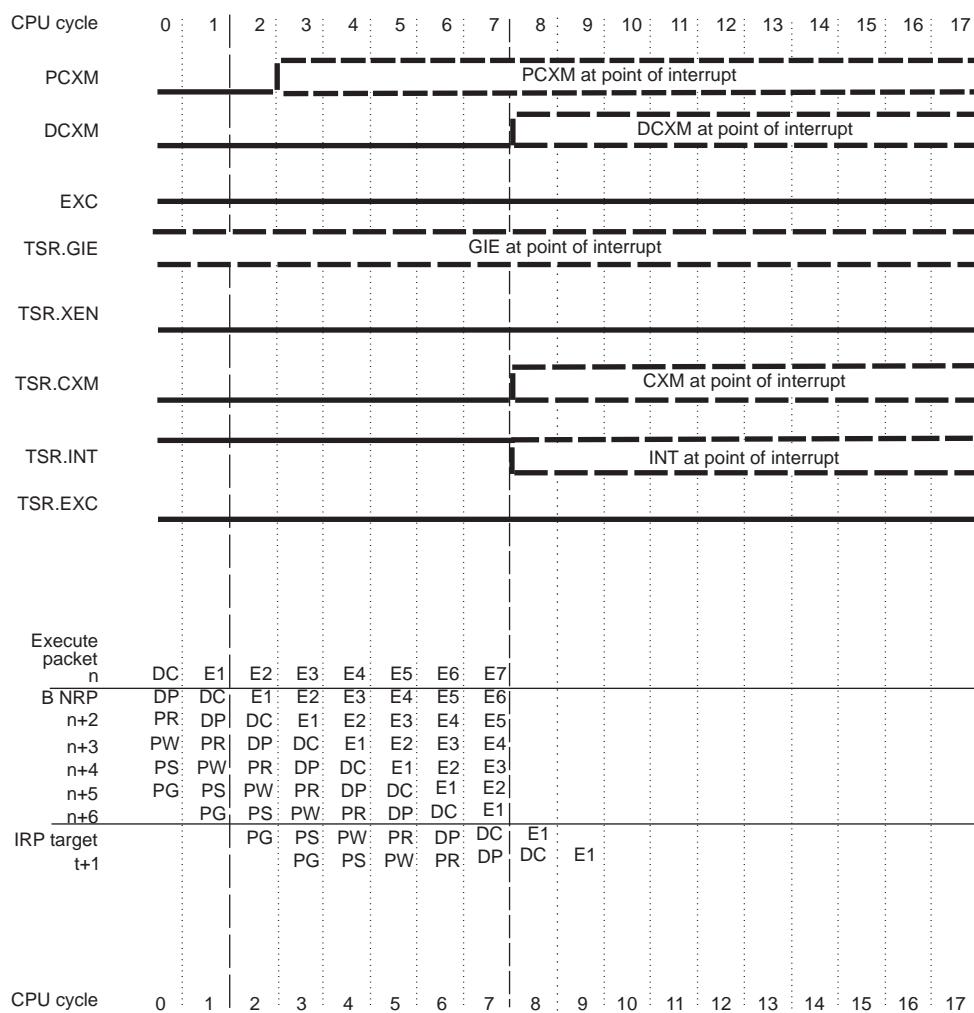
In clock cycle 4 of [Figure 6-6](#), a nonmaskable interrupt (NMI) in need of processing is detected. For this interrupt to be processed, the following conditions must be valid on the same clock cycle and are evaluated every clock cycle:

- The NMIF bit is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the interrupt logic.)
- Reset is not active.
- NMIE = 1
- The five previous execute packets (n through n + 4) do not contain a branch (even if the branch is not taken) and are not in the delay slots of a branch. Note that this functionality has changed when exceptions are enabled, see Chapter 7 “[CPU Exceptions](#)” on page 7-1 for more information.

A pending NMI will be taken as soon as pending branches are completed.

**Figure 6-6** CPU Nonmaskable Interrupt Detection and Processing: Pipeline Operation

(A) After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

**Figure 6-7 CPU Return from Nonmaskable Interrupt Execution and Processing: Pipeline Operation**


#### 6.4.6 Saving of Context in Nonmaskable Interrupt Processing

When control is transferred to the interrupt processing sequence, the context needed to return from the ISR is saved in the nonmaskable interrupt task state register (NTSR). The task state register (TSR) is set for the default NMI processing context. [Table 6-4](#) shows the behavior for each bit in TSR. [Figure 6-6](#) shows the timing of the changes to the TSR bits as well as the CPU outputs used in interrupt processing.

**Table 6-4 TSR Field Behavior When an NMI Interrupt is Taken**

Bit	Field	Action
0	GIE	Saved to GIE bit in NTSR. Unchanged.
1	SGIE	Saved to SGIE bit in NTSR. Cleared to 0.
2	GEE	Saved to GEE bit in NTSR. Unchanged.
3	XEN	Saved to XEN bit in NTSR. Cleared to 0.
7-6	CXM	Saved to CXM bits in NTSR. Cleared to 0 (Supervisor mode).
9	INT	Saved to INT bit in NTSR. Set to 1.
10	EXC	Saved to EXC bit in NTSR. Cleared to 0.
14	SPLX	SPLX is set in the TSR by the SPLOOP buffer whenever it is in operation. Upon interrupt, if the SPLOOP buffer is operating (thus SPLX = 1), then ITSR.SPLX will be set to 1, and the TSR.SPLX bit will be cleared to 0 after the SPLOOP buffer winds down and the interrupt vector is taken. See “ <a href="#">Task State Register (TSR), Interrupt Task State Register (ITSR), and NMI/Exception Task State Register (NTSR)</a> ” on page 8-4 for more information on SPLOOP.
15	IB	Saved to IB bit in NTSR. Set by CPU control logic.

#### 6.4.7 Actions Taken During Nonmaskable Interrupt Processing

During CPU cycles 6-14 of [Figure 6-6](#), the following interrupt processing actions occur:

- Processing of subsequent nonreset interrupts is disabled.
- The GIE and PGIE bits are unchanged. TSR context is saved into NTSR, and TSR is set to default NMI processing values as shown in [Table 6-4](#).
- The NMIE bit is cleared.
- The next execute packets (from n + 5 on) are annulled. If an execute packet is annulled during a particular pipeline stage, it does not modify any CPU state. Annulling also forces an instruction to be annulled in future pipeline stages.
- The address of the first annulled execute packet (n + 5) is loaded in NRP.
- A branch to the NMI ISFP (derived from ISTP) is forced into the E1 phase of the pipeline during cycle 9.
- NMIF is cleared during cycle 8.

#### 6.4.8 Setting the RESET Interrupt Flag

RESET must be held low for a minimum of 10 clock cycles. Four clock cycles after RESET goes high, processing of the reset vector begins. The flag for RESET (IF0) in the IFR is set by the low-to-high transition of the RESET signal on the CPU boundary. In [Figure 6-8](#), IF0 is set during CPU cycle 15. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle.

### 6.4.9 Actions Taken During **RESET** Interrupt Processing

A low signal on the **RESET** pin is the only requirement to process a reset. Once **RESET** makes a high-to-low transition, the pipeline is flushed and CPU registers are returned to their reset values. The GIE bit, the NMIE bit, and the ISTB bits in ISTP are cleared. For the CPU state after reset, see “[CPU State After RESET](#)” on page 6-14.

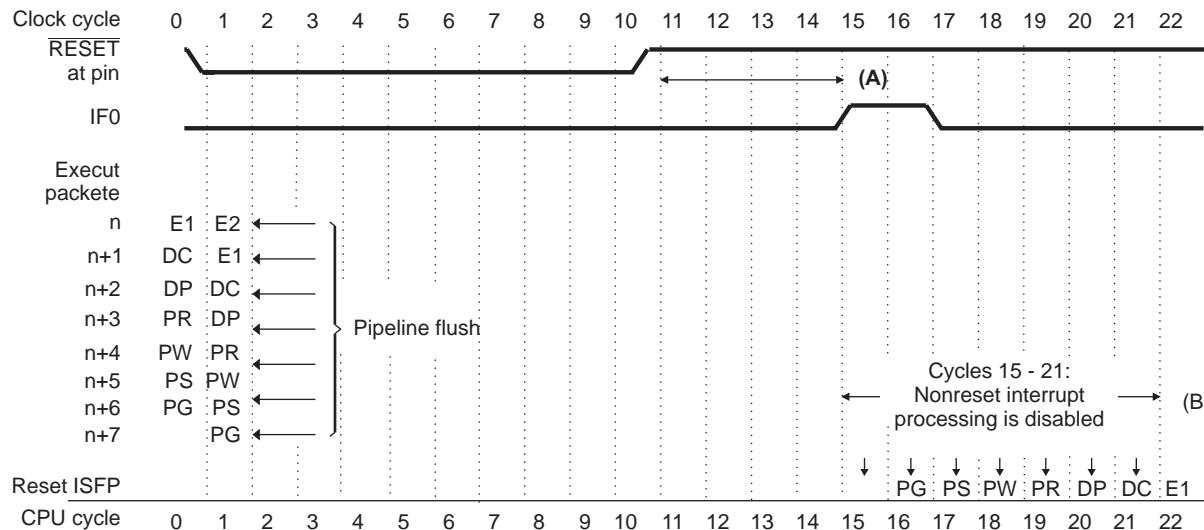
Note that a nested exception can force an internally-generated reset that does not reset all the registers to their hardware reset state. See “[Nested Exceptions](#)” on page 7-12 for more information.

During CPU cycles 15-21 of [Figure 6-8](#), the following reset processing actions occur:

- Processing of subsequent nonreset interrupts is disabled because the GIE and NMIE bits are cleared.
- A branch to the address held in ISTP (the pointer to the ISFP for INT0) is forced into the E1 phase of the pipeline during cycle 16.
- IF0 is cleared during cycle 17.

 **Note**—Code that starts running after reset must explicitly enable the GIE bit, the NMIE bit, and IER to allow interrupts to be processed.

**Figure 6-8** **RESET** Interrupt Detection and Processing: Pipeline Operation



(A) IF0 is set on the next CPU cycle boundary after a 4-clock cycle delay after the rising edge of.

(B) After this point, interrupts are still disabled. All nonreset interrupts are disabled when NMIE = 0. All maskable interrupts are disabled when GIE = 0.

## 6.5 Performance Considerations

The interaction of the C6000 CPU and sources of interrupts present performance issues for you to consider when you are developing your code.

### 6.5.1 General Performance

- **Overhead.** Overhead for all CPU interrupts is 9 cycles. You can see this in [Figure 6-4](#) and [Figure 6-5](#), where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 14.
- **Latency.** Interrupt latency is 13 cycles (21 cycles for  $\overline{\text{RESET}}$ ). In [Figure 6-6](#), although the interrupt is active in cycle 2, execution of interrupt service code does not begin until cycle 15.
- **Frequency.** The logic clears the nonreset interrupt (IFm) on cycle 8, with any incoming interrupt having highest priority. Thus, an interrupt is can be recognized every second cycle. Also, because a low-to-high transition is necessary, an interrupt can occur only every second cycle. However, the frequency of interrupt processing depends on the time required for interrupt service and whether you reenable interrupts during processing, thereby allowing nested interrupts. Effectively, only two occurrences of a specific interrupt can be recognized in two cycles.

### 6.5.2 Pipeline Interaction

Because the serial or parallel encoding of fetch packets does not affect the DC and subsequent phases of the pipeline, no conflicts between code parallelism and interrupts exist. There are three operations or conditions that can affect or are affected by interrupts:

- **Branches.** Nonreset interrupts are delayed, if any execute packets n through n + 4 in contain a branch or are in the delay slots of a branch.
- **Memory stalls.** Memory stalls delay interrupt processing, because they inherently extend CPU cycles.
- **Multicycle NOPs.** Multicycle NOPs (including the IDLE instruction) operate like other instructions when interrupted, except when an interrupt causes annulment of any but the first cycle of a multicycle NOP. In that case, the address of the next execute packet in the pipeline is saved in NRP or IRP. This prevents returning to an IDLE instruction or a multicycle NOP that was interrupted.

## 6.6 Programming Considerations

The interaction of the C6000 CPUs and sources of interrupts present programming issues for you to consider when you are developing your code.

### 6.6.1 Single Assignment Programming

Using the same register to store different variables (called here: multiple assignment) can result in unpredictable operation when the code can be interrupted.

To avoid unpredictable operation, you must employ the single assignment method in code that can be interrupted. When an interrupt occurs, all instructions entering E1 prior to the beginning of interrupt processing are allowed to complete execution (through E5). All other instructions are annulled and refetched upon return from interrupt. The instructions encountered after the return from the interrupt do not experience any delay slots from the instructions prior to processing the interrupt. Thus, instructions with delay slots prior to the interrupt can appear, to the instructions after the interrupt, to have fewer delay slots than they actually have.

[Example 6-14](#) shows a code fragment that stores two variables into A1 using multiple assignment. [Example 6-15](#) shows equivalent code using the single assignment programming method, which stores the two variables into two different registers.

For example, before reaching the code in [Example 6-14](#), suppose that register A1 contains 0 and register A0 points to a memory location containing a value of 10. The **ADD** instruction, which is in a delay slot of the **LDW**, sums A2 with the value in A1 (0) and the result in A3 is just a copy of A2. If an interrupt occurred between the **LDW** and **ADD**, the **LDW** would complete the update of A1 (10), the interrupt would be processed, and the **ADD** would sum A1 (10) with A2 and place the result in A3 (equal to A2 + 10). Obviously, this situation produces incorrect results.

In [Example 6-15](#), the single assignment method is used. The register A1 is assigned only to the **ADD** input and not to the result of the **LDW**. Regardless of the value of A6 with or without an interrupt, A1 does not change before it is summed with A2. Result A3 is equal to A2.

#### Example 6-14 Code Without Single Assignment: Multiple Assignment of A1

```
-----  
LDW  .D1      *A0,A1  
ADD  .L1      A1,A2,A3  
NOP  3  
MPY  .M1      A1,A4,A5 ; uses new A1
```

**End of Example 6-14**

#### Example 6-15 Code Using Single Assignment

```
-----  
LDW  .D1      *A0,A6  
ADD  .L1      A1,A2,A3  
NOP  3  
MPY  .M1      A6,A4,A5 ; uses A6
```

**End of Example 6-15**

Another method for preventing problems with nonsingle-assignment programming would be to disable interrupts before using multiple assignment, then reenable them afterwards. Of course, you must be careful with the trade-off between high-speed code that uses multiple-assignment and increasing interrupt latency. When using multiple assignment within software pipelined code, the SPLOOP buffer on the CPU can help you deal with the trade-off between performance and interruptibility. See “[Software Pipelined Loop \(SPLOOP\) Buffer](#)” on page 8-1 for more information.

### 6.6.2 Nested Interrupts

Generally, when the CPU enters an interrupt service routine, interrupts are disabled. However, when the interrupt service routine is for one of the maskable interrupts (INT4-INT15), an NMI can interrupt processing of the maskable interrupt. In other words, an NMI can interrupt a maskable interrupt, but neither an NMI nor a maskable interrupt can interrupt an NMI.

Also, there may be times when you want to allow an interrupt service routine to be interrupted by another (particularly higher priority) interrupt. Even though the processor by default does not allow interrupt service routines to be interrupted unless the source is an NMI, it is possible to nest interrupts under software control. To allow nested interrupts, the interrupt service routine must perform the following initial steps in addition to its normal work of saving any registers (including control registers) that it modifies:

1. The contents of IRP (or NRP) must be saved
2. The contents of the PGIE bit must be saved
3. The contents of ITSR must be saved
4. The GIE bit must be set to 1

Prior to returning from the interrupt service routine, the code must restore the registers saved above as follows:

1. The GIE bit must be first cleared to 0
2. The PGIE bit saved value must be restored
3. The contents of ITSR must be restored
4. The IRP (or NRP) saved value must be restored

Although steps 2, 3, and 4 above may be performed in any order, it is important that the GIE bit is cleared first. This means that the GIE and PGIE bits must be restored with separate writes to CSR. If these bits are not restored separately, then it is possible that the PGIE bit is overwritten by nested interrupt processing just as interrupts are being disabled.



---

**Note**—When coding nested interrupts for the CPU, the ITSR should be saved and restored to prevent corruption by the nested interrupt.

---

### 6.6.3 Manual Interrupt Processing (polling)

You can poll IFR and IER to detect interrupts manually and then branch to the value held in the ISTP as shown below in [Example 6-16](#).

The code sequence begins by copying the address of the highest priority interrupt from the ISTP to the register B2. The next instruction extracts the number of the interrupt, which is used later to clear the interrupt. The branch to the interrupt service routine comes next with a parallel instruction to set up the ICR word.

The last five instructions fill the delay slots of the branch. First, the 32-bit return address is stored in the B2 register and then copied to the interrupt return pointer (IRP). Finally, the number of the highest priority interrupt, stored in B1, is used to shift the ICR word in B1 to clear the interrupt.

#### Example 6-16 Manual Interrupt Processing

```

-----+
MVC      ISTP,B2          ; get related ISFP address
EXTU    B2,23,27,B1       ; extract HPEINT
[B1]    B     B2           ; branch to interrupt
[B1]    MVKL   1,A0         ; setup ICR word
[B1]    MVKL   RETADR,B2   ; create return address
[B1]    MVKH   RETADR,B2   ;
[B1]    MVC    B2,IRP        ; save return address
[B1]    SHL    A0,B1,B1      ; create ICR word
[B1]    MVC    B1,ICR        ; clear interrupt flag
RETADR:   (Post interrupt service routine Code)
-----+

```

**End of Example 6-16**

### 6.6.4 Traps

A trap behaves like an interrupt, but is created and controlled with software. The trap condition can be stored in any one of the conditional registers: A1, A2, B0, B1, or B2. If the trap condition is valid, a branch to the trap handler routine processes the trap and the return.

[Example 6-17](#) and [Example 6-18](#) show a trap call and the return code sequence, respectively. In the first code sequence, the address of the trap handler code is loaded into register B0 and the branch is called. In the delay slots of the branch, the context is saved in the B0 register, the GIE bit is cleared to disable maskable interrupts, and the return pointer is stored in the B1 register.

The trap is processed with the code located at the address pointed to by the label TRAP\_HANDLER. If the B0 or B1 registers are needed in the trap handler, their contents must be stored to memory and restored before returning. The code shown in [Example 6-18](#) should be included at the end of the trap handler code to restore the context prior to the trap and return to the TRAP\_RETURN address.

#### Example 6-17 Code Sequence to Invoke a Trap

```

-----+
[A1]    MVKL  TRAP_HANDLER,B0 ;load 32-bit trap address
[A1]    MVKH  TRAP_HANDLER,B0
[A1]    B     B0           ;branch to trap handler
[A1]    MVC   CSR,B0        ;read CSR
[A1]    AND   -2,B0,B1      ;disable interrupts:GIE=0
-----+

```

```
[A1]      MVC     B1,CSR          ; write to CSR
[A1]      MVKL   TRAP_RETURN,B1    ; load 32-bit return address
[A1]      MVKH   TRAP_RETURN,B1
TRAP_RETURN:    (post-trap code)
```

**Note**—A1 contains the trap condition.

**End of Example 6-17**

---

#### Example 6-18 Code Sequence for Trap Return

---

```
B      B1
MVC   B0,CSR          ; return
NOP   4               ; restore CSR
                  ; delay slots
```

**End of Example 6-18**

Often traps are used to handle unexpected conditions in the execution of the code. The CPU provides explicit exception handling support which may be used for this purpose.

Another alternative to using traps as software triggered interrupts is the software interrupt capability (SWI) provided by the DSP/BIOS real-time kernel.



# CPU Exceptions

This chapter describes CPU exceptions on the CPU. It details the related CPU control registers and their functions in controlling exceptions. It also describes exception processing, the method the CPU uses to detect automatically the presence of exceptions and divert program execution flow to your exception service code. Finally, the chapter describes the programming implications of exceptions.

- 7.1 ["Overview" on page 7-2](#)
- 7.2 ["Exception Control" on page 7-5](#)
- 7.3 ["Exception Detection and Processing" on page 7-8](#)
- 7.4 ["Performance Considerations" on page 7-12](#)
- 7.5 ["Programming Considerations" on page 7-15](#)

## 7.1 Overview

The exception mechanism on the CPU is intended to support error detection and program redirection to error handling service routines. Error signals generated outside of the CPU are consolidated to one exception input to the CPU. Exceptions generated within the CPU are consolidated to one internal exception flag with information as to the cause in a register. Fatal errors detected outside of the CPU are consolidated and incorporated into the NMI input to the CPU.

### 7.1.1 Types of Exceptions and Signals Used

There are three types of exceptions on the CPU.

- One externally generated maskable exception
- One externally generated nonmaskable exception,
- A set of internally generated nonmaskable exceptions

Check the device-specific data manual for your external exception specifications.

#### 7.1.1.1 Reset (RESET)

While reset can be classified as an exception, its behavior is fully described in the chapter on interrupts and its operation is independent of the exception mechanism.

#### 7.1.1.2 Nonmaskable Interrupt (NMI)

NMI is also described in the interrupt chapter, and as stated there it is generally used to alert the CPU of a serious hardware problem. The intent of NMI on C6000 devices was clearly for use as an exception. However, the inability of NMI to interrupt program execution independent of branch delay slots lessens its usefulness as an exception input. By default the NMI input retains its behavior for backward compatibility. When used in conjunction with the exception mechanism it will be treated as a nonmaskable exception with the primary behavioral difference being that branch delay slots will not prevent the recognition of NMI. The new behavior is enabled when exceptions are globally enabled. This is accomplished by setting the global exception enable (GEE) bit in the task state register (TSR) to 1. Once the exception mechanism has been enabled, it remains enabled until a reset occurs (GEE can only be cleared by reset). When the GEE bit is set to 1, NMI behaves as an exception. All further discussion of NMI in this chapter is in reference to its behavior as an exception.

For NMI processing to occur, the nonmaskable interrupt enable (NMIE) bit in the interrupt enable register (IER) must be set to 1. If the NMIE bit is set to 1, the only condition that can prevent NMI processing is the CPU being stalled.

The NMIE bit is cleared to 0 at reset to prevent interruption of the reset processing. It is cleared at the occurrence of an NMI to prevent another NMI from being processed. You cannot manually clear NMIE, but you can set NMIE to allow nested NMIs. While NMIE is cleared, all external exceptions are disabled. Internal exceptions are not affected by NMIE.

When NMI is recognized as pending, the NMI exception flag (NXF) bit in the exception flag register (EFR) is set. Unlike the NMIF bit in the interrupt flag register (IFR), the NXF bit is not cleared automatically upon servicing of the NMI. The NXF bit remains set until manually cleared in the exception service routine.

Transitions on the NMI input while the NXF bit is set are ignored. In the event an attempt to clear the flag using the MVC instruction coincides with the automated write by the exception detection logic, the automatic write takes precedence and the NXF bit remains set.

#### 7.1.1.3 Exception (EXCEP)

EXCEP is the maskable external exception input to the CPU. It is enabled by the XEN bit in TSR. For this exception to be recognized, the XEN bit must be set to 1, the GEE bit must be set to 1, and the NMIE bit must be set to 1.

When EXCEP is recognized as pending, the external exception flag (EXF) bit in EFR is set. The EXF bit remains set until manually cleared in the exception service routine.

#### 7.1.1.4 Internal Exceptions

Internal exceptions are those generated within the CPU. There are multiple causes for internal exceptions. Examples are illegal opcodes, illegal behavior within an instruction, and resource conflicts. Once enabled by the GEE bit, internal exceptions cannot be disabled. They are recognized independently of the state of the NMIE and XEN exception enable bits.

Instructions that have already entered E1 before a context switch begins are allowed to complete. Any internal exceptions generated by these completing instructions are ignored. This is true for both interrupt and exception context switches.

When an internal exception is recognized as pending, the internal exception flag (IXF) bit in EFR is set. The IXF bit remains set until manually cleared in the exception service routine.

#### 7.1.1.5 Exception Acknowledgment

The exception processing (EXC) bit in TSR is provided as an output at the CPU boundary. This signal in conjunction with the IACK signal alerts hardware external to the CPU that an exception has occurred and is being processed.

### 7.1.2 Exception Service Vector

When the CPU begins processing an exception, it references the interrupt service table (IST). The NMI interrupt service fetch packet (ISFP) is the fetch packet used to service all exceptions (external, internal, and NMI).

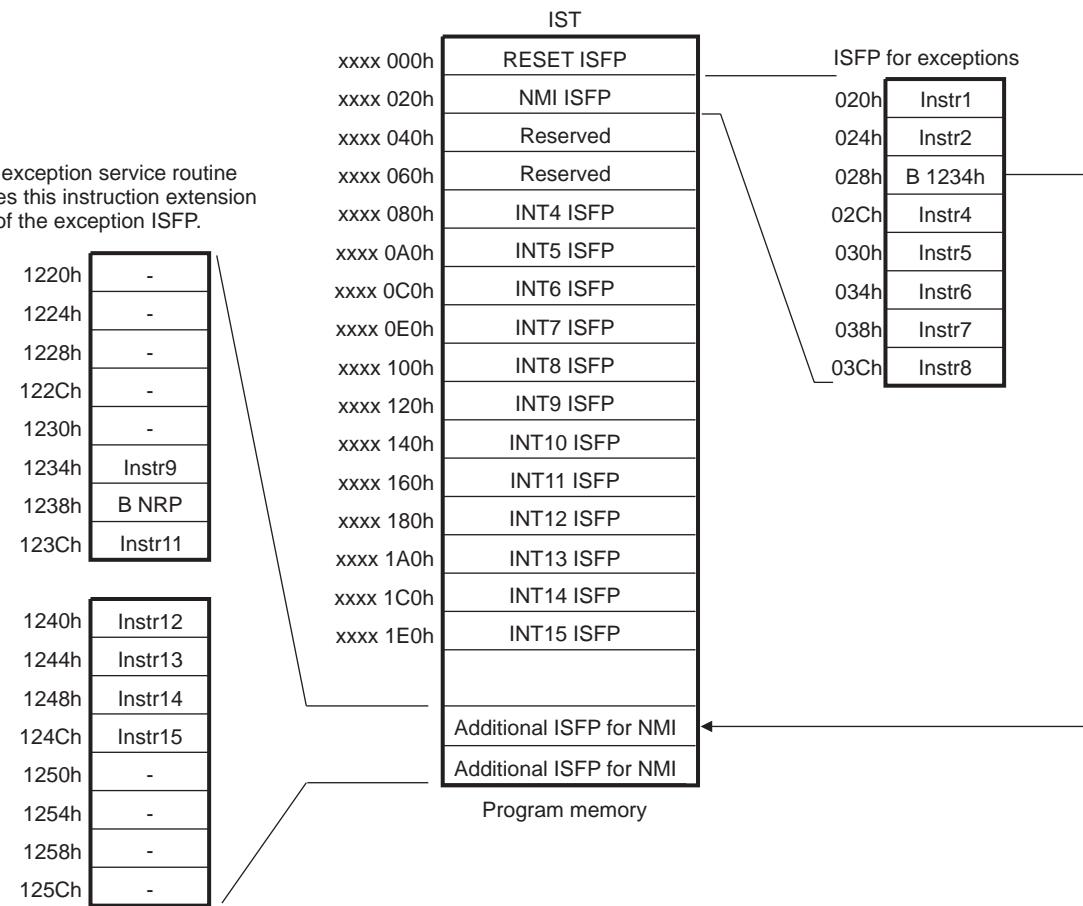
In general, the exception service routine for an exception is too large to fit in a single fetch packet, so a branch to the location of additional exception service routine code is required. This is shown in [Figure 7-1](#).

### 7.1.3 Summary of Exception Control Registers

Table 7-1 lists the control registers related to exceptions on the CPU.

**Table 7-1 Exception-Related Control Registers**

Acronym	Register Name	Description	Section
ECR	Exception clear register	Used to clear pending exception flags	2.9.2 "Exception Clear Register (ECR)" on page 2-24
EFR	Exception flag register	Contains pending exception flags	2.9.3 "Exception Flag Register (EFR)" on page 2-25
IER	Interrupt enable register	Contains NMI exception enable (NMIE) bit	2.8.7 "Interrupt Enable Register (IER)" on page 2-19
IERR	Internal exception report register	Indicates cause of an internal exception	2.9.6 "Internal Exception Report Register (IERR)" on page 2-27
ISTP	Interrupt service table pointer register	Pointer to the beginning of the interrupt service table that contains the exception interrupt service fetch packet	2.8.11 "Interrupt Service Table Pointer Register (ISTP)" on page 2-22
NRP	Nonmaskable interrupt return pointer register	Contains the return address used on return from an exception. This return is accomplished via the <b>B NRP</b> instruction	2.8.12 "Nonmaskable Interrupt (NMI) Return Pointer Register (NRP)" on page 2-23
NTSR	Nonmaskable interrupt/exception task state register	Stores contents of TSR upon taking an exception	2.9.9 "NMI/Exception Task State Register (NTSR)" on page 2-29
REP	Restricted entry point address register	Contains the address to where the <b>SWENR</b> instruction transfers control	2.9.10 "Restricted Entry Point Register (REP)" on page 2-29
TSR	Task state register	Contains global exception enable (GEE) and exception enable (XEN) bits	2.9.14 "Task State Register (TSR)" on page 2-33

**Figure 7-1** Interrupt Service Table With Branch to Additional Exception Service Code Located Outside the IST


## 7.2 Exception Control

Enabling and disabling individual exceptions is done with the task state register (TSR) and the interrupt enable register (IER). The status of pending exceptions is stored in the exception flag register (EFR). The nonmaskable interrupt return pointer register (NRP) and the nonmaskable interrupt/exception task state register (NTSR) are used to restore context after servicing exceptions. In many cases it is not possible to return to the interrupted code since exceptions can be taken at noninterruptible points.

### 7.2.1 Enabling and Disabling External Exceptions

Exceptions are globally enabled by the GEE bit in TSR. This bit must be set to 1 to enable any exception processing. Once it is set to 1, the GEE bit can only be cleared by a reset. The GEE bit is the only enable for internal exceptions. Therefore, once internal exceptions have been enabled they cannot be disabled. Global enabling of exceptions also causes the NMI input to be treated as an exception rather than an interrupt.

External exceptions are also qualified by the NMIE bit in IER. An external exception (EXCEP or NMI) can trigger exception processing only if this bit is set. Internal exceptions are not affected by NMIE. The IER is shown in [Figure 2-7](#) and described in [Table 2-12](#) on page 2-19. The EXCEP exception input can also be disabled by clearing the XEN bit in TSR.

When NMIE = 0, all interrupts and external exceptions are disabled, preventing interruption of an exception service routine. The NMIE bit is cleared at reset to prevent any interruption of processor initialization until you enable exceptions. After reset, you must set the NMIE bit to enable external exceptions and to allow INT15-INT4 to be enabled by the GIE bit and the appropriate IER bit. You cannot manually clear the NMIE bit; the NMIE bit is unaffected by a write of 0. The NMIE bit is also cleared by the occurrence of an NMI. If cleared, the NMIE bit is set only by completing a **B NRP** instruction or by a write of 1 to NMIE.

### 7.2.2 Pending Exceptions

EFR contains four bits that indicate which exceptions have been detected. It is possible for all four bits to be set when entering the exception service routine. The prioritization and handling of multiple exceptions is left to software. Clearing of the exception flags is done by writing a 1 to the bit position to be cleared in the exception clear register (ECR). Bits that are written as 0 to ECR have no effect. The EFR is shown in [Figure 2-15](#) and described in [Table 2-17](#) on page 2-25.

### 7.2.3 Exception Event Context Saving

TSR contains the CPU execution context that is saved into NTSR at the start of exception processing. TSR is then set to indicate the transition to supervisor mode and that exception processing is active. The TSR is shown in [Figure 2-26](#) and described in [Table 2-21](#) on page 2-33.

Execution of a **B NRP** instruction causes the saved context in NTSR to be loaded into TSR to resume execution. Similarly, a **B IRP** instruction restores context from ITSR into TSR.

Information about the CPU context at the point of an exception is retained in NTSR. [Table 7-2](#) shows the behavior for each bit in NTSR. The information in NTSR is used upon execution of a **B NRP** instruction to restore the CPU context before resuming the interrupted instruction execution. The HWE bit in NTSR is set when an internal or external exception is taken. The HWE bit is cleared by the **SWE** and **SWENR** instructions. The NTSR is shown in [Figure 2-21](#) and described in [Table 2-19](#) on page 2-29.

**Table 7-2 NTSR Field Behavior When an Exception is Taken**

Bit	Field	Action
0	GIE	GIE bit in TSR at point exception is taken.
1	SGIE	SGIE bit in TSR at point exception is taken.
2	GEE	GEE bit in TSR at point exception is taken (must be 1).
3	XEN	XEN bit in TSR at point exception is taken.
7-6	CXM	CXM bits in TSR at point exception is taken.
9	INT	INT bit in TSR at point exception is taken.
10	EXC	EXC bit in TSR at point exception is taken (must be 0).
14	SPLX	Terminated an SPLOOP
15	IB	Exception occurred while interrupts were blocked.
16	HWE	Hardware exception taken (NMI, EXCEP, or internal).

### 7.2.4 Returning From Exception Servicing

The NMI return pointer register (NRP) stores the return address used by the CPU to resume correct program execution after NMI processing. A branch using the address in the NRP (**B NRP**) in your exception service routine causes the program to exit the exception service routine and return to normal program execution.

It is not always possible to safely exit the exception handling routine. Conditions that can prevent a safe return from exceptions include:

- SPLOOPs that are terminated by an exception cannot be resumed correctly. The SPLX bit in NTSR should be verified to be 0 before returning.
- Exceptions that occur when interrupts are blocked cannot be resumed correctly. The IB bit in NTSR should be verified to be 0 before returning.
- Exceptions that occur at any point in the code that cannot be interrupted safely (for example, a tight loop containing multiple assignments) cannot be safely returned to. The compiler will normally disable interrupts at these points in the program; check the GIE bit in NTSR to be 1 to verify that this condition is met.

**Example 7-1** shows code that checks these conditions.

If the exception cannot be safely returned from, the appropriate response will be different based on the specific cause of the exception. In some cases, a warm reset will be required. In other cases, restarting a user task may be sufficient.

The NRP contains the 32-bit address of the first execute packet in the program flow that was not executed because of an exception. Although you can write a value to this register, any subsequent exception processing may overwrite that value. The NRP is shown in [Figure 2-12](#) on page 2-23.

#### Example 7-1 Code to Return From Exception

```

-----+
      STNDW B1:B0,*SP--          ; save B0 and B1
      ||    MVC     NTSR,B0        ; read NTSR
      ||    EXTU   B0,16,30,B1    ; B1 = NTSR.IB and NTSR.SPLX
      ||    AND    B0,1,B0        ; B0 = NTSR.GIE
[B1]   MVK     0,B0            ; B0 = 1 if resumable
[B0]   B      NRP             ; if B0 != 0, return
      LDNDW   *SP++,B1:B0       ; restore B0 and B1
      NOP     4                ; delay slots
cant_restart:
      code to handle non-resumable case starts here
-----+

```

**End of Example 7-1**

## 7.3 Exception Detection and Processing

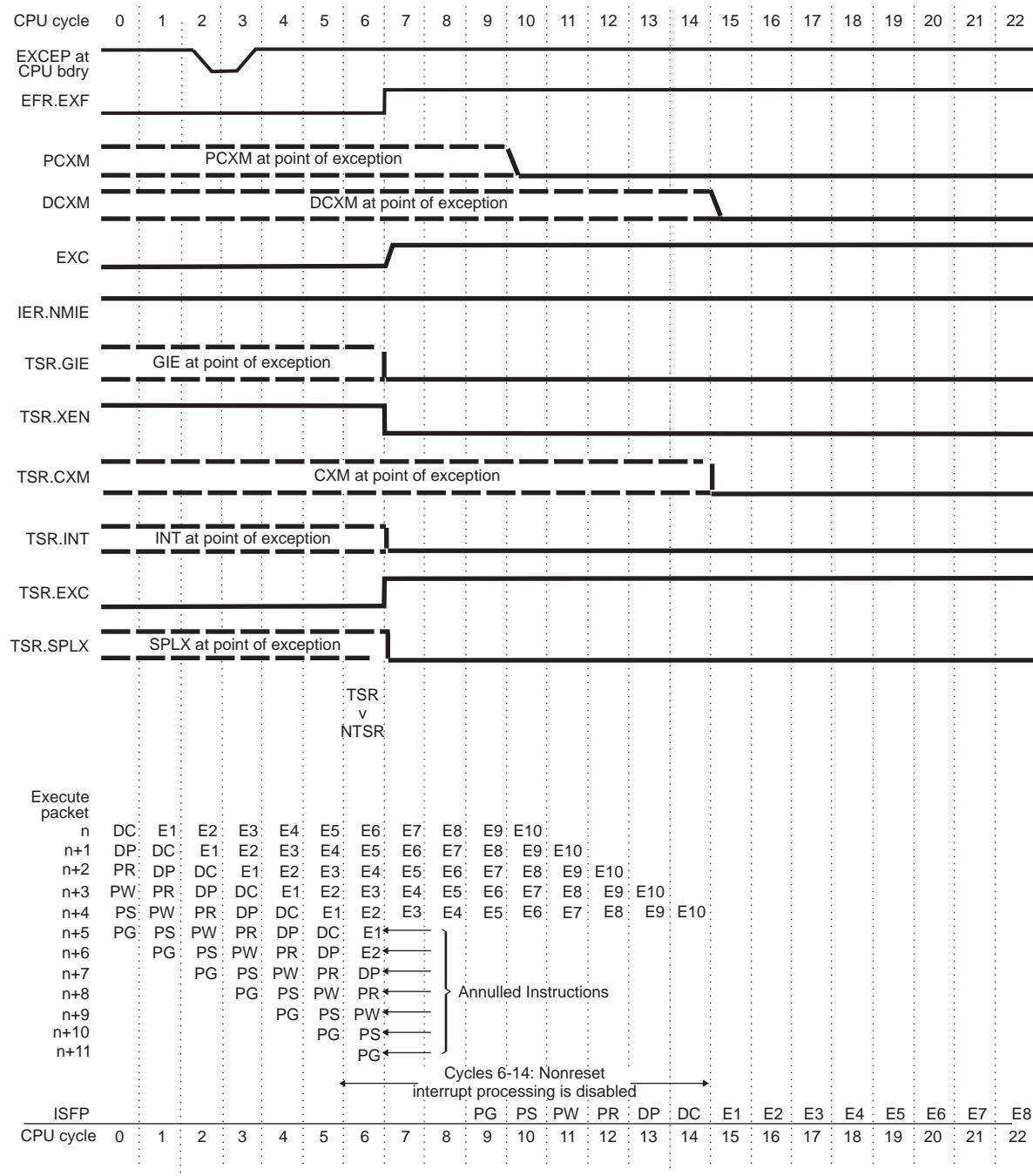
When an exception occurs, it sets a flag in the exception flag register (EFR). Depending on certain conditions, the exception may or may not be processed. This section discusses the mechanics of setting the flag bit, the conditions for processing an exception, and the order of operation for detecting and processing an exception.

### 7.3.1 Setting the Exception Pending Flag

[Figure 7-2](#) shows the processing of an external exception (EXCEP) for the CPU. The internal pending flag for EXCEP is set following the low-to-high transition of the EXCEP signal on the CPU boundary. This transition is detected on a clock-cycle by clock-cycle basis and is not affected by memory stalls that might extend a CPU cycle. Two clock cycles after detection, the EXCEP exception's corresponding flag bit (EXF) in EFR is set (cycle 6).

Figure 7-2 assumes EXCEP is enabled by the XEN, NMIE, and GEE bits, as necessary.

**Figure 7-2 External Exception (EXCEP) Detection and Processing: Pipeline Operation**



### 7.3.2 Conditions for Processing an External Exception

In clock cycle 4 of Figure 7-2, a nonreset exception in need of processing is detected. For this exception to be processed, the following conditions must be valid on the same clock cycle and are evaluated every clock cycle:

- EXF or NXF bit is set during CPU cycle 6. (This determination is made in CPU cycle 4 by the exception/interrupt logic.)

- NMIE = 1
- GEE = 1
- For EXCEP, XEN = 1

Any pending exception will be taken as soon as any stalls are completed.

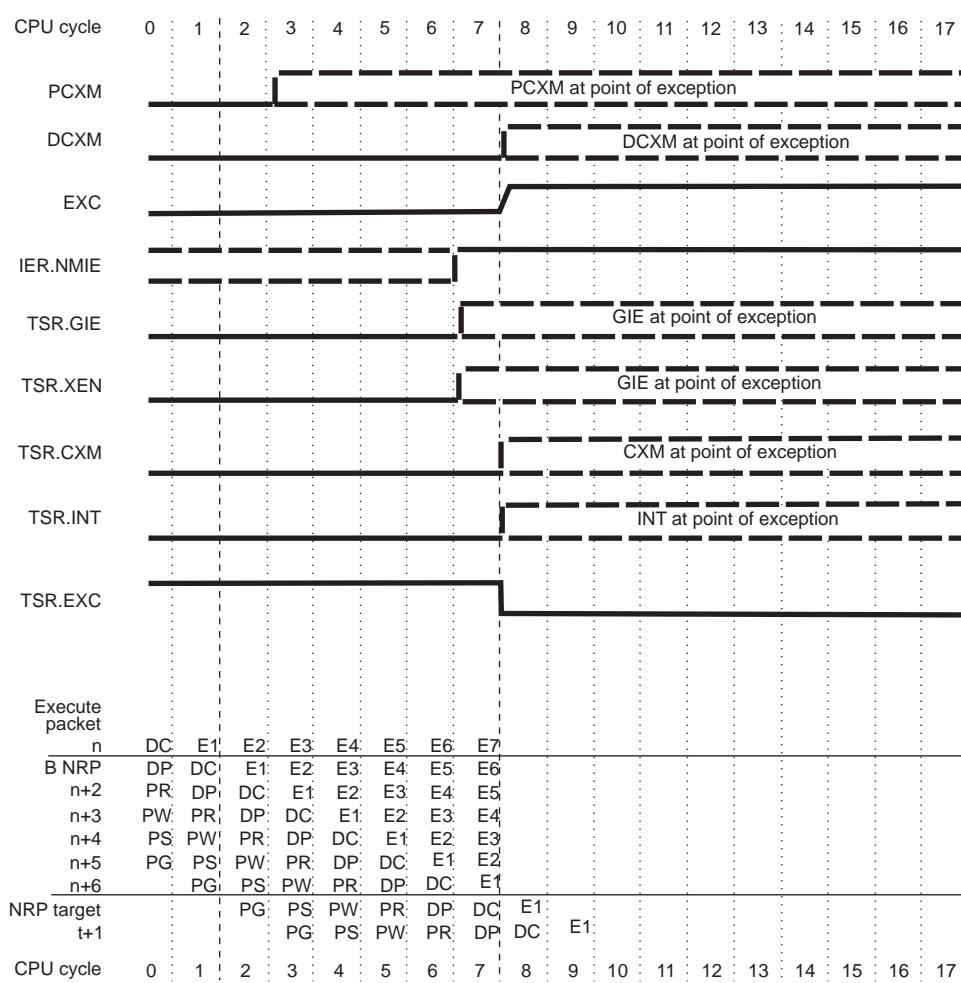
When control is transferred to the interrupt processing sequence the context needed to return from the ISR is saved in ITSR. TSR is set for the default interrupt processing context. [Table 7-3](#) shows the behavior for each bit in TSR. [Figure 7-2](#) shows the timing of the changes to the TSR bits as well as the CPU outputs used in exception processing.

Fetches from program memory use the PS-valid register that is only loaded at the start of a context switch. This value is an output on the program memory interface and is shown in the timing diagram as PCXM. As the target execute packet progresses through the pipeline, the new mode is registered for that stage. Each stage uses its registered version of the execution mode. The field in TSR is the E1-valid version of CXM. It always indicates the execution mode for the instructions executing in E1. The mode is used in the data memory interface, and is registered for all load/store instructions when they execute in E1. This is shown in the timing diagram as DCXM.

[Figure 7-3](#) shows the transitions in the case of a return from exception initiated by executing a **B NRP** instruction.

**Table 7-3 TSR Field Behavior When an Exception is Taken (EXC = 0)**

Bit	Field	Action
0	GIE	Saved to GIE bit in NTSR. Cleared to 0.
1	SGIE	Saved to SGIE bit in NTSR. Cleared to 0.
2	GEE	Saved to GEE bit in NTSR (will be 1). Unchanged.
3	XEN	Saved to XEN bit in NTSR. Cleared to 0.
7-6	CXM	Saved to CXM bits in NTSR. Set to Supervisor mode.
9	INT	Saved to INT bit in NTSR. Cleared to 0.
10	EXC	Saved to EXC bit in NTSR. Set to 1.
14	SPLX	Saved to SPLX bit in NTSR. Cleared to 0.
15	IB	Saved to IB bit in NTSR. Set by CPU control logic.

**Figure 7-3 Return from Exception Processing: Pipeline Operation**

### 7.3.3 Actions Taken During External Exception (EXCEP) Processing

During CPU cycles 6-14 of Figure 7-2, the following exception processing actions occur:

- Processing of subsequent EXCEP exceptions is disabled by clearing the XEN bit in TSR.
- Processing of interrupts is disabled by clearing the GIE bit in TSR.
- The next execute packets (from n + 5 on) are annulled. If an execute packet is annulled during a particular pipeline stage, it does not modify any CPU state. Annulling also forces an instruction to be annulled in future pipeline stages.
- The address of the first annulled execute packet (n + 5) is loaded into NRP.
- A branch to the NMI ISFP is forced into the E1 phase of the pipeline during cycle 9.
- During cycle 7, IACK and EXC are asserted to indicate the exception is being processed. INUM is also valid in this cycle with a value of 1.

### 7.3.4 Nested Exceptions

When the CPU enters an exception service routine, the EXC bit in TSR is set to indicate an exception is being processed. If a new exception is recognized while this bit is set, then the reset vector is used when redirecting program execution to service the second exception. In this case, NTSR and NRP are left unchanged. TSR is copied to ITSR and the current PC is copied to IRP. TSR is set to the default exception processing value and the NMIE bit in IER is cleared in this case preventing any further external exceptions.

The NTSR, ITSR, IRP, and the NRP can be tested in the users boot code to determine if reset pin initiated reset or a reset caused by a nested exception.

## 7.4 Performance Considerations

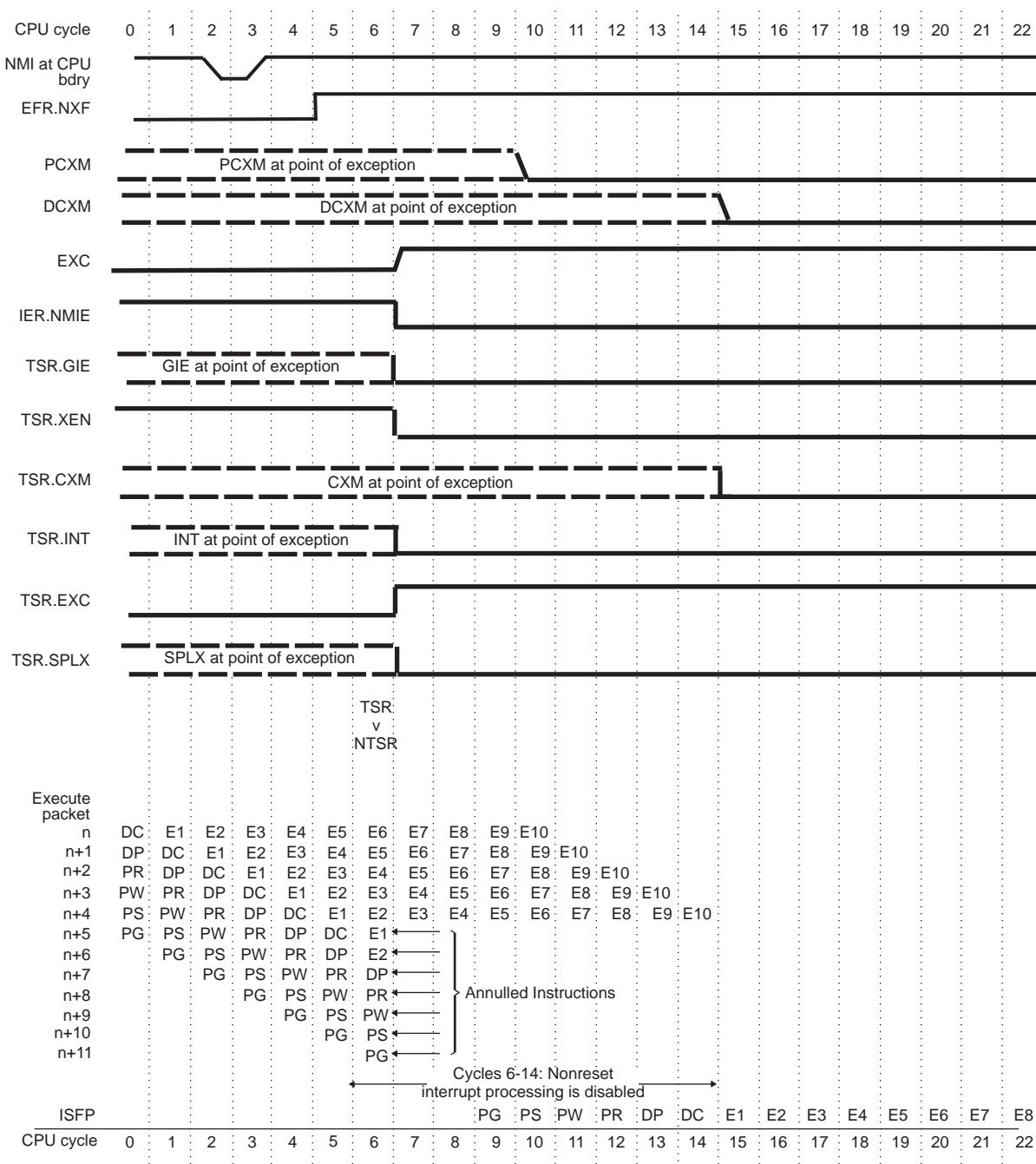
### 7.4.1 General Performance

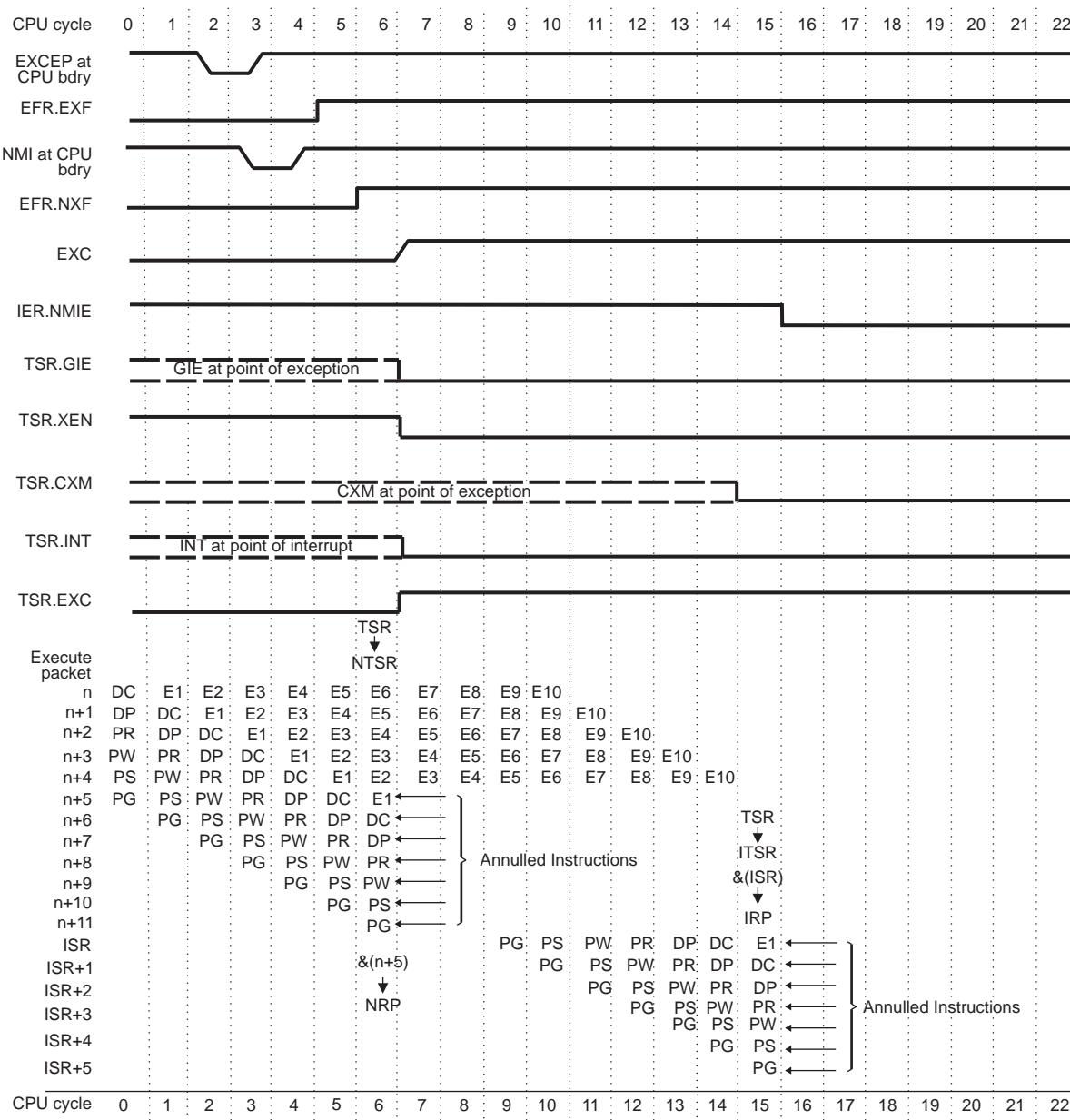
- **Overhead.** Overhead for all CPU exceptions on the CPU is 9 cycles. You can see this in [Figure 7-2](#), where no new instructions are entering the E1 pipeline phase during CPU cycles 6 through 14.
- **Latency.** Exception latency is 13 cycles. If the exception is active in cycle 2, execution of exception service code does not begin until cycle 15.
- **Frequency.** The pending exceptions are not automatically cleared upon servicing as is the case with interrupts.

### 7.4.2 Pipeline Interaction

Because the serial or parallel encoding of fetch packets does not affect the DC and subsequent phases of the pipeline, no conflicts between code parallelism and exceptions exist. There are two operations or conditions that can affect, or are affected by, exceptions:

- **Memory stalls.** Memory stalls delay exception processing, because they inherently extend CPU cycles.
- **Multicycle NOPs.** Multicycle NOPs (including the IDLE instruction) operate like other instructions when interrupted by an exception, except when an exception causes annulment of any but the first cycle of a multicycle NOP. In that case, the address of the next execute packet in the pipeline is saved in NRP. This prevents returning to an IDLE instruction or a multicycle NOP that was interrupted.

**Figure 7-4** NMI Exception Detection and Processing: Pipeline Operation

**Figure 7-5 Double Exception Detection and Processing: Pipeline Operation**

## 7.5 Programming Considerations

There are two types of exceptions that can result directly from instruction execution. The first is an intentional use via the **SWE** or **SWENR** instructions. The second is execution error detection exceptions that are internally generated within the CPU. These internal exceptions are primarily intended to facilitate program debug.

### 7.5.1 Internal Exceptions

Causes of internal exceptions:

- Fetch error
  - Program memory fetch error (privilege, parity, etc.)
- Single input from L1P returned with data indicates error
  - Two branches taken in same execute packet
  - Branch to middle of 32-bit instruction in header-based fetch packet
  - Branch to header
- Illegal fetch packets
  - Reserved fetch packet header
- Illegal opcode
  - Specified set of reserved opcodes
  - Header not in word 7
- Privilege violation
  - Access to restricted control register
  - Attempt to execute restricted instruction
- Register write conflicts
- Loop buffer exceptions (SPLOOP, SPKERNEL)
  - Unit conflicts
  - Missed (but required) stall
  - Attempt to enter early-exit in reload while draining
- Unexpected SPKERNEL
  - Write to ILC or RILC in prohibited timing window
  - Multicycle NOP prior to SPKERNEL or SPKERNELR instruction

### 7.5.2 Internal Exception Report Register (IERR)

The internal exception report register (IERR) contains flags that indicate the cause of the internal exception. In the case of simultaneous internal exceptions, the same flag may be set by different exception sources. In this case, it may not be possible to determine the exact causes of the individual exceptions. The IERR is shown in [Figure 2-18](#) and described in [Table 2-18](#) on page 2-27.

## 7.5.3 Software Exception

### 7.5.3.1 SWE Instruction

When an **SWE** instruction is executed, the SXF bit in EFR is set. On the following cycle, the exception detection logic sees the SXF bit in EFR as a 1, and takes an exception. This instruction can be used to effect a system call while running in User mode. Execution of the instruction results in transfer of control to the exception service routine operating in Supervisor mode. If the SXF bit is the only bit set in EFR, then the exception can be interpreted as a system service request. An appropriate calling convention using the general-purpose registers may be adopted to provide parameters for the call. This is left as a programming choice. An example of code to quickly detect the system call case at the beginning of the exception service routine is shown in [Example 7-2](#). Since other exceptions are in general error conditions and interrupt program execution at nonreturnable points, the need to process these is not particularly time critical.

#### Example 7-2 Code to Quickly Detect OS Service Request

```

-----+
      STW      B0,*SP--      ; save B0
||      MVC      EFR,B0      ; read EFR
      CMPEQ   B0,1,B0      ; is SEF the only exception?
[B0]    B       OS_Service ; if so,
[B0]    ...          ; conditionally execute service
[B0]    ...          ; code until branch takes effect
-----+

```

**End of Example 7-2**

### 7.5.3.2 SWENR Instruction

The **SWENR** instruction causes a software exception to be taken similarly to that caused by the **SWE** instruction. It is intended for use in systems supporting a secure operating mode. The **SWENR** instruction can be used as a mechanism for nonsecure programs to return to secure Supervisor mode services such as an interrupt dispatcher. It differs from the **SWE** instruction in four ways:

1. TSR is not copied into NTSR
2. No return address is placed in NRP (it stays unmodified)
3. A branch to restricted entry point control register (REP) is forced in the context switch rather than the ISTP-based exception (NMI) register.
4. The IB bit in TSR is set to 1. This is observable only in the case where another exception is recognized simultaneously.

If another exception (internal or external) is recognized simultaneously with the **SWENR**-raised exception, then the other exception(s) take priority and normal exception behavior occurs; that is, NTSR and NRP are used, execution is directed to the NMI vector. In this case, the setting of the IB bit in TSR by the **SWENR** instruction is registered in NTSR. Assuming the **SWE** or **SWENR** instruction was not placed in an execute slot where interrupts are architecturally blocked (as should always be the case), then the IB bit in NTSR will differentiate whether the simultaneous exception occurred with **SWE** or **SWENR**.

The **SWENR** instruction causes a change in control to the address contained in REP. It should have been previously initialized to a correct value by a privileged supervisor mode process.

# Software Pipelined Loop (SPLOOP) Buffer

This chapter describes the software pipelined loop (SPLOOP) buffer hardware and software mechanisms.

Under normal circumstances, the compiler/assembly optimizer will do a good job coding SPLOOPs and it will not be necessary for the programmer to hand code usage of the SPLOOP buffer. This chapter is intended to describe the functioning of the buffer hardware and the instructions that control it.

- 8.1 ["Software Pipelining" on page 8-2](#)
- 8.2 ["Software Pipelining" on page 8-3](#)
- 8.3 ["Terminology" on page 8-3](#)
- 8.4 ["SPLOOP Hardware Support" on page 8-3](#)
- 8.5 ["SPLOOP-Related Instructions" on page 8-5](#)
- 8.6 ["Basic SPLOOP Example" on page 8-9](#)
- 8.7 ["Loop Buffer" on page 8-13](#)
- 8.8 ["Execution Patterns" on page 8-18](#)
- 8.9 ["Loop Buffer Control Using the Unconditional SPLOOP\(D\) Instruction" on page 8-22](#)
- 8.10 ["Loop Buffer Control Using the SPLOOPW Instruction" on page 8-29](#)
- 8.11 ["Using the SPMASK Instruction" on page 8-31](#)
- 8.12 ["Program Memory Fetch Control" on page 8-35](#)
- 8.13 ["Interrupts" on page 8-36](#)
- 8.14 ["Branch Instructions" on page 8-39](#)
- 8.15 ["Instruction Resource Conflicts and SPMASK Operation" on page 8-40](#)
- 8.16 ["Restrictions on Crosspath Stalls" on page 8-41](#)
- 8.17 ["Restrictions on AMR-Related Stalls" on page 8-41](#)
- 8.18 ["Restrictions on Instructions Placed in the Loop Buffer" on page 8-41](#)

## 8.1 Software Pipelining

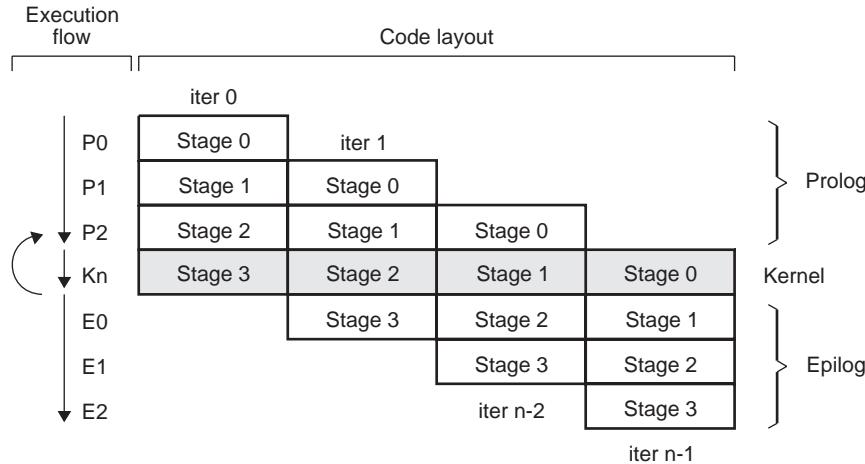
Software pipelining is a type of instruction scheduling that exploits instruction level parallelism (ILP) across loop iterations. Modulo scheduling is a form of software pipelining that initiates loop iterations at a constant rate, called the initiation interval ( $i_i$ ). To construct a modulo scheduled loop, a single loop iteration is divided into a sequence of stages, each with length  $i_i$ . In the steady state of the execution of the software pipelined loop, each of the stages is executing in parallel.

The instruction schedule for a modulo scheduled loop has three components: a kernel, a prolog, and an epilog (Figure 8-1). The kernel is the instruction schedule that executes the pipeline steady state. The prolog and epilog are the instruction schedules that setup and drain the execution of the loop kernel. In Figure 8-1, the steady state has four stages, each from a different iteration, executing in parallel. A single iteration produces a result in the time it takes four stages to complete, but in the steady state of the software pipeline, a result is available every stage (that is, every  $i_i$  cycles).

The first prolog stage,  $P_0$ , is equal to the first loop stage,  $S_0$ . Each prolog stage,  $P_n$  (where  $n > 0$ ), is made up of the loop stage,  $S_n$ , plus all the loop stages in the previous prolog stage,  $P_{n-1}$ . The kernel includes all the loop stages. The first epilog stage,  $E_0$ , is made up of the kernel stage minus the first loop stage,  $S_0$ . Each epilog stage,  $E_n$  (where  $n > 0$ ), is made up of the previous epilog stage,  $E_{n-1}$ , minus the loop stage,  $S_n$ .

The dynamic length (dynlen) of the loop is the number of instruction cycles required for one iteration of the loop to complete. The length of the prolog is ( $\text{dynlen} - i_i$ ). The length of the epilog is the same as the length of the prolog.

**Figure 8-1 Software Pipelined Execution Flow**



## 8.2 Software Pipelining

The SPLOOP facility on the DSP stores a single iteration of loop in a specialized buffer and contains hardware that will selectively overlay copies of the single iteration in a software pipeline manner to construct an optimized execution of the loop.

This provides the following benefits.

- Since the prolog and epilog do not need to be explicitly code, code size is significantly reduced.
- The SPLOOP version of the loop can be easily interrupted unlike the non-SPLOOP version of the same loop.
- Since the instructions in the loop do not need to be fetched on each cycle, the memory bandwidth and power requirements are reduced.
- Since the loop executes out of a buffer, the branch to the start of loop is implicit (hence not required). In some cases this may permit a tighter loop since a .S unit is freed.

## 8.3 Terminology

The following terminology is used in the discussion in this chapter.

- Iteration interval (ii) is the interval (in instruction cycles) between successive iterations of the loop.
- A stage is the code executed in one iteration interval.
- Dynamic length (dynlen) is the length (in instruction cycles) of a single iteration of the loop. It is therefore equal to the number of stages times the iteration interval.<sup>5</sup>
- The kernel is the period when the loop is executing in a steady state with the maximum number of loop iterations executing simultaneously. For example: in [Figure 8-1](#) the kernel is the set of instructions contained in stage 0, stage 1, stage 2, and stage 3.
- The prolog is the period before the loop reaches the kernel in which the loop is winding up. The length of the prolog will be the dynamic length minus the iteration interval (dynlen - ii).
- The epilog is the period after the loop leaves the kernel in which the loop is winding down. The length of the prolog will be the dynamic length minus the iteration interval (dynlen - ii).

## 8.4 SPLOOP Hardware Support

The basic hardware support for the SPLOOP operation is:

- Loop buffer
- Loop buffer count register (LBC)
- Inner loop count register (ILC)
- Reload inner loop count register (RILC)
- Task state register (TSR)
- Interrupt task state register (ITSR)
- NMI/Exception task state register (NTSR)

### 8.4.1 Loop Buffer

The loop buffer is used to store the instructions that comprise the loop and information describing the sequence that the instructions were added to the buffer and the state (active or inactive) of each instruction.

The loop buffer has enough storage for up to 14 execute packets.

### 8.4.2 Loop Buffer Count Register (LBC)

A loop buffer count register (LBC) is maintained as an index into the loop buffer. It is cleared to 0 when an **SPLOOP**, **SPLOOPD** or **SPLOOPW** instruction is encountered and is incremented by 1 at the end of each cycle. When LBC becomes equal to the iteration interval (ii) specified by the **SPLOOP**, **SPLOOPD** or **SPLOOPW** instruction, then a stage boundary has been reached and LBC is reset to 0 and the inner loop count register (ILC) is decremented.

There are two LBCs to support overlapped nested loops. LBC is not a user-visible register.

### 8.4.3 Inner Loop Count Register (ILC)

The inner loop count register (ILC) is used as a down counter to determine when the SPLOOP is complete when the SPLOOP is initiated by either a **SPLOOP** or **SPLOOPD** instruction. When the loop is initiated using a **SPLOOPW** instruction, the ILC is not used to determine when the SPLOOP is complete. It is decremented once each time a stage boundary is encountered; that is, whenever the loop buffer count register (LBC) becomes equal to the iteration interval (ii).

There is a 4 cycle latency between when ILC is loaded and when its contents are available for use. When used with the **SPLOOP** instruction, it should be loaded 4 cycles before the **SPLOOP** instruction is encountered. ILC must be loaded explicitly using the **MVC** instruction.

### 8.4.4 Reload Inner Loop Count Register (RILC)

The reload inner loop count register (RILC) is used for resetting the inner loop count register (ILC) for the next invocation of a nested inner loop. There is a 4 cycle latency between when RILC is loaded with the **MVC** instructions and when the value loaded to RILC is available for use. RILC must be loaded explicitly using the **MVC** instruction.

### 8.4.5 Task State Register (TSR), Interrupt Task State Register (ITSR), and NMI/Exception Task State Register (NTSR)

The SPLX bit in the task state register (TSR) indicates whether an SPLOOP is currently executing or not executing.

When an interrupt occurs, the contents of TSR (including the SPLX bit) is copied to the interrupt task state register (ITSR).

When an exception or non-maskable interrupt occurs, the contents of TSR (including the SPLX bit) is copied to the NMI/Exception task state register (NTSR).

See section [2.9.14](#) on page 2-33 for more information on TSR. See section [2.9.8](#) on page 2-28 for more information on ITSR. See [2.9.9](#) on page 2-29 for more information on NTSR.

## 8.5 SPLOOP-Related Instructions

The following instructions are used to control the operation of an SPLOOP:

- SPLOOP, SPLOOPD, and SPLOOPW
- SPKERNEL and SPKERNELR
- SPMASK and SPMASKR

### 8.5.1 SPLOOP, SPLOOPD, and SPLOOPW Instructions

One of the **SPLOOP**, **SPLOOPD**, or **SPLOOPW** (collectively called **SPLOOP(D/W)**) instructions are used to invoke the loop buffer mechanism. They each fulfil the same basic purpose, but differ in details. In each case, they must be the first instruction of the execute packet containing it. They cannot be placed in the same execute packet as any instruction that initiates a multicycle NOP (for example: BNOP or NOP $n$ ).

When you know in advance the number of iterations that the loop will execute, you can use the **SPLOOP** or **SPLOOPD** instructions. If you do not know the exact number of iterations that the loop should execute, you can use the **SPLOOPW** in a fashion similar to a do-while loop.

The **SPLOOP(D/W)** instructions each clear the loop buffer count register (LBC), load the iteration interval (ii), and start the LBC counting.

#### 8.5.1.1 SPLOOP Instruction

The **SPLOOP** instruction is coded as:

[cond] SPLOOP ii

The ii parameter is the iteration interval that specifies the interval (in instruction cycles) between successive iterations of the loop.

The **SPLOOP** instruction is used when the number of loop iterations is known in advance. The number of loop iterations is determined by the value loaded to the inner loop count register (ILC). ILC should be loaded with an initial value 4 cycles before the **SPLOOP** instruction is encountered.

The (optional) conditional predication is used to indicate when and if a nested loop should be reloaded. The contents of the reload inner loop counter (RILC) is copied to ILC when either a **SPKERNELR** or a **SPMASKR** instruction is executed with the predication condition on the **SPLOOP** instruction true. If the loop is not nested, then the conditional predication should not be used.

#### 8.5.1.2 SPLOOPD Instruction

The **SPLOOPD** instruction is coded as:

[cond] SPLOOPD ii

The ii parameter is the iteration interval which specifies the interval (in instruction cycles) between successive iterations of the loop.

The **SPLOOPD** instruction is used to initiate a loop buffer operation when the known minimum iteration count of the loop is great enough that the inner loop count register (ILC) can be loaded in parallel with the **SPLOOPD** instruction and the 4 cycle latency will have passed before the last iteration of the loop.

Unlike the **SPLOOP** instruction, the load of ILC is performed in parallel with the **SPLOOPD** instruction. Due to the inherent latency of the load to ILC, the value to ILC should be predecremented to account for the 4 cycle latency. The amount of the predecrement is given in [Table 8-3](#).

The number of loop iterations is determined by the value loaded to ILC.

The (optional) conditional predication is used to indicate when and if a nested loop should be reloaded. The contents of the reload inner loop counter (RILC) is copied to ILC when either a **SPKERNELR** or a **SPMASKR** instruction is executed with the predication condition on the **SPLOOP** instruction true. If the loop is not nested, then the conditional predication should not be used.

The use of the **SPLOOPD** instruction can result in reducing the time spent in setting up the loop by eliminating up to 4 cycles that would otherwise be spent in setting up ILC. The trade-off is that the **SPLOOPD** instruction cannot be used if the loop is not long enough to accommodate the 4 cycle delay.

#### **8.5.1.3 SPLOOPW Instruction**

The **SPLOOPW** instruction is coded as:

[cond] SPLOOPW ii

The ii parameter is the iteration interval which specifies the interval (in instruction cycles) between successive iterations of the loop.

The **SPLOOPW** instruction is used to initiate a loop buffer operation when the total number of loops required is not known in advance. The **SPLOOPW** instruction must be predicated. The loop terminates if the predication condition is true. The value in the inner loop count register (ILC) is not used to determine the number of loops.

Unlike the **SPLOOP** and **SPLOOPD** instructions, predication on the **SPLOOPW** instruction does not imply a nested SPLOOP operation. The **SPLOOPW** instruction cannot be used in a nested SPLOOP operation.

When using the **SPLOOPW** instruction, the predication condition is used to determine the exit condition for the loop. The ILC is not used for this purpose when using the **SPLOOPW** instruction.

When the **SPLOOPW** instruction is used to initiate a loop buffer operation, the epilog is skipped when the loop terminates.

#### **8.5.2 SPKERNEL and SPKERNELR Instructions**

The **SPKERNEL** or the **SPKERNELR** (collectively called **SPKERNEL(R)**) instruction is used to mark the end of the software pipelined loop. The **SPKERNEL(R)** instruction is placed in parallel with the last execute packet of the SPLOOP code body indicating that there are no more instructions to load to the loop buffer.

The **SPKERNEL(R)** instruction also controls the point in the epilog that the execution of post-SPLOOP instructions begin.

In each case, the **SPKERNEL(R)** instruction must be the first instruction in an execute packet and cannot be placed in the same execute packet as any instruction that initiates multicycle NOPs.

### 8.5.2.1 SPKERNEL Instruction

The **SPKERNEL** instruction is coded as:

```
SPKERNEL (fstg, fcyc)
```

The (optional) *fstg* and *fcyc* parameters specify the delay interval between the **SPKERNEL** instruction and the start of the post epilog code. The *fstg* specifies the number of complete stages and the *fcyc* specifies the number of cycles in the last stage in the delay.

The **SPKERNEL** instruction has arguments that instruct the SPLOOP hardware to begin execution of post-SPLOOP instructions by an amount of delay (stages/cycles) after the start of the epilog.

Note that the post-epilog instructions are fetched from program memory and overlaid with the epilog instructions fetched from the SPLOOP buffer. Functional unit conflicts can be avoided by either coding for a sufficient delay using the **SPKERNEL** instruction arguments or by using the **SPMASK** instruction to inhibit the operation of instructions from the buffer that might conflict with the instructions from the epilog.

### 8.5.2.2 SPKERNELR Instruction

The **SPKERNELR** instruction is coded as:

```
SPKERNELR
```

The **SPKERNELR** instruction is used to support nested SPLOOP execution where a loop needs to be restarted with perfect overlap of the prolog of the second loop with the epilog of the first loop.

If a reload is required with a delay between the **SPKERNEL** and the point of reload (that is, nonperfect overlap) use the **SPMASKR** instruction with the **SPKERNEL** (not **SPKERNELR**) to indicate the point of reload.

The **SPKERNELR** instruction has no arguments. The execution of post-SPLOOP instructions commences simultaneous with the first cycle of epilog. If the predication of the **SPLOOP** instruction indicates that the loop is being reloaded, the instructions are fetched from both the SPLOOP buffer and of program memory.

The **SPKERNELR** instruction cannot be used in the same SPLOOP operation as the **SPMASKR** instruction.

### 8.5.3 SPMASK and SPMASKR Instructions

The **SPMASK** and **SPMASKR** (collectively called **SPMASK(R)**) instructions are used to inhibit the operation of instructions on specified functional units within the current execute packet.

- If there is an instruction from the buffer that would utilize the specified functional unit in the current cycle, the execution of that instruction is inhibited.
- If the buffer is in the loading stage and there is an instruction (regardless of functional unit) that is scheduled for execution during that cycle, the execution of that instruction proceeds, but the instruction is not loaded into the buffer.
- If the case where an **SPMASK(R)** instruction is encountered while the loop is resuming after returning from an interrupt, the **SPMASK(R)** instruction causes the instructions coming from the buffer to execute, but instructions coming from program memory to be inhibited and are not loaded to the buffer.

The **SPMASKR** instruction is identical to the function of the **SPMASK** instruction with one additional operation. In the case of nested loops where it is not desired that the reload of the buffer happen immediately after the **SPKERNEL** instruction, the **SPMASKR** instruction can be used to mark the point in the epilog that the reload should begin.

The **SPMASKR** instruction cannot be used in the same SPLOOP operation as the **SPKERNEL** instruction.

The **SPMASK** and **SPMASKR** instructions are coded as:

```
SPMASK (unitmask)
SPMASKR (unitmask)
```

The unitmask parameter specifies which functional units are masked by the **SPMASK** or **SPMASKR** instruction. The units may alternatively be specified by marking the instructions with a caret (^) symbol. The following two forms are equivalent and will each mask the .D1 unit. [Example 8-1](#) and [Example 8-2](#) show the two ways of specifying the masked instructions.

#### **Example 8-1 SPMASK Using Unit Mask to Indicate Masked Unit**

```
-----  

||| SPMASK D1 ;Mask .D1 unit  

||| LDW .D1 *A0,A1 ;This instruction is masked  

||| MV .L1 A2,A3 ;This instruction is NOT masked
```

**End of Example 8-1**

#### **Example 8-2 SPMASK Using Caret to Indicate Masked Unit**

```
-----  

||^ SPMASK D1 ;Mask .D1 unit  

||^ LDW .D1 *A0,A1 ;This instruction is masked  

||^ MV .L1 A2,A3 ;This instruction is NOT masked
```

**End of Example 8-2**

## 8.6 Basic SPLOOP Example

This section discusses a simple SPLOOP example. [Example 8-3](#) shows an example of a loop coded in C and [Example 8-4](#) shows an implementation of the same loop using the **SPLOOP** instruction. The loop copies a number of words from one location in memory to another.

### Example 8-3 Copy Loop Coded as C Fragment

```
-----  
for (I=0; i<val; I++) {  
    dest[i]=source[i];  
}
```

**End of Example 8-3**

### Example 8-4 SPLOOP Implementation of Copy Loop

```
-----  
MVC      8,ILC          ;Do 8 loops  
NOP      3              ;4 cycle for ILC to load  
SPLOOP   1              ;Iteration interval is 1  
LDW      *A1++,A2        ;Load source  
NOP      4              ;Wait for source to load  
MV       .L1X  A2,B2     ;Position data for write  
SPKERNEL 6,0            ;End loop and store value  
||       STW      B2,*B0++
```

**End of Example 8-4**

[Example 8-5](#) is an alternate implementation of the same loop using the **SPLOOPD** instruction. The load of the inner loop count register (ILC) can be made in the same cycle as the **SPLOOPD** instruction, but due to the inherent delay between loading the ILC and its use, the value needs to be predecremented to account for the 4 cycle delay.

### Example 8-5 SPLOOPD Implementation of Copy Loop

```
-----  
||       SPLOOPD  1          ;Iteration interval is 1  
         MVC      8-4,ILC      ;Do 8 iterations  
         LDW      *A1++,A2      ;Load source  
         NOP      4              ;Wait for source to load  
         MV       .L1X  A2,B2     ;Position data for write  
         SPKERNEL 6,0            ;End loop and store value  
||       STW      B2,*B0++
```

**End of Example 8-5**

**Table 8-1** SPLOOP Instruction Flow for [Example 8-4](#) and [Example 8-5](#)

Cycle	Loop							
	1	2	3	4	5	6	7	8
1	LDW							
2	NOP	LDW						
3	NOP	NOP	LDW					
4	NOP	NOP	NOP	LDW				
5	NOP	NOP	NOP	NOP	LDW			
6	MV	NOP	NOP	NOP	NOP	LDW		
7	STW	MV	NOP	NOP	NOP	NOP	LDW	
8		STW	MV	NOP	NOP	NOP	NOP	LDW
9			STW	MV	NOP	NOP	NOP	NOP
10				STW	MV	NOP	NOP	NOP
11					STW	MV	NOP	NOP
12						STW	MV	NOP
13							STW	MV
14								STW

### 8.6.1 Some Points About the Basic SPLOOP Example

Note the following points about [Example 8-4](#), [Example 8-5](#), and [Table 8-1](#).

- In [Example 8-4](#), due to the 4 cycle latency of loading ILC, the load to ILC happens at least 4 cycles before the **SPLOOP** instruction is encountered. In this case, the **MVC** instruction that loads ILC is followed by 3 cycles of NOP.
- In [Example 8-5](#), the use of the **SPLOOPD** instruction allows you to load ILC in the same cycle of the **SPLOOPD** instruction; but the value loaded to ILC is adjusted to account for the inherent 4 cycle delay between loading the ILC and its use.
- The iteration interval (ii) is specified in the argument of the SPLOOP instruction.
- The termination condition (ILC equal to 0) is tested at every stage boundary. In this example, with ii equal to 1, it is tested at each instruction cycle. Once the termination condition is true, the loop starts draining.
- Cycles 1 through 6 constitute the prolog. Until cycle 7, the pipeline is filling.
- Cycles 7 and 8 each constitute a single iteration of the kernel. During each of these cycles, the pipeline is filled as full as it is going to be.
- Cycles 9 through 14 constitute the epilog. During this interval, the pipeline is draining.
- In this example, the iteration interval is 1. A new iteration of the loop is started each cycle.
- The dynamic length (dynlen) is 7. One cycle for the **LDW** instruction. One cycle for the **MV** instruction. One instruction for the **SPKERNEL** and **STW** instructions executed in parallel. Four cycles of NOP.
- The length of the prolog is (dynlen - ii) = 6 cycles. The length of the epilog is equal to the length of the prolog.
- There is no branch back to the start of the loop. The instructions are executed from the SPLOOP buffer and the **SPKERNEL** instruction marks the point that the execution is reset to the start of the buffer.

- The body of the SPLOOP is a single scheduled iteration without pipeline optimization. The execution of the SPLOOP overlays the execution of the instructions to optimize the execution of the loop.
- The argument in the **SPKERNEL** instruction indicates that the post-epilog code is delayed until after the epilog (6 cycles) completes.
- The **MV** instruction needs to be there to move the data between the A side and the B side. If it were not there, there would eventually be a unit conflict between the **LDW** and **STW** instructions as they try to execute in the same cycle.

### 8.6.2 Same Example Using the SPLOOPW Instruction

For completeness, [Example 8-6](#) shows an example of a loop coded in C and [Example 8-7](#) is the same example using the **SPLOOPW** instruction. The **SPLOOPW** instruction is intended to support while-loop constructions in which the number of loops is not known in advance and (in general) is determined as the loop progresses.

#### Example 8-6 Example C Coded Loop

```
do      {
    I--;
    dest[i]=source[i];
} while (I);
```

**End of Example 8-6**

#### Example 8-7 SPLOOPW Implementation of C Coded Loop

```

[ !A0]   MVK      8,A0          ;Do 8 loops
          SPLOOPW  1           ;Check loop
          LDW      .D1 *A1++,A2  ;Load source value
          NOP      1
          SUB      .S1 A0,1,A0  ;Adjust loop counter
          NOP      2           ;Wait for source to load
          MV       .L2X A2,B2  ;Position data for write
          SPKERNEL 0,0          ;End loop
          ||        STW      .D2 B2,*B0++  ;Store value

```

**End of Example 8-7**

**Table 8-2 SPLOOPW Instruction Flow for Example 8-7**

Cycle	Loop											
	1	2	3	4	5	6	7	8	9	10	11	12
1	LDW											
2	NOP	LDW										
3	SUB	NOP	LDW									
4	NOP	SUB	NOP	LDW								
5	NOP	NOP	SUB	NOP	LDW							
6	MV	NOP	NOP	SUB	NOP	LDW						
7	STW	MV	NOP	NOP	SUB	NOP	LDW					
8		STW	MV	NOP	NOP	SUB	NOP	LDW				
9			STW	MV	NOP	NOP	SUB	NOP	LDW			
10				STW	MV	NOP	NOP	SUB	NOP	LDW		
11					STW	MV	NOP	NOP	SUB	NOP	LDW	
12						STW	MV	NOP	NOP	SUB	NOP	LDW

### 8.6.3 Some Points About the SPLOOPW Example

Note the following points about [Example 8-7](#) and [Table 8-2](#).

- Unlike the **SPLOOP** and **SPLOOPD** instructions, the number of loops does not depend on ILC. It depends, instead, on the value in the predication register used with the **SPLOOPW** instruction (in this case, the value in A0).
- The termination condition (A0 equal to 0) is tested at every stage boundary. In this example, with ii equal to 1, it is tested at each instruction cycle. Once the termination condition is true, the loop terminates abruptly without executing the epilog.
- The termination condition (A0 equal to 0) needs to be true 3 cycles before you actually test it, so the value of A0 needs to be adjusted 4 cycles before the **SPKERNEL** instruction. In this case, the SUB instruction was positioned using the **NOP** instructions so that its result would be available 3 cycles before the **SPKERNEL**.
- Unlike the **SPLOOP** and **SPLOOPD** instructions, the **SPLOOPW** instruction causes the loop to exit without an epilog. In cycle 13, the loop terminates abruptly.
- Loop 9 through loop 14 begin, but they do not finish. The loop exits abruptly on the stage boundary 3 cycles after the termination condition becomes true. It is important that the loop is coded so that the extra iterations of the early instructions do not cause problems by overwriting significant locations. For example: if the loop contains an early write to a buffer we might find that in incorrectly coded loop might write beyond the end of the buffer overwriting data unintentionally.

## 8.7 Loop Buffer

The basic facility to support software pipelined loops is the loop buffer. The loop buffer has storage for up to 14 execute packets. The buffer is filled from the SPLOOP body that follows an **SPLOOP(D/W)** instruction and ends with the first execute packet containing an **SPKERNEL** instruction.

The SPLOOP body is a single, scheduled iteration of the loop. It consists of one or more stages of ii cycles each. The execution of the prolog, kernel, and epilog are generated from copies of this single iteration time shifted by multiples of ii cycles and overlapped for simultaneous execution. The final stage may contain fewer than ii cycles, omitting the final cycles if they have only **NOP** instructions.

The dynamic length (dynlen) is the length of the SPLOOP body in cycles starting with the cycle after the **SPLOOP(D)** instruction. The dynamic length counts both execute packets and NOP cycles, but does not count stall cycles. The loop buffer can accommodate a SPLOOP body of up to 48 cycles.

**Example 8-8** demonstrates counting of dynamic length. There are 4 cycles of NOP that could be combined into a single **NOP 4**. It is split up here to be clearer about the cycle and stage boundaries.

### Example 8-8 SPLOOP, SPLOOP Body, and SPKERNEL

```

-----+
| | SPLOOP    2          ;ii=2, dynlen=7
| |   MV        A0,A1      ;save previous cond reg
;----Start of stage 0
| |   LDW        *B7[A0],A5    ;cycle 1
| |   NOP        2          ;cycle 2
;----stage boundary. End of stage 0, start of stage 1
| |   NOP        2          ;cycles 3 and 4
;----stage boundary. End of stage 1, start of stage 2
| |   HOP        2          ;cycle 5
| |   EXTU       A5,12,7,A6  ;cycle 6
;----stage boundary. End of stage 2, start of stage 3
| |   SPKERNEL   0,0        ;last exe pkt of SPLOOP body
| |   ADD        .D1        A6,A7,A7  ;accumulate (cycle 7)
| |   NOP        2          ;can omit final NOP of last
;----stage boundary. End of stage 3
-----+

```

**End of Example 8-8**

### 8.7.1 Software Pipeline Execution From the Loop Buffer

The loop buffer is the mechanism that both generates the execution of the loop prolog, kernel, and epilog, and saves the repeated fetching and decoding of instructions in the loop. As the SPLOOP body is fetched and executed the first time, it is loaded into the loop buffer. By the time the entire SPLOOP body has been loaded into the loop buffer, the loop kernel is present in the loop buffer and the execution of the loop kernel can be entirely from the loop buffer. The last portion of the software pipeline is the epilog; which is generated by removing instructions from the buffer in the order that they were loaded into it.

In [Table 8-3](#), the instructions in the CPU pipeline are executed from program memory. The instructions in the SPL buffer are executed from the SPLOOP buffer. At K0 for example, stage3 is being executed from program memory and stage0, stage1, and stage2 are being executed from the SPLOOP buffer. At Kn and later, by contrast, all stages are being executed from the SPLOOP buffer.

**Table 8-3 Software Pipeline Instruction Flow Using the Loop Buffer**

Execution Flow	CPU Pipeline	SPL Buffer			
P0	stage0	-			
P1	stage1	stage0			
P2	stage2	stage0	stage1		
K0	stage3	stage0	stage1	stage2	
Kn	-	stage0	stage1	stage2	stage3
E0	-	-	stage1	stage2	stage3
E1	-	-	-	stage2	stage3
E2	-	-	-	-	stage3

### 8.7.2 Stage Boundary Terminology

A stage boundary is reached every ii cycles. The following terminology is used to describe specific stage boundaries.

- **First loading stage boundary:** The first stage boundary after the **SPLOOP(D/W)** instruction. The stage boundary at the end of P0 in [Table 8-3](#).
- **Last loading stage boundary:** The first stage boundary that occurs in parallel with or after the **SPKERNEL** instruction. The stage boundary at the end of K0 in [Table 8-3](#). This is the same as the first kernel stage boundary.
- **First kernel stage boundary:** The same as the last loading stage boundary.
- **Last kernel stage boundary:** The last stage boundary before the loop is only executing epilog instructions. The stage boundary at the end of Kn in [Table 8-3](#).

### 8.7.3 Loop Buffer Operation

On the cycle after an **SPLOOP(D/W)** instruction is encountered, instructions are loaded into the loop buffer. A loop buffer count register (LBC) is maintained as an index into the loop buffer. At the end of each cycle, LBC is incremented by 1. If LBC becomes equal to the ii, then a stage boundary has been reached and LBC is reset to 0. There are two LBCs to support overlapped nested loops.

The loop buffer has four basic operations:

- **Load:** instructions fetched from program memory are executed and written into the loop buffer at the current LBC and marked as valid on the next cycle
- **Fetch:** valid instructions are fetched from the loop buffer at the current LBC and executed in parallel with any instructions fetched from program memory.
- **Drain:** instructions at the current LBC are marked as invalid on the current cycle and not executed.
- **Reload:** instructions at the current LBC are marked as valid on the current cycle and executed.

The execution of a software pipeline prolog and the first kernel stage are implemented by fetching valid instructions from the loop buffer and executing them in parallel with instructions fetched from program memory. The instructions fetched from program memory are loaded into the loop buffer and marked as valid on the next cycle. The execution of the remaining kernel stages is implemented by exclusively fetching valid instructions from the loop buffer. The execution of a software pipeline epilog is implemented by draining the loop buffer by marking instructions as invalid, while fetching the remaining valid instructions from the loop buffer.

For example: referring to [Example 8-4](#) on page 8-9 and [Table 8-1](#) on page 8-10; as each instruction in loop 1 is reached in turn, it is fetched from program memory, executed and stored in the loop buffer. When each instruction is reached in loop 2 through loop 12, it is fetched from the loop buffer and executed. As cycles 8 through 12 execute, instructions in the loop buffer are marked as invalid so that for each cycle fewer instructions are fetched from the loop buffer.

The loop buffer supports the execution of a nested software pipelined loop by reenabling the instructions stored in the loop buffer. The reexecution of the software pipeline prolog is implemented by reenabling instructions in the loop buffer (by marking them as valid) and then fetching valid instructions from the loop buffer. The point of reload for the nested loop is signaled by the **SPKERNELR** or **SPMASKR** instruction.

The loop buffer also supports do-while type of constructs in which the number of iterations is not known in advance, but is determined in the course of the execution of the loop. In this case, the loop immediately completes after the last kernel stage without executing the epilog.

#### 8.7.3.1 Interrupt During SPLOOP Operation

If an interrupt occurs while a software pipeline is executing out of the loop buffer, the loop will pipe down by executing an epilog and then service the interrupt. The interrupt return address stored in the interrupt return pointer register (IRP) or the nonmaskable interrupt return pointer register (NRP) is the address of the execute packet containing the **SPLOOP** instruction. The task state register (TSR) is copied into the interrupt task state register (ITSR) or the NMI/exception task state register (NTSR) with the SPLX bit set to 1. On return from the interrupt with ITsr or NTSR copied back into TSR and the SPLX bit set to 1, execution is resumed at the address of the **SPLOOP(D/W)** instruction, and the loop is piped back up by executing a prolog.

#### 8.7.3.2 Loop Buffer Active or Idle

After reset the loop buffer is idle. The loop buffer becomes active when an **SPLOOP(D/W)** instruction is encountered. The loop buffer remains active until one of the following conditions occur:

- The loop buffer is not reloading and after the last delay slot of a taken branch.
- The **SPLOOPW** loop stage boundary termination condition is true.
- An interrupt occurs and the loop finishes draining in preparation for interrupt (prior to taking interrupt).
- The **SPLOOP(D)** loop is finished draining and the loop is not reloading.

When the loop buffer is active, the SPLX bit in TSR is set to 1; when the loop buffer is idle, the SPLX bit in TSR is cleared to 0.

There is one case where the SPLX bit is set to 1 when the loop buffer is idle. When executing a **B IRP** instruction to return to an interrupted SPLOOP, the ITSR is copied back into TSR in the E1 stage of the branch. The SPLX bit is set to 1 beginning in the E2 stage of the branch, which is before the loop buffer has restarted. If the loop buffer state machine is started in the branch delay slots of a **B IRP** or **B NRP** instruction, it uses the SPLX bit to determine if this is a restart of an interrupted SPLOOP. The SPLX bit is not checked if starting an SPLOOP outside the delay slots of one of these branches.

### 8.7.3.3 Loading Instructions into the Loop Buffer

A loading counter is used to keep track of the current offset from the beginning of the loop and to determine the dynlen. The loading counter is incremented each cycle until an **SPKERNEL** instruction is encountered. When an **SPLOOP(D/W)** instruction is encountered, LBC and the loading counter are cleared to 0. On each cycle thereafter, the instructions fetched from program memory are stored in the loop buffer indexed by LBC along with a record of the loading counter. On the next cycle, these instructions appear as valid in the loop buffer.

When the **SPKERNEL** instruction is encountered, the loop is finished loading, the dynlen is assigned the current value of the loading counter, and program memory fetch is disabled. If the **SPKERNEL** is on the last kernel stage boundary, program memory fetch may immediately be reenabled (or effectively never disabled).

**SPMASKed** instructions from program memory are not stored in the loop buffer. The **BNOP <displacement>** instruction does not use a functional unit and cannot be specified by the **SPMASK** instruction, so this instruction is treated in the same way as an **SPMASKed** instruction.

When returning to an **SPLOOP(D)** instruction with the SPLX bit in TSR set to 1, **SPMASKed** instructions from program memory execute like a **NOP**. The NOP cycles associated with **ADDKPC**, **BNOP**, or protected **LD** instructions that are masked, are always executed when resuming an interrupted **SPLOOP(D)**.

A warning or error (detected by the assembler) occurs when loading if:

- An **MVC**, **ADDKPC**, or S-unit **B** (branch) instruction appears in the **SPLOOP** body and the instruction is not masked by an **SPMASK** instruction.
- Another **SPLOOP(D)** instruction is encountered.
- The loading counter reaches 48 before an **SPKERNEL** instruction is encountered.
- A resource conflict occurs when storing an instruction in the loop buffer.
- The dynlen is less than  $ii + 1$  for **SPLOOP(D)** or  $dynlen < ii$  for **SPLOOPW**.

The assembler will ensure that there are no resource conflicts that would occur if the first kernel stage were actually reached.

### 8.7.3.4 Fetching (Dispatching) Instructions from the Loop Buffer

After the first loading stage boundary, instructions marked as valid in the loop buffer at the current LBC are fetched from the loop buffer and executed in parallel with any instructions fetched from program memory. Once fetching begins, it continues until the loop buffer is no longer active for the given loop.

Instructions fetched from the loop buffer that are masked by an **SPMASK** instruction are not executed. An instruction fetched from program memory may execute on the units that were used by an **SPMASKed** instruction. (See “[Instruction Resource Conflicts and SPMASK Operation](#)” on page 8-40).

### 8.7.3.5 Disabling (Draining) Instructions in the Loop Buffer

The loop buffer starts draining:

- On the cycle after the loop termination condition is true
- On the cycle after the interrupt is detected and the conditions required for taking the interrupt are met (see “[Interrupts](#)” on page 8-36).

The draining counter is used to retrace the order in which instructions were loaded into the loop buffer. The draining counter is initialized to 0 and then incremented by 1 each cycle. Instructions in the loop buffer are marked as invalid in the order that they were loaded.

Instructions in the loop buffer indexed by LBC are marked as invalid if their loading counter value (from when they were loaded into the loop buffer) is equal to the draining counter value.

When the draining counter is equal to ( $\text{dynlen} - \text{ii}$ ), draining is complete. Any remaining valid instructions for the loop (with a loading counter  $> (\text{dynlen} - \text{ii})$ ) are all marked as invalid.

If the loop is interrupt draining, then program memory fetch remains disabled until the interrupt is taken. If the loop is normal draining, program memory fetch is enabled after a delay specified by the **SPKERNEL(R)** instruction.

### 8.7.3.6 Enabling (Reloading) Instructions in the Loop Buffer

On the cycle after the reload condition is true (see [8.9.6](#) on page 8-25), the loop buffer begins reloading instructions in the loop buffer. Instructions in the loop buffer are marked as valid in the order that they were originally loaded.

The reloading counter is initialized to 0 and then incremented by 1 each cycle until it equals the  $\text{dynlen}$ . The reloading counter is used to retrace the order in which instructions were loaded into the loop buffer.

Instructions in the loop buffer indexed by LBC are marked as valid, if their loading counter value (from when they were written into the loop buffer) is equal to the reloading counter value.

Reloading does not have to start on a stage boundary. Reloading and draining may access different offsets in the loop buffer. Therefore, there are two LBCs. When reload begins, the unused LBC (the one not being used for draining) is allocated for reloading.

When the reloading counter is equal to the  $\text{dynlen}$ , the reloading of the software pipeline loop is complete, all the original loop instructions have been reenabled, and the reloading counter stops incrementing.

Program memory fetch of the epilog is disabled when the reload counter equals the dynlen or after the last delay slot of a branch that executed with a true condition. In general, the branch is used in a nested loop to place the PC back at the address of the execute packet after the **SPKERNEL(R)** instruction to reuse the same epilog code between each execution of the inner loop.

A hardware exception is raised while reloading if the termination condition is true and the draining counter for the previous invocation of the loop has not reached the value of dynlen -ii. This describes a condition where both invocations of the loop are attempting to drain at the same time (this could happen, for example, if the RILC value was smaller than the ILC value).

## 8.8 Execution Patterns

The four loop buffer operations (load, fetch, drain, and reload) are combined in ways that implement various software pipelined loop execution patterns. The three execution patterns are:

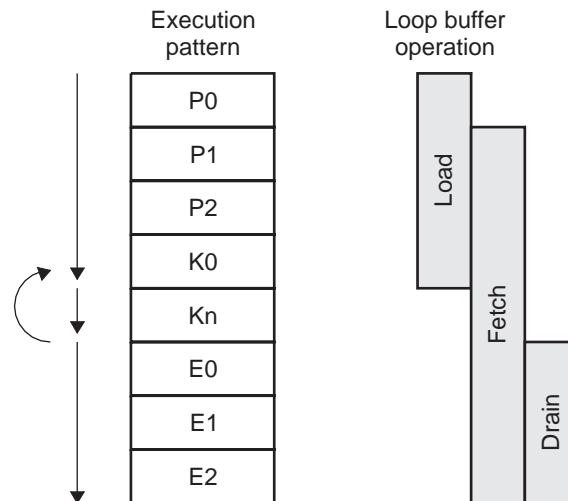
- Full execution of a single loop (Section 8.8.1 )
- Early exit from a loop (Section 8.8.2 )
- Reload of a loop (Section 8.8.3 )

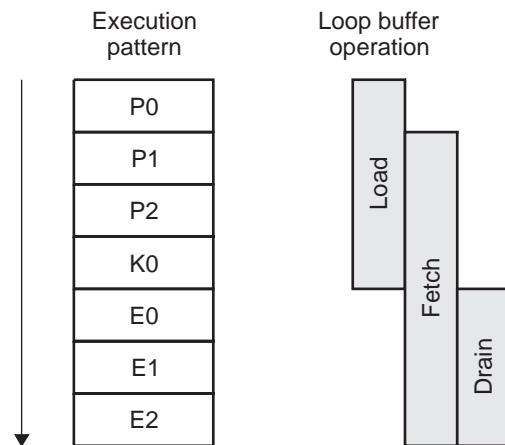
### 8.8.1 Prolog, Kernel, and Epilog Execution Patterns

Figure 8-2 shows a generalization of the basic prolog, kernel, and epilog execution pattern. For simplicity these patterns assume that the **SPKERNEL** instruction appears on a stage boundary.

In Figure 8-3, the termination condition is true on the first kernel stage boundary K0, and falling through to the epilog, the software pipeline only executes a single kernel stage.

**Figure 8-2 General Prolog, Kernel, and Epilog Execution Pattern**



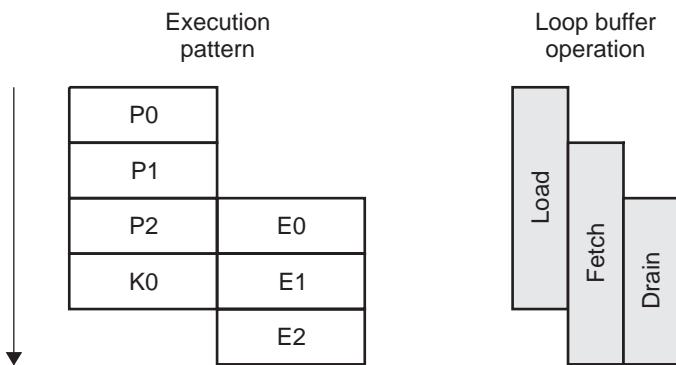
**Figure 8-3 Single Kernel Stage Execution Pattern**


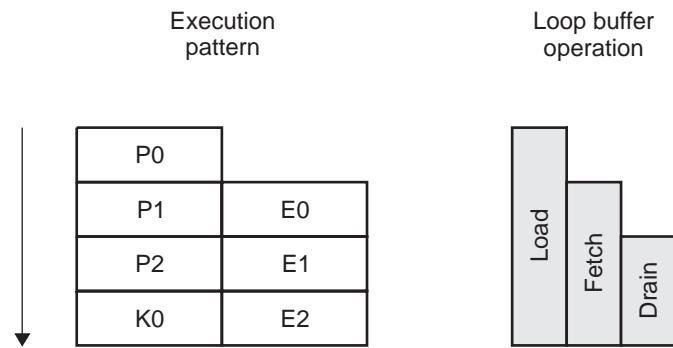
### 8.8.2 Early-Exit Execution Pattern

If the termination condition is true before an SPKERNEL(R) instruction is encountered, then the epilog execution pattern begins before the prolog execution pattern is complete. Since the loop has started draining before it has finished loading, this is referred to as an early-exit.

The execution of a software pipeline early-exit is implemented by beginning to drain the loop buffer by disabling instructions in the order that they were originally loaded, fetching the remaining valid instructions from the loop buffer, and then loading the instructions fetched from program memory into the loop buffer and marking them as valid on the next cycle. An early-exit execution pattern is shown in Figure 8-4. In this case the termination condition was found to be true at the end of P1.

If the termination condition is encountered on the first stage boundary (end of P0) as in Figure 8-5, then no instructions actually execute from the loop buffer. In this special case of early-exit, the loop is only executing a single iteration.

**Figure 8-4 Early-Exit Execution Pattern**


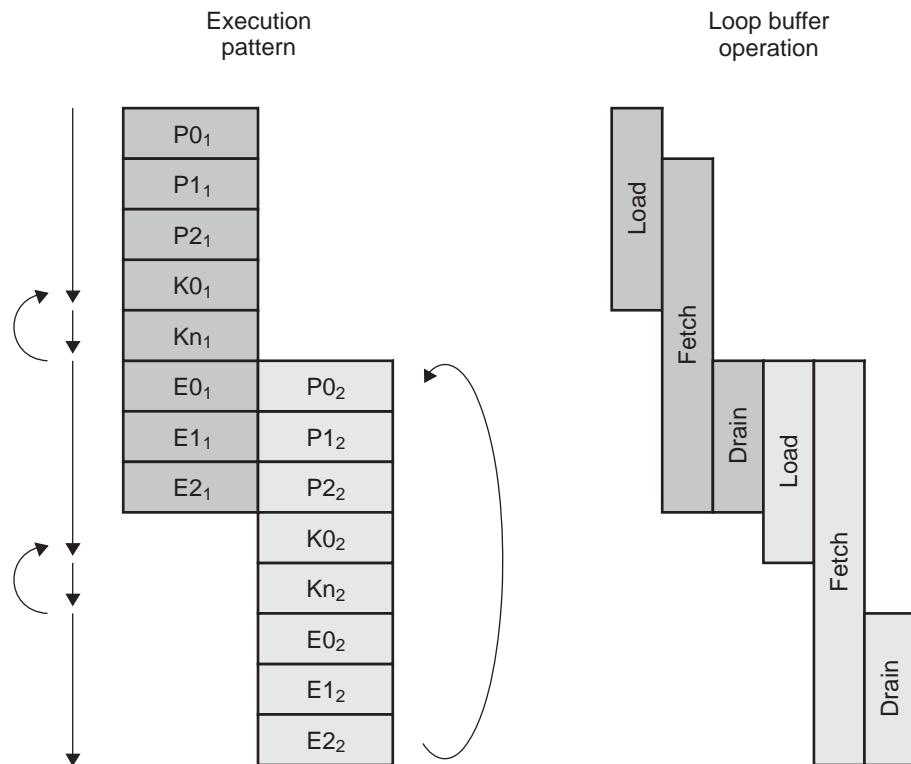
**Figure 8-5 Single Loop Iteration Execution Pattern**


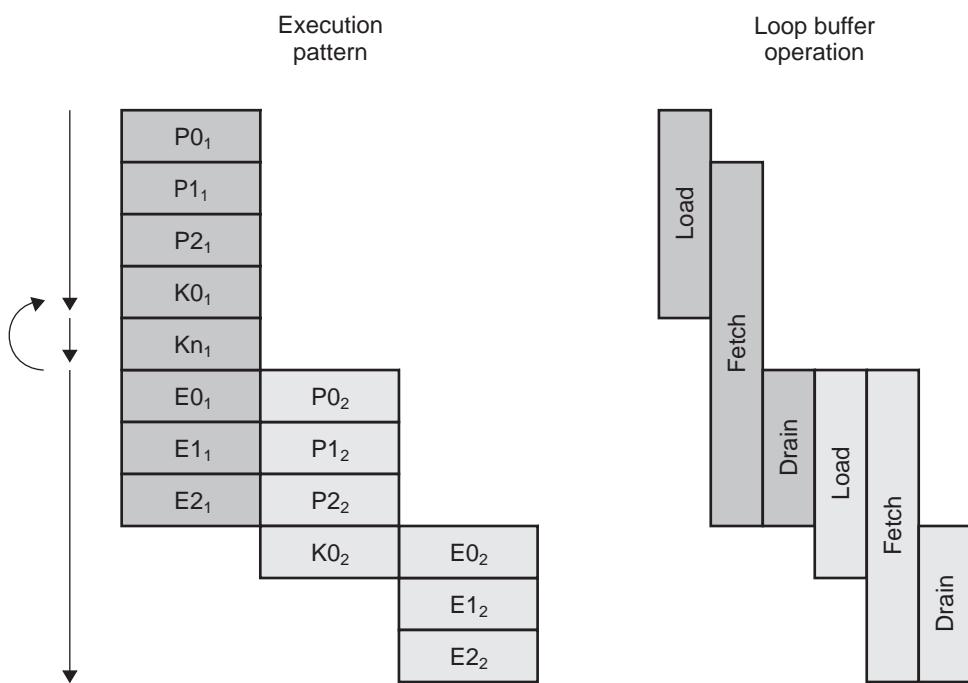
### 8.8.3 Reload Execution Pattern

The loop buffer can reload a software pipeline by reactivating the instructions that are stored in the loop buffer. A reload prolog uses the information stored in the loop buffer to reenable instructions in the order that they were originally loaded during the initial prolog.

In [Figure 8-6](#), the loop buffer begins executing a reload prolog while completing the epilog of a previous invocation of the same loop.

The execution of a reload early-exit is implemented by reloading (marking as valid) instructions in the loop buffer, disabling (marking as invalid) instructions in the loop buffer, and then fetching the remaining valid instructions from the loop buffer. A reload early-exit execution pattern is shown in [Figure 8-7](#).

**Figure 8-6 Reload Execution Pattern**


**Figure 8-7 Reload Early-Exit Execution Pattern**


## 8.9 Loop Buffer Control Using the Unconditional SPLOOP(D) Instruction

The unconditional form of the **SPLOOP(D)** instruction uses an inner loop count register (ILC) as a down counter, it can delay execution of program memory instructions overlapped with epilog instructions, and it can reload to support nested loops.

### 8.9.1 Initial Termination Condition Test and ILC Decrement

The termination condition is the set of conditions which determine whether or not to continue the execution of an SPLOOP. The initial termination condition is the value of the termination condition upon entry to the SPLOOP. When using the SPLOOPW or SPLOOPD, the initial termination condition is always false. When using the SPLOOP, the initial termination condition is true if ILC is equal to zero, false otherwise.

If the initial termination condition is true, then the following occur:

- Non-SPMASKed instructions are stored in the loop buffer as disabled.
- Non-SPMASKed instructions execute as NOPs.
- SPMASKed program memory instructions execute as normal.

If the initial termination condition is true and the **SPKERNEL** instruction is unconditional, the loop buffer is idle after the last loading stage boundary. If the **SPKERNEL** instruction is not on a stage boundary, the loop buffer issues NOPs until the last loading stage boundary. If the **SPKERNEL** instruction is conditional, indicating a possible reload, then the reload condition is evaluated at the last loading stage boundary.

When all of the following conditions are true, ILC is decremented:

- An unconditional **SPLOOP** (not SPLOOPD) instruction is encountered.
- ILC is not 0.

The bottom line is, the minimum number of iterations:

- Is zero for an SPLOOP;
- Depends on the iteration interval, but will be at least one iteration for an SPLOOPD;
- Will be at least one iteration for an SPLOOPW.

### 8.9.2 Stage Boundary Termination Condition Test and ILC Decrement

The stage boundary termination condition is true when a stage boundary is reached and ILC is equal to 0; otherwise, the stage boundary termination condition is false.

When the stage boundary termination condition is true, the loop buffer starts draining instructions.

When all of the following conditions are true, ILC is decremented:

- A stage boundary has been reached.
- ILC is not 0.
- The loop is not interrupt draining.
- The loop will not start interrupt draining on the next cycle.

For the first 3 cycles of a loop initiated by an unconditional **SPLOOPD** instruction, the stage boundary termination condition is always false, ILC decrement is disabled, and the loop cannot be interrupted.

If the loop is interrupted and after interrupt draining is complete, ILC contains the current number of remaining loop iterations.

**Example 8-9** shows a case in which the value loaded to ILC is determined at run time. The loop may begin draining at any point whenever the ILC decrements to zero (that is, the loop may execute 0 or more iterations). The comments in the example show the stage number (N), the test for termination and the conditional decrement of ILC. ILC will not decrement below zero.

### **Example 8-9 Using ILC With the SPLOOP Instruction**

```

MVC    A1,ILC      ;ILC = A1
      NOP          ;delay slot 1
      NOP          ;delay slot 2
      ZERO     A3      ;delay slot 3
SPLOOP1           ;Initial, term_condition!=ILC, if (ILC) ILC--;
LDW    *A5++,A2    ;Stage 0, term_condition!=ILC, if (ILC) ILC--;
NOP              ;Stage 1, term_condition!=ILC, if (ILC) ILC--;
NOP              ;Stage 2, term_condition!=ILC, if (ILC) ILC--;
NOP              ;Stage 3, term_condition!=ILC, if (ILC) ILC--;
NOP              ;Stage 4, term_condition!=ILC, if (ILC) ILC--;
SPKERNEL 5,0      ;Delay fetch until done with epilog
ADD    A2,A3,A3    ;StageN, term_condition!=ILC, if (ILC) ILC--;
MV     A3,A4

```

End of Example 8-9

### 8.9.3 Using SPLOOPD for Loops with Known Minimum Iteration Counts

For loops with known iteration counts, the unconditional **SPLOOPD** instruction is used to compensate for the 4-cycle latency to the assignment of ILC. The unconditional **SPLOOPD** instruction differs from the **SPLOOP** instruction in the following ways:

- The initial termination condition test is always false and the initial ILC decrement is disabled. The loop must execute at least one iteration.
  - The stage boundary termination condition is forced to false, and ILC decrement is disabled for the first 3 cycles of the loop.
  - The loop cannot be interrupted for the first 3 cycles of the loop.

The **SPLOOPD** will test the SPLX bit in the TSR to determine if it is already set to one (indicating a return from interrupt). In this case the **SPLOOPD** instruction executes like an unconditional **SPLOOP** instruction.

The **SPOOPD** instruction is used when the loop is known to execute for a minimum number of loop iterations. The required minimum of number of iterations is a function of  $i_i$ , as shown in [Table 8-4](#).

**Table 8-4 SPLOOPD Minimum Loop Iterations**

<b>ii</b>	<b>Minimum Number of Loop Iterations</b>
1	4
2	2
3	2
$\geq 4$	1

When using the **SPLOOPD** instruction, ILC must be loaded with a value that is biased to compensate for the required minimum number of loop iterations. As shown in [Example 8-10](#), for a loop with an *ii* equal to 1 that will execute 100 iterations, ILC is loaded with 96.

## **Example 8-10 Using ILC With a SPLOOPD Instruction**

```

MVK      96,B1          ;Execute 96+4 iterations
||       SPLOOPD 1        ;Initial, term condition=false
||       MVC    B1,ILC     ;ILC=A1 (E1 Stage)
||       ZERO   A3
||       LDW    *A1++,A2   ;Stage0, term condition=false
||       NOP                ;Stage1, term condition=false
||       NOP                ;Stage2, term condition=false
||       NOP                ;Stage3, term cond!=ILC; if (ILC) ILC--;
||       NOP                ;Stage3, term_cond!=ILC; if (ILC) ILC--;
||       SPKERNEL          ;StageN, term_cond!=ILC; if (ILC) ILC--;
||       ADD    A2,A3,A3     ;

```

**End of Example 8-10**

#### 8.9.4 Program Memory Fetch Enable Delay During Epilog

After the last kernel stage boundary, program memory fetch is enabled such that instructions are fetched from program memory and executed in parallel with instructions fetched from the loop buffer. This provides for overlapping post-loop instructions with loop epilog instructions.

Program memory fetch enable is delayed until a specific stage and cycle in the execution of the epilog. The **SPKERNEL** instruction fstg and fcyc operands are combined (by the assembler) to calculate the delay in instruction cycles:

$$\text{delay} = (fstg * \text{ii}) + fcyc$$

Program memory fetch is delayed until the following conditions are all true:

- The loop has reached the last kernel stage boundary
  - The loop is not interrupt draining
  - The draining counter has reached the delay value specified by  $fstg$  and  $fyc$ .

Referring back to [Example 8-4](#) on page 8-9, the program memory fetch delay is set to start fetching after the last epilog instruction.

If the loop buffer goes to idle (for example, if the epilog is smaller than the specified delay or if the loop early-exit execution pattern), program memory fetch is enabled and the fetch enable delay is ignored.

### **8.9.5 Stage Boundary and SPKERNEL(R) Position**

An **SPKERNEL(R)** instruction does not have to occur on a stage boundary. If an **SPKERNEL(R)** instruction is not on a stage boundary and the loop executes 0 or 1 iteration, then the loop buffer executes until the last loading stage boundary. If there are instructions in between the **SPKERNEL(R)** instruction and the last loading stage boundary, the loop buffer issues **NOP** instructions and program memory fetch remains disabled.

If the loop is reloading and the loop executes 0 or 1 iteration, then the loop buffer executes until the last reloading stage boundary. Between when the reloading counter becomes equal to the dynlen and the last reloading stage boundary, the loop buffer issues **NOP** instructions.

## 8.9.6 Loop Buffer Reload

Using the conditional form of the **SPLOOP(D)** instruction, the loop buffer supports the execution of a nested loop by reloading a new loop invocation while draining the previous invocation of the loop. A loop that reloads must have either an **SPKERNELR** instruction to end the loop body or an **SPMASKR** instruction in the post-**SPLOOP** code.

Under all of the following conditions, the reload condition is true.

- The loop is on the last kernel stage boundary (that is, ILC = 0).
- The **SPLOOP(D)** instruction condition is true 4 cycles before the last kernel stage boundary.

### 8.9.6.1 Reload Start

When the reload condition is true, reenabling of instructions in the loop buffer begins on the cycle after:

- the last kernel stage boundary for loops using **SPKERNELR**
- an **SPMASKR** is encountered in the post-**SPLOOP** instruction stream.

The reload does not have to start on a stage boundary of the draining loop as indicated by the second and third conditions above.

### 8.9.6.2 Resetting ILC With RILC

The reload inner loop count register (RILC) is used for resetting the inner loop count register (ILC) for the next invocation of the nested inner loop. There is a 4-cycle latency (3 delay slots) between an instruction that writes a value to RILC and the value appearing to the loop buffer.

If the initial termination condition is false, then the value stored in RILC is extracted, decremented and copied into ILC and normal reloading begins. The value of RILC is unchanged.

If RILC is equal to 0 on the cycle before the reload begins, the initial termination condition is true for the reloaded loop. If the initial termination condition is true, then the reloaded loop invocation is skipped: the instructions in the loop buffer execute as NOPs until the last reloading stage boundary and the reload condition is evaluated again.

### 8.9.6.3 Program Memory Fetch Disable During Reload

After the reload condition becomes true, program memory fetch is disabled after the last delay slot of a branch that executed with a true condition or after the reload counter equals the dynlen.

The PC remains at its current location when program memory fetch is disabled. If a branch disabled program memory fetch, then the PC remains at the branch target address.

Note that the first condition above is the only time that the loop buffer will not go to idle after the last delay slot of a taken branch.

#### 8.9.6.4 Restrictions on Interruptible Loops that Reload

When the loop buffer has finished loading after returning from an interrupt, the PC points at the address after the **SPKERNEL** instruction. A reloaded loop is not interruptible if a branch does not execute during reloading that places the PC back at the execute packet after the **SPKERNEL** instruction. You should disable interrupts around these types of loops.

#### 8.9.6.5 Restrictions on Reload Enforced by the Assembler

By enforcing the following restrictions by issuing either errors or warnings, the assembler enforces the most common and useful cases for using reload. An assembler switch disables these checks for advanced users.

There must be at least one valid outer loop branch that will always execute with a true condition when the loop is reloading. An outer loop branch is valid under all of the following conditions:

- The branch always executes if the reload condition was true.
- The branch target is the execute packet after the **SPKERNEL** execute packet.
- The last delay slot of the branch occurs before the reloading counter equals the dynlen. Note that this restriction implies a minimum for dynlen of 6 cycles.

There may be one or more valid post loop branch instructions that will always execute with a false condition when the loop is reloading, and that may execute with a true condition when the loop is not reloading.

For loops initiated with a conditional **SPLOOP** or **SPLOOPD** instruction, an exception (detected by the assembler) occurs if:

- There is not a valid outer loop branch instruction after the **SPKERNEL(R)** instruction.
- A reload has not been initiated by an **SPMASKR** instruction before the delay slots of the outer branch have completed.
- There is a branch instruction after the **SPKERNEL** instruction that may execute when the loop is reloading that is neither a valid outer loop branch nor a valid post loop branch.
- An **SPMASKR** is encountered for a loop that uses **SPKERNELR**.
- An **SPMASKR** is encountered for an unconditional (nonreload) loop.

**Example 8-11** is a nested loop using the reload condition. **Figure 8-8** shows the instruction execution flow for an invocation of the inner loop, the outer loop code, and then another inner loop. Notice that the reload starts after the first epilog stage of the inner loop as specified by the **SPMASKR** instruction in the last cycle of that stage.

##### Example 8-11 Using ILC With a SPLOOPD Instruction

```

-----*
; *-----*
; *   ;* for (j=0; j<32; j++)
; *     for (I=0; i<32; I++)
; *       y[j] += x[i+j] * h[i]
; *-----*
; * x=a4, h=b4, y=a6
MVK      .S2 32,B0
MVC      .S2 B0,ILC      ;Inner loop count
NOP      3
[BO]      SPLOOP2
          MVC  B0,RILC    ;Reload inner loop count
          SUB  B0,1,B0    ;Outer loop count
          MVK  .S1 62,A5    ;X delta
          MV   .L2 B4,B5    ;Copy h
-----*
  |||

```

```

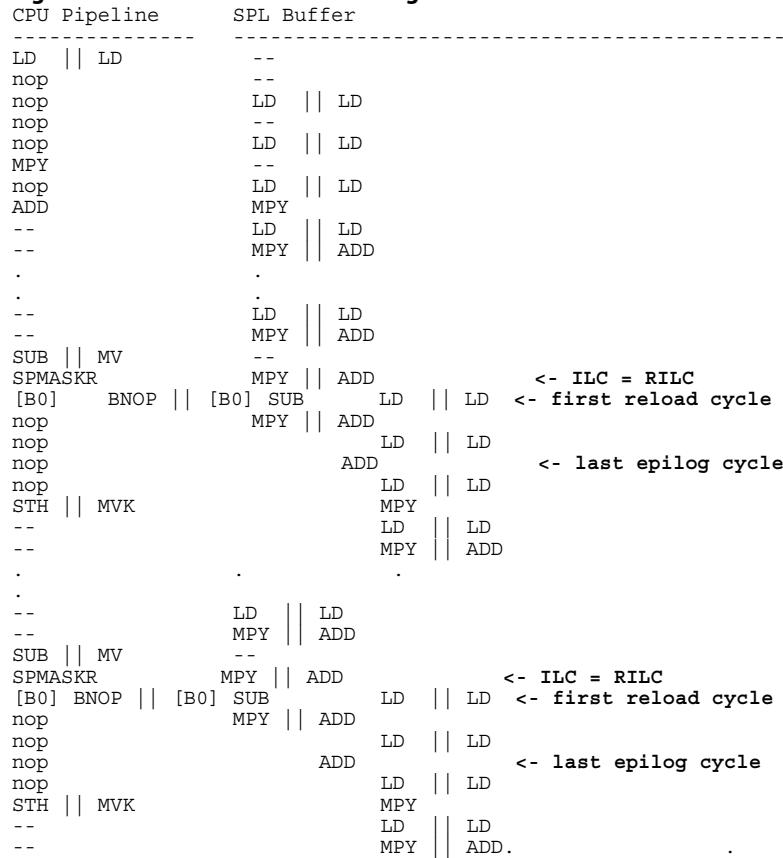
||          ZERO      .D1 A7           ;Sum = 0
;-----Start of loop-----
||          LDH       .D1T1 *A4++,A2   ;t1 = *x
||          LDH       .D2T2 *B4++,B2   ;t2 = *h
||          NOP       4
||          MPY       .M1X A2,B2,A2   ;p = t1*t2
||          NOP       1
||          SPKERNEL  0
||          ADD       .L1 A2,A7,A7   ;sum += p
outer:
;-----start epilog
||          SUB       .D1 A4,A5,A4   ;x -= 64
||          MV        .D2 B4,B5     ;h -= 64

SPMASKR
;-----start reload, I=0
[B0]        BNOP      .S1 outer,4
|| [B0]        SUB       .S2 B0,1,b0   ;j -= 1
||          STH       .D1 A7,*A6++   ;*Y++ = sum
||          MVK       .S1 0,A7     ;Sum = 0
;-----branch, stop fetching

```

**End of Example 8-11**

**Figure 8-8 Instruction Flow Using Reload**



### 8.9.7 Restrictions on Accessing ILC and RILC

There is a 4-cycle latency (3 delay slots) between an instruction that writes a value to the inner loop count register (ILC) or the reload inner loop count register (RILC) and a read of the register by the loop buffer.

If an **SPLOOP** (not SPLOOPD or SPLOOPW) instruction is used, then ILC is used for the 3 cycles before the **SPLOOP** instruction and until the loop buffer is draining and not reloading.

If an **SPLOOPD** instruction is used, then ILC is used on the first cycle after the **SPLOOPD** instruction and until the loop buffer is draining and not reloading.

In general, it is an error to read or write ILC or RILC while the loop buffer is using them. This error is enforced by the following hardware and assembler exceptions. The value obtained by reading ILC during loading is not assured to be consistent across different implementations, due to potential differences in timing of the decrement of the register by the loop hardware.

An exception (detected by hardware) occurs if:

- RILC is written in the 3 cycles before the loop buffer reads it on the cycle before reloading begins.
- ILC is written in the 3 cycles before an unconditional **SPLOOP** (not SPLOOPD) instruction.

An error or warning (detected by the assembler) occurs if:

- An **MVC** instruction that writes ILC appears in parallel or in the 3 execute packets preceding an **SPLOOP** (not SPLOOPD) instruction.
- An **MVC** instruction that reads or writes ILC appears in the **SPLOOP** body.
- An **MVC** instruction that writes the RILC appears in the 3 execute packets preceding the execute packet before a reload prolog is initiated.
- An **MVC** instruction that reads or writes the ILC appears in an execute packet after an **SPKERNEL** instruction in a nested loop, and the **MVC** instruction may execute during or in three cycles preceding the reload prolog of the loop.

## 8.10 Loop Buffer Control Using the SPLOOPW Instruction

For the **SPLOOPW** instruction, the termination condition is determined by evaluating the **SPLOOPW** instruction condition operand. When the **SPLOOPW** instruction is encountered, the condition operand is recorded by the loop buffer. The initial termination testing is the same as for the **SPLOOPD** instruction, that is, no checking is done for the first four cycles of the loop.

The **SPLOOPW** instruction is intended to be used for do-while loops. These are loops whose termination condition is more complex than a simple down counter by 1. In addition, these types of loops compute the loop termination condition and exit without executing an epilog. This technique may require over executing (or speculating) some instructions.

When using the **SPLOOPW** instruction condition operand as the termination condition, the following behavior occurs:

- Termination is determined by the **SPLOOPW** instruction condition that will be on a stage boundary.
- ILC and RILC are not accessed or modified.
- The loop cannot be reloaded.
- After the last kernel stage boundary, the loop buffer goes idle.
- The stage boundary termination condition is evaluated while interrupt draining.
- The SPKERNEL fetch delay must be 0.
- When returning to a conditional SPLOOPW from an interrupt with the SPLX bit set to 1 in TSR, the SPLOOPW retains its delayed initial termination testing behavior. This is different from the **SPLOOPD** instruction.

### 8.10.1 Initial Termination Condition Using the SPLOOPW Condition

The initial termination condition is always false when an **SPLOOPW** instruction is encountered. The loop must execute at least one iteration.

### 8.10.2 Stage Boundary Termination Condition Using the SPLOOPW Condition

The stage boundary termination condition is true when a stage boundary is reached and the **SPLOOPW** instruction condition operand evaluates as false 3 cycles before the stage boundary; otherwise, the termination condition is false. The termination condition is always false for the first 3 cycles of the loop.

### 8.10.3 Interrupting the Loop Buffer When Using SPLOOPW

If the loop is interrupted when using the conditional form of the **SPLOOPW** instruction, the stage boundary termination condition is evaluated on each stage boundary while interrupt draining. If the stage boundary termination condition is true while interrupt draining, the loop buffer goes to idle, execution resumes at the instruction after the loop body, and that instruction is interrupted.

The instruction that defines the termination condition register must occur at least 4 cycles before a stage boundary and at least 4 cycles before the last instruction in the loop. If the termination condition is determined in the last loading stage, the dynlen must be a multiple of ii. These restrictions ensure that on return from an interrupt to a **SPLOOPW** instruction, the loop executes 1 or more iterations.

Note that when returning to a **SPLOOPW** instruction from an interrupt service routine with the SPLX bit set to 1 in TSR, the **SPLOOPW** instruction termination condition behavior is unchanged, that is, the initial termination condition is always false, and the stage boundary termination condition is always false for the first 3 cycles of the loop.

An exception occurs if the termination condition register is not defined properly to ensure correct interrupt behavior.

**Example 8-12** shows a loop with a loop counter that down counts by an unknown value. For this loop, it must be safe to over-execute the **LDH** instructions 8 times.

**Example 8-13** shows a string copy implementation. **Figure 8-9** shows the execution flow if the source points to a null string. In this version, it must be safe to over-execute the **LDB** instruction 4 times.

#### Example 8-12 Using the SPLOOPW Instruction

```

-----*
;* do  {
;    sum += *x++ * *y++;
;    n -= m;
;    } while (n >= 0)
;*-
[!A1]   SPLOOPW 1
||      MVK    .S1 0x0,A1      ;C = false
||      LDH    .D1T1 *A5++,A3  ;t1 = *x++
||      LDH    .D2T2 *B5++,B6  ;t2 = *y++
||      NOP    2
||      SUB    .L2 B4,B7,B4  ;n -=m
||      CMPLT  .L2 B4,0,A1  ;c = n < 0          // term_cond = !A1
||      MPY    .M1X B6,A3,A4  ;p = t1 * t2        // delay slot 1
||      NOP    1              // delay slot 2
||      ADD    .L1 A4,A6,A6  ;sum += p;           // delay slot 3
||      SPKERNEL  ;if (c) break; // cycle term_cond
||                                // used
  
```

**End of Example 8-12**

#### Example 8-13 strcpy() Using the SPLOOPW Instruction

```

-----*
;*-
;* do  {
;    t = *src++;
;    *dst++ = t;
;    } while (t != 0)
;*-
[A0]   SPLOOPW 1
||      MVK    .S2 1,B0
||      MVK    .S1 1,A0
||      [A0]  LDB    .D1 *A4++,A0      ;t = *src++
||      NOP    4
||      [B0]  MV     .L2X A0,B0      ;if (!t) break;
||      NOP    2                  ;Ensure A0 set 4 cycles early
||      SPKERNEL
||      [B0]  STB   .D2 B0,*B4++    ;*dest++ = t
||            B0,*B4             ;*t = '\0'
  
```

**End of Example 8-13**

**Figure 8-9 Instruction Flow for strcpy() of Null String**

CPU Pipeline	SPL buffer
[a0] LDB *a4++,a0	--
nop	[a0] LDB *a4++,a0
nop	[a0] LDB *a4++,a0
nop	[a0] LDB *a4++,a0
nop	[a0] LDB *a4++,a0 0 written to a0 in this cycle
[b0] mv a0,b0	[a0] LDB *a4++,a0 <- a0 = 0, term_cond = true
nop	[a0] LDB *a4++,a0    [b0] mv a0,b0 <- b0 = 0
nop	[a0] LDB *a4++,a0    [b0] mv a0,b0
[b0] STB b0,*b4++	[a0] LDB *a4++,a0    [b0] mv a0,b0
STB b0,*a4	- terminate string in post epilog with /0

### 8.10.4 Under-Execution of Early Stages of SPLOOPW When Termination Condition Becomes True While Interrupt Draining

Usually an SPLOOPW block terminates abruptly when the termination condition is true without executing an epilog; however, when an SPLOOPW block is interrupted, it executes an epilog to drain the loop prior to servicing the interrupt.

If the termination condition becomes true while interrupt draining, the action of interrupt draining results in the under-execution of the early stages of the loop body in comparison to the same loop when not interrupted. The loop body must be coded such that the under-execution of the early stages of the loop body are safe.

## 8.11 Using the SPMASK Instruction

A logical progression for a loop might be:

- Do initial setup
- Execute the loop
- Do post loop operations

If the loop were to be reloaded, the progression might loop like:

- Do initial setup
- Execute the loop
- Adjust setup for reloaded loop
- Reload the loop
- Do post loop operations

The initial setup, the post loop operations, and adjusting the setup for the reloaded loop are all overhead that may be minimized by moving their execution to within the same instruction cycles as the operation of the SPLOOP.

If some setup code is required to do some initialization that is not used until late in the loop; you can save instruction cycles by using the **SPMASK** instruction to overlay the setup code with the first few cycles of the SPLOOP. The **SPMASK** will cause the masked instructions to be executed once without being loaded to the SPLOOP buffer.

[Example 8-14](#) shows how this might be done.

If the **SPMASK** is used in the outer loop code (that is, post epilog code), it will force the substitution of the **SPMASKed** instructions in the outer loop code for the instruction using the same functional unit in the SPLOOP buffer for the first iteration of the reloaded inner loop. For example, if pointers need to be reset at the point that a loop is

reloaded, the instructions that do the reset can be inhibited using the **SPMASK** instruction so that the instructions that originally adjusted the pointers are replaced in the execution flow with instruction in the outer loop that are marked with the **SPMASK** instruction. [Example 8-15](#) shows how this might be done.

### 8.11.1 Using SPMASK to Merge Setup Code Example

[Example 8-14](#) copies a number of words (the number is passed in the A6 register from one buffer to an offset into another buffer). The size of the offset is passed in the B6 register. Due to the use of the **SPMASK** instruction, the ADD instruction is executed only once and is not loaded to the SPLOOP buffer. The caret (^) symbol is used to identify the instructions masked by the **SPMASK** instruction. [Table 8-4](#) shows the instruction flow for the first three iterations of the loop.

**Example 8-14 Using the SPMASK Instruction to Merge Setup Code with SPLOOPW**

```

-----  

;-----  

; dst=&(dst [n] )  

;* do {  

;     t = *src++;  

;     *dst++ = t;  

; } while (count--)  

;  

;A4 = Source address  

;B4 = Destination address  

;A6 = Number of words to copy  

;B6 = Offset into destination to do copy  

;*-----  

[A1]      SPLOOPW 1  

||          ADD    .L1 A6,1,A1      ;Position loop cnt to valid reg  

||          SHL    .S2 B6,2,B6      ;Adjust offset for size of WORD  

||          SPMASK  

||^          ADD    .L2 B6,B4,B4  ;Add offset into buffer to dest  

||          LDW    .D1 *A4++,A0  ;Load word and inc ptr  

||          NOP    1              ;Wait for portion of delay  

[A1]      SUB    .S1 A1,1,A1      ;Decrement loop count  

||          NOP    2              ;Complete necessary wait  

||          MV     .L2X A0,B0      ;Position Word for write  

||          SPKERNEL 0,0  

||          STW    .D2 B0,*B4++    ;Store word
-----
```

**End of Example 8-14**

**Table 8-5 SPLOOP Instruction Flow for First Three Cycles of Example 8-15**

Cycle	Loop			Notes
	1	2	3	
0	ADD SHL			Instructions are in parallel with the SPLOOP, so they execute only once.
1	ADD LDW			The ADD is SPMASKed so it executes only once. The LDW is loaded to the SPLOOP buffer.
2	NOP	LDW		The ADD was not added to the SPLOOP buffer in cycle 2, so it is not executed here.
3	SUB	NOP	LDW	The SUB is a conditional instruction and may not execute.
4	NOP	SUB	NOP	The SUB is a conditional instruction and may not execute.
5	NOP	NOP	SUB	The SUB is a conditional instruction and may not execute.
6	MV	NOP	NOP	
7	STW	MV	NOP	
8		STW	MV	
9			STW	

### 8.11.2 Some Points About the SPMASK to Merge Setup Code Example

Note the following points about the execution of [Example 8-15](#):

- The **ADD** and **SHL** instructions in the same execute packet as the **SPLOOPW** instruction are only executed once. They are not loaded to the SPLOOP buffer.
  - Because of the **SPMASK** instruction in the execute packet, the **ADD** in the same execute packet as the **SPMASK** instruction is executed only once and is not loaded to the SPLOOP buffer. Without the **SPMASK**, the **ADD** would conflict with the **MV** instruction.
  - The **SHL** and the 2nd **ADD** instructions could have been placed before the start of the SPLOOP, but by placing the **SHL** in parallel with the **SPLOOP** instruction and by using the **SPMASK** to restrict the **ADD** to a single execution, you have saved a couple of instruction cycles.

### 8.11.3 Using SPMASK to Merge Reset Code Example

**Example 8-15** copies a number of words (the number is passed in the A8 register from one buffer to another buffer). The loop is reloaded and the contents of a second source buffer are copied to a second destination buffer. **Table 8-5** shows the instruction flow for the first 13 cycles of the example.

#### **Example 8-15 Using the SPMASK Instruction to Merge Reset Code with SPLOOP**

```

;*-----*
;    dst=&(dst[n])
;* do  {
;        t = *src++;
;        *dst++ = t;
;    } while (count--)
; adjust buffer pointers
;* do  {
;        t = *src++;
;        *dst++ = t;
;    } while (count--)
;
;A4 = 1st source address
;B4 = 1st destination address
;A6 = 2nd source address
;B6 = 2nd destination address
;A8 = number of locations to copy from each buffer
;*-----*
MVC    A8,ILC           ;Setup number of loops
MVC    A8,RILC          ;Reload count
MVK    1,A1              ;Reload flag
        NOP    3            ;Wait for ILC load to complete
[A1] SPLOOP 1             ;Start SPLOOP with ii=1
    LDW    .D1 *A4++,A0   ;Load value from buffer
    NOP    4              ;Wait for it to arrive
    MV     .L2X A0,B0      ;Move it to other side for xfer
    SPKERNELR             ;End of SPLOOP, immediate reload
    STW    .D2 B0,*B4++    ;...and store value to buffer
||| BR_TARGET:             ;Mask LDW instruction
||| [A1] B      BR_TARGET  ;Branch to start if post-epilog
||| [A1] SUB   .S1 A1, 1, A1 ;Adjust reload flag
||| [A1] LDW   .D1 *A6,A0   ;Load first word of 2nd buffer
||| [A1] ADD   .L1 A6,4,A4   ;Select new source buffer
        NOP    4            ;Keep in sync with SPLOOP body
        OR     .S2 B6,0,B4   ;Adjust destination to 2nd buffer
        NOP

```

**End of Example 8-15**

**Table 8-6 SPLOOP Instruction Flow for Example 8-15**

Cycle	Loop												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	LDW												
2	NOP	LDW											
3	NOP	NOP	LDW										
4	NOP	NOP	NOP	LDW									
5	NOP	NOP	NOP	NOP	LDW								
6	MV	NOP	NOP	NOP	NOP	LDW							
7	STW	MV	NOP	NOP	NOP	NOP	LDW						
8		STW	MV	NOP	NOP	NOP	NOP	LDW	SUB	ADD			
9			STW	MV	NOP	NOP	NOP	NOP	LDW				
10				STW	MV	NOP	NOP	NOP	NOP	LDW			
11					STW	MV	NOP	NOP	NOP	NOP	LDW		
12						STW	MV	NOP	NOP	NOP	NOP	LDW	
13							STW	MV	NOP	NOP	NOP	NOP	LDW
14								STW	MV	NOP	NOP	NOP	NOP

#### 8.11.4 Some Points About the SPMASK to Merge Reset Code Example

Note the following points about the execution of [Example 8-15](#) (see [Table 8-5](#) for the instruction flow):

- The loop begins reloading from the SPLOOP buffer immediately after the **SPKERNELR** instruction with no delay. In [Table 8-5](#), the **SPKERNELR** is in cycle 7 and the reload happens in cycle 8.
- Because of the **SPMASK** instruction, the **LDW** instruction in the post epilog code replaces the **LDW** instruction within the loop, so that the first word copied in the reloaded loop is from the new input buffer. The **ADD** instruction is used to adjust the source buffer address for subsequent iterations within the SPLOOP body. In [Table 8-5](#), this happens in loop 8. Note that the D1 operand in the **SPMASK** instruction indicates that the **SPMASK** applies to the .D1 unit. This could have been indicated by marking the **LDW** instruction with a caret (^) instead.
- The **OR** instructions are used to adjust the destination address. It is positioned in the post-epilog code as the **MV** instruction is within the SPLOOP body so that it will not corrupt the data from the **STW** instructions within the SPLOOP epilog still executing from before the reload. In [Table 8-5](#), this happens in cycle 13 (loop 8).
- The **B** instruction is used to reset the program counter to the start of the epilog between executions of the inner loop.

#### 8.11.5 Returning from an Interrupt

When an SPLOOP is piping up after returning from an interrupt, the **SPMASKed** instructions coming from the buffer are executed and instructions coming from program memory are not executed.

## 8.12 Program Memory Fetch Control

When the loop buffer is active and program memory fetch is enabled, then instructions are fetched from program memory and the loop buffer and executed in parallel.

When the loop buffer is active and under certain conditions as described below, instruction execution from program memory is suspended. When this occurs, instructions are only fetched and executed from the loop buffer and the PC is unchanged.

### 8.12.1 Program Memory Fetch Disable

Instruction fetch from program memory is disabled under the following conditions:

- **When loading:** on the cycle after the **SPKERNEL** instruction is encountered, and that cycle is either a kernel cycle or a draining cycle where fetch has not yet been reenabled.
- **When reloading:** on the cycle after the last delay slot of a branch that executes with a true condition or after the reload counter equals the dynlen.

If program memory fetch is disabled on the last loading or reloading stage boundary, the stage boundary termination condition is true, and the program memory fetch enable delay has completed, then program memory fetch is not disabled.

Program memory fetch remains disabled while interrupt draining or until a specific stage and cycle during noninterrupt draining as determined by the program fetch enable delay operand of the **SPKERNEL** instruction.

### 8.12.2 Program Memory Fetch Enable

Program memory fetch is enabled, if the loop buffer goes idle.

## 8.13 Interrupts

When an **SPLOOP(D/W)** instruction is encountered, the address of the execute packet containing the **SPLOOP(D/W)** instruction is recorded. If the loop buffer is interrupted, the address stored in the interrupt return pointer register (IRP) is the address of the execute packet containing the **SPLOOP(D/W)** instruction.

### 8.13.1 Interrupting the Loop Buffer

Interrupts are automatically disabled 2 cycles before an **SPLOOP(D/W)** instruction is encountered. When all of the following conditions are true, the loop buffer begins interrupt draining.

- An enabled interrupt is pending and not architecturally blocked (for example, in branch delay slots).
- The loop is on a stage boundary.
- The termination condition is false.
- The loop is not loading.
- The loop is not draining.
- The loop is not reloading or waiting to reload.
- The loop is not within the first 3 CPU cycles after an **SPLOOPD** or **SPLOOPW** instruction. This means that a minimum number of 4 cycles of an **SPLOOP(D/W)** loop must be executed before an interrupt can be taken.
- The loop is not within the first 3 CPU cycles after an **SPLOOPD** or **SPLOOPW** instruction.
- For **SPLOOP** or **SPLOOPD** instructions, the current ILC  $\geq \text{ceil}(\text{dynlen}/\text{ii})$ . This prevents returning to a loop that would early-exit. The value of  $\text{ceil}(\text{dynlen}/\text{ii})$  is equal to the number of loading stages.

When the loop is finished draining and all pending register writes are complete the interrupt is taken. This means that the interrupt latency has increased by the number of instruction cycles in the epilog compared to the non-SPLOOP case.

The above conditions mean SPLOOP loops starting initial execution or starting reload with  $\text{ILC} \leq (\text{ceil}(\text{dynlen} / \text{ii}) + 3)$  are not interruptible because there are not enough kernel stages to allow an interrupt to be taken without violating the last requirement.

After an **SPLOOP(D/W)** instruction is encountered, the SPLX bit is set to 1 in TSR. While the loop buffer is active, the SPLX bit is 1. When the loop buffer is idle, the SPLX bit in TSR is cleared to 0.

Program memory fetch is disabled when interrupt draining. When the draining is finished, the address of the execute packet that contains the **SPLOOP** instruction is stored in IRP or NRP, and TSR is copied to ITSR or NTSR. The SPLX bit in TSR is cleared to 0. The SPLX bit in ITSR or NTSR is set to 1.

Interrupt service routines must save and restore the ITSR or NTSR, ILC, and RILC registers. A **B IRP** instruction copies ITSR to TSR, and a **B NRP** restores TSR from NTSR. The value of the SPLX bit in ITSR or NTSR when the return branch is executed is used to alter the behavior of **SPLOOP(D/W)** when it is restarted upon returning from the interrupt.

### 8.13.2 Returning to an SPLOOP(D/W) After an Interrupt

When returning from an interrupt to an SPLOOP(D/W) instruction with the SPLX bit set to 1 in ITSR, the loop buffer executes normally with the following exceptions:

- Instructions executing in parallel with the **SPLOOP(D/W)** instruction are not executed.
- SPMASKed instructions from program memory execute as a NOP.
- SPMASKed instructions in the loop buffer execute as normal - the SPMASK is ignored.
- BNOP *label,n* instructions are executed as NOP*n* + 1
- An **SPLOOPD** instruction executes as an **SPLOOP** instruction.

Note that if returning to an unconditional **SPLOOP(D)** instruction, the interrupt return code must restore the value of ILC 4 cycles before the **SPLOOP(D)** instruction is executed (if the ISR modified ILC).

### 8.13.3 Exceptions

If an internal or external exception occurs while the loop buffer is active, then the following occur:

- The exception is recognized immediately and the loop buffer becomes idle.
- The loop buffer does not execute an epilog to drain the currently executing loop.
- TSR is copied into NTSR with the SPLX bit set to 1 in NTSR and cleared to 0 in TSR.

### 8.13.4 Branch to Interrupt, Pipe-Down Sequence

1. Hardware detects an interrupt.
2. Execute until the end of a stage boundary (if termination condition is false).
3. Pipe-down the SPLOOP by draining.
4. Fetch enable condition is false.
5. Store return address of SPLOOP in IRP or NRP.
6. Copy TSR to ITSR or NTSR with SPLX bit set to 1.
7. Complete all pending register writes (drain pipeline).
8. Begin execution at interrupt service routine target address.

### 8.13.5 Return from Interrupt, Pipe-Up Sequence

9. Copy ITSR or NTSR to TSR.
10. Pipe-up the SPLOOP.
11. The **SPLOOPD** instruction executes like the **SPLOOP** instruction.
12. The **SPMASKed** instructions from program memory are executed like NOPs.
13. The **SPMASKed** instructions in the loop buffer execute as normal.
14. Instructions in parallel with the **SPLOOP(D/W)** instruction are executed like NOPs.

### 8.13.6 Disabling Interrupts During Loop Buffer Operation

Instructions that disable interrupts should not be executed within 4 cycles of reaching dynlen while loading or reloading. If this condition is violated, there is a possibility that an interrupt is recognized as enabled causing the loop to drain for an interrupt with the interrupt no longer enabled when draining is completed. In this case, the loop terminates and execution continues with the post-**SPKERNEL** instruction stream with no interrupt being serviced at that point.

## 8.14 Branch Instructions

If a branch executes with a true condition or is unconditional, then the branch is taken on the cycle after the 5 delay slots have expired.

If a branch is taken and the loop buffer is not reloading, the loop buffer becomes idle, and execution continues from the branch target address.

If a branch executes with a false condition (the branch is not taken), the execution of the **SPLOOP(D/W)** instruction is unaffected by the presence of the untaken branch except that interrupts are blocked during the delay slots of the branch.

This behavior allows the code in [Example 8-16](#) to run as you expect, branching around the loop if the condition is false before beginning.

If a branch is taken anytime while the loop buffer is active, except when in reloading, the loop buffer goes to idle, and execution continues from the branch target address. If a branch is taken while reloading, the PC is assigned the branch target and program memory fetch is disabled.

### **Example 8-16 Initiating a Branch Prior to SPLOOP Body**

```
-----  
[!A0]      B      around  
||  
          MVC   A0, ILC  
          NOP   3  
          SPLOOPii  
; loop body  
; . . .  
; end of loop body  
around:  
; code following loop
```

**End of Example 8-16**

## 8.15 Instruction Resource Conflicts and SPMASK Operation

There are three execution candidates for each unit: prolog instructions coming from the buffer (BP), epilog instructions coming from the buffer (BE), and nonbuffer instructions coming from program memory (PM). There are four phases where conflict can occur:

- Loading phase, between BP and PM
- Draining only phase, between BE and PM
- Draining/reload phase, between BE, BP, and PM
- Reload only phase, between BP and PM

In the case of any conflict, an **SPMASK(R)** instruction must be present specifying all units having conflicts. SPMASKed units for that cycle:

- Disable execution of any loop buffer instructions: BP, BE, or both.
- Execute a PM instruction, if present, with no effect on the buffer contents.

The only special behavior is in the case of restarting SPLOOP(D/W) after return from interrupt. In this case, during loading SPMASKed units:

- Do not disable execution of any loop buffer instructions.
- Do not execute a present PM instruction.

If an **SPMASK** instruction is encountered when the loop buffer is idle or not loading or not draining, the **SPMASK** instruction executes as a NOP.

### 8.15.1 Program Memory and Loop Buffer Resource Conflicts

A hardware exception occurs if:

- An instruction fetched from program memory has a resource conflict with an instruction fetched from the loop buffer and the instruction coming from the loop buffer is not masked by an **SPMASK(R)** instruction.
- An instruction fetched from the loop buffer as part of draining has a resource conflict with an instruction fetched from the loop buffer for reload and the unit with the conflict is not masked by an **SPMASK(R)** instruction.

### 8.15.2 Restrictions on Stall Detection Within SPLOOP Operation

There are two CPU stalls that occur because of certain back-to-back execute packets. In both of these cases, the CPU generates a 1-cycle stall between the instruction doing the write and instruction using the written value.

- The crosspath register file read stall where the source for an instruction executing on one side of the datapath is a register from the opposite side and that register was written in the previous cycle. No stall is required or inserted when the register being read has data placed by a load instruction. See 3.8.5 “[Cross Path Stalls](#)” on page 3-17 for more information.
- The AMR use stall where an instruction uses an address register in the cycle immediately following a write to the addressing mode register (AMR).

Stall detection is one critical speed path in the CPU design. Adding to that path for the case where instructions are coming from the loop buffer is undesirable and unnecessary. There are no compelling cases where you would want to schedule a stall within the loop body. In fact, the compiler works to ensure this does not happen. For these reasons, the CPU will not stall for instructions coming from the loop buffer that read/use values written on the previous cycle that require a stall for correct behavior.

In the event that a case occurs where a stall is required for correct operation but did not occur, an internal exception is generated. This internal exception sets the LBX and MSX bits in the internal exception report register (IERR), indicating a missed stall with loop buffer operation. The exception is only generated in the event that the stall is actually required.

There is one special case that causes an unnecessary stall in normal operation and can be generated by the compiler. It is the case where the two instructions involved in the stall detection are predicated on opposite conditions. This means only one of the instructions actually executes and a stall was not required for correct behavior. Since the stall detection is earlier in the pipeline, the decision to stall must be made before it is known whether the instructions execute. Thus a stall is caused, even though it later turns out not to be needed. In this case, the lack of detection for the instruction coming from the loop buffer does not cause incorrect behavior. This allows the compiler to continue to generate code using this case that can result in improved scheduling and performance. The internal exception is not generated in this case.

## 8.16 Restrictions on Crosspath Stalls

The following restriction is enforced by the assembler (that is, an assembly error will be signaled): an instruction fetched from the loop buffer that reads a source operand from the register file crosspath must be scheduled such that the read does not require a crosspath stall (the register being read cannot be written in the previous cycle).

It is possible for the assembly language programmer to place an instruction in the delay slots of a branch to an SPLOOP that causes a pipelined write to happen while the loop buffer is active. It is also possible for the assembly language programmer to predicate the write and reads with different predicate values that are not mutually exclusive. The assembler cannot prevent these cases from occurring; if they do the internal exception will occur.

## 8.17 Restrictions on AMR-Related Stalls

The following restriction is enforced by the assembler: an instruction fetched from the loop buffer that uses an address register (A4-A7 or B4-B7) must be scheduled such that a write to the addressing mode register (AMR) does not occur in the preceding cycle.

## 8.18 Restrictions on Instructions Placed in the Loop Buffer

The following instructions cannot be placed in the loop buffer and must be masked by SPMASK(R) when occurring in the loop body: **ADDKPC**, **B reg**, **BNOP reg**, **CALLP**, and **MVC**.

The **NOP**, **NOP n**, and **BNOP** instructions are the only unitless instructions allowed to be used in an SPLOOP(D/W) body. The assembler disallows the use of any other unitless instruction in the loop body.



# CPU Privilege

This chapter describes the CPU privilege system.

- 9.1 "Overview" on page 9-1
- 9.2 "Execution Modes" on page 9-2
- 9.3 "Interrupts and Exception Handling" on page 9-4
- 9.4 "Operating System Entry" on page 9-5

### 9.1 Overview

The CPU includes support for a form of protected-mode operation with a two-level system of privileged program execution.

The privilege system is designed to support several objectives:

- Support the emergence of higher capability operating systems on the C6000 family architecture.
- Support more robust end-equipment, especially in conjunction with exceptions.
- Provide protection to support system features such as memory protection.

The support for powerful operating systems is especially important. By dividing operation into privileged and unprivileged modes, the operating mode for the operating system is differentiated from applications, allowing the operating system to have special privilege to manage the processor and system. In particular, privilege allows the operating system to:

- control the operation of unprivileged software
- protect access to critical system resources (that is, interrupts)
- control entry to itself

The privilege system allows two distinct types of operation.

- Supervisor-only execution. This is used for programs that require full access to all control registers, and have no need to run unprivileged (User mode) programs.
- Two-tiered system. This is where the OS and trusted applications execute in Supervisor mode, and less trusted applications execute in User mode.

## 9.2 Execution Modes

There are two execution modes:

- Supervisor Mode
- User Mode

### 9.2.1 Privilege Mode After Reset

Reset forces the CPU to the Supervisor mode. Execution of the reset interrupt service fetch packet (ISFP) begins in Supervisor mode.

### 9.2.2 Execution Mode Transitions

Mode transitions occur only on the following events:

- Interrupt: goes to Supervisor mode and saves mode (return with **B IRP** instruction)
- **B IRP** instruction: returns to saved mode from interrupt
- Nonmaskable interrupt (NMI): goes to Supervisor mode and saves mode (return with **B NRP** instruction)
- Exception: goes to Supervisor mode and saves mode (return with **B NRP** instruction, if restartable)
- Operating system service request: goes to Supervisor mode and saves mode (return with **B NRP** instruction)
- **B NRP** instruction: returns to saved mode from NMI or exception

### 9.2.3 Supervisor Mode

The Supervisor mode serves two purposes:

1. It is the compatible execution mode.
2. It is the privileged execution mode where all functions of the processor are available. In User mode, the privileged operations and resources that are restricted are listed in Section 9.2.4 .

### 9.2.4 User Mode

The User mode provides restricted capabilities such that more privileged supervisory software may manage the machine with complete authority. User mode restricts access to certain instructions and control registers to prevent an unprivileged program from bypassing the management of the hardware by the supervisory software.

#### 9.2.4.1 Restricted Control Register Access in User Mode

Certain control registers are not available for use in User mode. An attempt to access one of these registers in User mode results in an exception. The resource access exception (RAX) and privilege exception (PRX) bits are set in the internal exception report register (IERR) when this exception occurs. The following control registers are restricted from access in User mode:

- Exception clear register (ECR)
- Exception flags register (EFR)
- Interrupt clear register (ICR)
- Interrupt enable register (IER)
- Internal exception report register (IERR)
- Interrupt flags register (IFR)
- Interrupt service table pointer register (ISTP)

- Interrupt task state register (ITSR)
- NMI/exception task state register (NTSR)
- Restricted entry point register (REP)

#### 9.2.4.2 Partially Restricted Control Register Access in User Mode

The following control registers are partially restricted from access in User mode:

- Control status register (CSR)
- Task state register (TSR)

All bits in these registers can be read in User mode; however, only certain bits in these registers can be written while in User mode. Writes to these restricted bits have no effect. Since access to some bits is allowed, there is no exception caused by access to these registers.

##### 9.2.4.2.1 Restrictions on Using CSR in User Mode

The following functions of CSR are restricted when operating in User mode:

- PGIE, PWRD, PCC, and DCC bits cannot be written in User mode. Writes to these bits have no effect.
- GIE and SAT bits are not restricted in User mode, and their behavior is the same as in Supervisor mode.

##### 9.2.4.2.2 Restrictions on Using TSR in User Mode

The GIE and SGIE bits are not restricted in User mode. All other bits are restricted from being written; writes to these bits have no effect.

#### 9.2.4.3 Restricted Instruction Execution in User Mode

Certain instructions are not available for use in User mode. An attempt to execute one of these instructions results in an exception. The opcode exception (OPX) and privilege exception (PRX) bits are set in the internal exception report register (IERR) when this exception occurs. The following instructions are restricted in User mode:

- B IRP
- B NRP
- IDLE

## 9.3 Interrupts and Exception Handling

As described in Section 9.2.2, mode switching mostly occurs for interrupt or exception handling. This section describes the execution mode behavior of interrupt and exception processing.

### 9.3.1 Inhibiting Interrupts in User Mode

The GIE bit in the control status register (CSR) can be used to inhibit interrupts in User mode. This allows a usage model where User mode programs may be written to conform to a required level of interruptibility while still protecting segments of code that cannot be interrupted safely. Nonconforming behavior may be detected at the system level, and control can be taken from the User mode program by asserting the EXCEP input to the CPU.

### 9.3.2 Privilege and Interrupts

When an interrupt occurs, the interrupted execution mode and other key information is saved in the interrupt task state register (ITSR). The CXM bit in the task state register (TSR) is set to indicate that the current execution mode is Supervisor mode. Explicit (**MVC**) writes to TSR are completed before being saved to ITSR.

The interrupt handler begins executing at the address formed by adding the offset for the particular interrupt event to the value of the interrupt service table pointer register (ISTP). The return from interrupt (**B IRP**) instruction restores the saved values from ITSR into TSR, causing execution to resume in the execution mode of the interrupted program.

The transition to the restored execution mode is coincident to the execution of the return branch target. Execution of instructions in the delay slot of the branch are in Supervisor mode.

### 9.3.3 Privilege and Exceptions

When an exception occurs, the interrupted execution mode and other key information is saved in the NMI/exception task state register (NTSR). The CXM bit the task state register (TSR) is set to indicate that the current execution mode is Supervisor mode. Explicit (**MVC**) writes to TSR are completed before saved to ITSR.

The exception handler begins executing at the address formed by adding the offset for the exception/NMI event to the value of the interrupt service table pointer register (ISTP). The return from exception (**B NRP**) instruction restores the saved values from NTSR into TSR.

### 9.3.4 Privilege and Memory Protection

The data and program memory interfaces at the boundary of the CPU include signals indicating the execution mode in which an access was initiated. This information can be used at the system level to raise an exception in the event of an access rights violation.

## 9.4 Operating System Entry

A protected interface is needed so that User mode code can safely enter the operating system to request service.

There is one potential problem with allowing direct calling into the operating system: the caller can choose where to enter the OS and, if allowed to choose any OS location to enter, can:

- Bypass operand checking by OS routines
- Access undocumented interfaces
- Defeat protection
- Corrupt OS data structures by bypassing consistency checks or locking

In short, allowing unrestricted entry into an OS is a very bad idea. Instead, you need to give a very controlled way of entering the operating system and switching from User mode to Supervisor mode. The mechanism chosen is essentially an exception, where the handler decodes the requested operation and dispatches to a Supervisor mode routine that validates the arguments and services the request.

### 9.4.1 Entering User Mode from Supervisor Mode

There are two reasons that the CPU might need to enter User mode while operating in Supervisor mode:

- To spawn a User mode task
- To return to User mode after an interrupt or exception<sup>3</sup>

Both cases are handled by one of two related procedures:

- Place the address in NRP, ensure that the NTSR.CXM bit is set to 1, and execute a **B NRP** instruction to force a context switch to the User mode task.
- Place the desired address in IRP, ensure that the ITSR.CXM bit is set to 1, and execute a **B IRP** instruction to force a context switch to the User mode task.

When returning from an interrupt or exception, the IRP or NRP should already have the correct return address and the ITSR.CXM or NTSR.CXM bit should already be set to 1.

When spawning a user mode task, the appropriate CXM bit and the IRP or NRP will need to be initialized explicitly with the entry point address of the User mode task. In addition, the restricted entry point address register (REP) should be loaded with the desired return address that the User mode task will use when it terminates.

### 9.4.2 Entering Supervisor Mode from User Mode

The operating mode will change from User mode to Supervisor mode in the following cases:

- While processing any interrupt
- While processing an exception

The User mode task can force a change to Supervisor mode by forcing an exception by executing either an **SWE** or **SWENR** instruction.

The **SWE** and **SWENR** instructions both force a software exception. The **SWE** instruction is used when a return from the exception back to the point of the exception is desired. The **SWENR** instruction is used when a return to the User mode routine is not desired.

OS entry and switching from User to Supervisor mode is accomplished by forcing a software exception using either the **SWE** or **SWENR** instructions. See Section [7.5.3](#) for information about software exceptions.

Execution of an **SWE** instruction results in an exception being taken before the next execute packet is processed. The return pointer stored in the nonmaskable interrupt return pointer register (NRP) points to this unprocessed packet. The value of the task state register (TSR) is copied to the NMI/exception task state register (NTSR) at the end of the cycle containing the **SWE** instruction, and the interrupt/exception default value is written to TSR. The **SWE** instruction should not be placed in the delay slots of a branch since all instructions behind the **SWE** instruction in the pipe are annulled. All writes to registers in the pipe from instructions executed before and in parallel with the **SWE** instruction will complete before execution of the exception service routine, therefore, the instructions prior to the **SWE** will complete (along with all their delay slots) before the instructions after the **SWE**.

If the **SWE** instruction is executed while in User mode, the mode is changed to Supervisor mode as part of the exception servicing process. The TSR is copied to NTSR, the return address is placed in the NRP register, and a transfer of control is forced to the NMI/Exception vector pointed to by current value of the ISTP. Any code necessary to interpret a User mode request should reside in the exception service routine. After processing the request the exception handler will return control to the user task by executing a **B NRP** command.

The **SWENR** instruction can also be used to terminate a user mode task. The **SWENR** instruction is similar to the **SWE** instruction except that no provision is made for returning to the user mode task and the transfer of control is to the address pointed to by REP instead of the NMI/exception vector. The supervisor mode should have earlier placed the correct address in REP.

## Appendix A

# Instruction Compatibility

Table A-1 lists the instructions that are common to the C62x, C64x, C64x+, C67x, C67x+, C674x, and C66x DSPs.

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 1 of 8)**

Instruction	C62x DSP	C64x DSP	C64x+ DSP	C67x DSP	C67x+ DSP	C674x DSP	C66x DSP
ABS	✓	✓	✓	✓	✓	✓	✓
ABS2		✓	✓			✓	✓
ABSDP				✓	✓	✓	✓
ABSSP				✓	✓	✓	✓
ADD	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
ADDAB	✓	✓	✓	✓	✓	✓	✓
ADDAD	✓	✓	✓	✓	✓	✓	✓
ADDAH	✓	✓	✓	✓	✓	✓	✓
ADDAW	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
ADDDP				✓	✓	✓	✓
ADDK	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
ADDKPC		✓	✓			✓	✓
ADDSP				✓	✓	✓	✓
ADDSUB			✓			✓	✓
ADDSUB2			✓			✓	✓
ADDU	✓	✓	✓	✓	✓	✓	✓
ADD2	✓	✓	✓	✓	✓	✓	✓
ADD4	✓		✓			✓	✓
AND	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
ANDN	✓		✓			✓	✓
AVG2	✓	✓				✓	✓
AVGU4	✓	✓				✓	✓
B displacement	✓	✓	✓	✓	✓	✓	✓
B register	✓	✓	✓	✓	✓	✓	✓
B IRP	✓	✓	✓	✓	✓	✓	✓
B NRP	✓	✓	✓	✓	✓	✓	✓
BDEC		✓	✓	✓	✓	✓	✓
BITC4	✓	✓				✓	✓

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 2 of 8)**

<b>Instruction</b>	<b>C62x DSP</b>	<b>C64x DSP</b>	<b>C64x+ DSP</b>	<b>C67x DSP</b>	<b>C67x+ DSP</b>	<b>C674x DSP</b>	<b>C66x DSP</b>
BITR	✓	✓ <sup>2</sup>			✓	✓	
BNOP displacement	✓	✓ <sup>1</sup>			✓	✓	
BNOP register	✓	✓			✓	✓	
BPOS	✓	✓			✓	✓	
CALLP		✓ <sup>1</sup>			✓	✓	
CCMATMPY						✓	
CCMATMPYR1						✓	
CCMPY32R1						✓	
CLR	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
CMATMPY						✓	
CMATMPYR1						✓	
CMPEQ	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
CMPEQ2	✓	✓			✓	✓	
CMPEQ4	✓	✓			✓	✓	
CMPEQDP				✓	✓	✓	✓
CMPEQSP				✓	✓	✓	✓
CMPGT	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
CMPGT2	✓	✓			✓	✓	
CMPGTD <sub>P</sub>				✓	✓	✓	✓
CMPGTSP				✓	✓	✓	✓
CMPGTU	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
CMPGTU4	✓	✓			✓	✓	
CMPLT	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
CMPLT2	✓	✓			✓	✓	
CMPLTDP				✓	✓	✓	✓
CMPLTSP				✓	✓	✓	✓
CMPLTU	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
CMPLTU4	✓	✓			✓	✓	
CMPY		✓			✓	✓	
CMPY32R1						✓	
CMPYR		✓			✓	✓	
CMPYR1		✓			✓	✓	
CMPYSP						✓	
CROT270						✓	
CROT90						✓	
DADD						✓	
DADD2						✓	
DADDSP						✓	
DAPYS2						✓	
DAVG2						✓	
DAVGNR2						✓	
DAVGNRU4						✓	
DAVGU4						✓	
DCCMPY						✓	

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 3 of 8)**

<b>Instruction</b>	<b>C62x DSP</b>	<b>C64x DSP</b>	<b>C64x+ DSP</b>	<b>C67x DSP</b>	<b>C67x+ DSP</b>	<b>C674x DSP</b>	<b>C66x DSP</b>
DCCMPYR1						✓	
DCMPEQ2						✓	
DCMPEQ4						✓	
DCMPGT2						✓	
DCMPGTU4						✓	
DCMPY						✓	
DCMPYR1						✓	
DCROT270						✓	
DCROT90						✓	
DDOTP4		✓			✓	✓	
DDOTP4H		✓			✓	✓	
DDOTPH2		✓			✓	✓	
DDOTPH2R		✓			✓	✓	
DDOTPL2		✓			✓	✓	
DDOTPL2R		✓			✓	✓	
DDOTPSU4H		✓			✓	✓	
DEAL	✓	✓			✓	✓	
DINT		✓			✓	✓	
DINTHSP (16-bit)						✓	
DINTHSP (32-bit)						✓	
DINTHSPU						✓	
DINTSPU							
DMAX2						✓	
DMAXU4						✓	
DMIN2						✓	
DMINU4						✓	
DMPY2						✓	
DMPYSP						✓	
DMPYSU4						✓	
DMPYU2						✓	
DMPYU4						✓	
DMV		✓				✓	
DMVD						✓	
DOTP2	✓	✓			✓	✓	
DOTP4H	✓	✓			✓	✓	
DOTPN2	✓	✓			✓	✓	
DOTPNRSU2	✓	✓			✓	✓	
DOTPNRUS2	✓	✓			✓	✓	
DOTPRS2	✓	✓			✓	✓	
DOTPRUS2	✓	✓			✓	✓	
DOTPSU4	✓	✓			✓	✓	
DOTPSU4H	✓	✓			✓	✓	
DOTPUS4	✓	✓			✓	✓	
DOTPU4	✓	✓			✓	✓	

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 4 of 8)**

<b>Instruction</b>	<b>C62x DSP</b>	<b>C64x DSP</b>	<b>C64x+ DSP</b>	<b>C67x DSP</b>	<b>C67x+ DSP</b>	<b>C674x DSP</b>	<b>C66x DSP</b>
DPACK2		✓			✓	✓	
DPACKH2						✓	
DPACKH4						✓	
DPACKHL2						✓	
DPACKL2						✓	
DPACKL4						✓	
DPACKLH2						✓	
DPACKLH4						✓	
DPACKX2		✓			✓	✓	
DPINT			✓	✓	✓	✓	
DPSP			✓	✓	✓	✓	
DPTRUNC			✓	✓	✓	✓	
DSADD						✓	
DSADD2						✓	
DSHL						✓	
DSHL2						✓	
DSHR						✓	
DSHR2						✓	
DSHRU						✓	
DSHRU2						✓	
DSMPY2						✓	
DSPACKU4						✓	
DSPINT						✓	
DSPINTH						✓	
DSSUB						✓	
DSSUB2						✓	
DSUB						✓	
DSUB2						✓	
DSUBSP						✓	
DXPND2						✓	
DXPND4						✓	
EXT	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
EXTU	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
FADDDP						✓	
FADDSP						✓	
FMPYDP						✓	
FSUBDP						✓	
FSUBSP						✓	
GMPY			✓		✓	✓	
GMPY4		✓	✓		✓	✓	
IDLE	✓	✓	✓	✓	✓	✓	✓
INTDP				✓	✓	✓	✓
INTDPU				✓	✓	✓	✓
INTSP				✓	✓	✓	✓

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 5 of 8)**

<b>Instruction</b>	<b>C62x DSP</b>	<b>C64x DSP</b>	<b>C64x+ DSP</b>	<b>C67x DSP</b>	<b>C67x+ DSP</b>	<b>C674x DSP</b>	<b>C66x DSP</b>
INTSPU				✓	✓	✓	✓
LAND							✓
LANDN							✓
LDB and LDB(U)	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
LDB and LDB(U) (15-bit offset)	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
LDW	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
LDH and LDH(U)	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
LDH and LDH(U) (15-bit offset)	✓	✓	✓	✓	✓	✓	✓
LDNDW	✓	✓	✓ <sup>1</sup>			✓	✓
LDNW	✓	✓	✓ <sup>1</sup>			✓	✓
LDW	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
LDW (15-bit offset)	✓	✓	✓	✓	✓	✓	✓
LMBD	✓	✓	✓	✓	✓	✓	✓
LOR							✓
MAX2	✓	✓				✓	✓
MAXU4	✓	✓				✓	✓
MFENCE							✓
MIN2	✓	✓				✓	✓
MINU4	✓	✓				✓	✓
MPY	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MPY2	✓	✓				✓	✓
MPY2IR		✓				✓	✓
MPY32 (32-bit result)		✓				✓	✓
MPY32 (64-bit result)		✓				✓	✓
MPY32SU		✓				✓	✓
MPY32U		✓				✓	✓
MPY32US		✓				✓	✓
MPYDP				✓	✓	✓	✓
MPYH	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MPYHI		✓	✓			✓	✓
MPYHIR		✓	✓			✓	✓
MPYHL	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MPYHLU	✓	✓	✓	✓	✓	✓	✓
MPYHSLU	✓	✓	✓	✓	✓	✓	✓
MPYHSU	✓	✓	✓	✓	✓	✓	✓
MPYHU	✓	✓	✓	✓	✓	✓	✓
MPYHULS	✓	✓	✓	✓	✓	✓	✓
MPYHUS	✓	✓	✓	✓	✓	✓	✓
MPYI				✓	✓	✓	✓
MPYID				✓	✓	✓	✓
MPYIH	✓	✓				✓	✓
MPYIHR	✓	✓				✓	✓
MPYIL	✓	✓				✓	✓
MPYILR	✓	✓				✓	✓

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 6 of 8)**

Instruction	C62x DSP	C64x DSP	C64x+ DSP	C67x DSP	C67x+ DSP	C674x DSP	C66x DSP
MPYLH	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MPYLHU	✓	✓	✓	✓	✓	✓	✓
MPYLI	✓	✓				✓	✓
MPYLIR	✓	✓				✓	✓
MPYLSHU	✓	✓	✓	✓	✓	✓	✓
MPYLUHS	✓	✓	✓	✓	✓	✓	✓
MPYSP				✓	✓	✓	✓
MPYSPDP				✓	✓	✓	✓
MPYSP2DP				✓	✓	✓	✓
MPYSU	✓	✓	✓	✓	✓	✓	✓
MPYSU4	✓	✓				✓	✓
MPYU	✓	✓	✓	✓	✓	✓	✓
MPYU2							✓
MPYU4		✓	✓			✓	✓
MPYUS	✓	✓	✓	✓	✓	✓	✓
MPYUS4	✓	✓				✓	✓
MV	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MVC	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MVD		✓	✓			✓	✓
MVK	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
MVKH/MVKLH	✓	✓	✓	✓	✓	✓	✓
MVKL	✓	✓	✓	✓	✓	✓	✓
NEG	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
NOP	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
NORM	✓	✓	✓	✓	✓	✓	✓
NOT	✓	✓	✓	✓	✓	✓	✓
OR	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
PACK2		✓				✓	✓
PACKH2		✓	✓			✓	✓
PACKH4		✓	✓			✓	✓
PACKHL2		✓	✓			✓	✓
PACKLH2		✓	✓			✓	✓
PACKL4		✓	✓			✓	✓
QMPY32							✓
QMPYSP							✓
QSMPY32R1							✓
RCPDP				✓	✓	✓	✓
RCPSP				✓	✓	✓	✓
RINT		✓				✓	✓
ROTL	✓	✓				✓	✓
RPACK2		✓				✓	✓
RSQRDP				✓	✓	✓	✓
RSQRSP				✓	✓	✓	✓
SADD	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 7 of 8)**

Instruction	C62x DSP	C64x DSP	C64x+ DSP	C67x DSP	C67x+ DSP	C674x DSP	C66x DSP
SADD2		✓	✓			✓	✓
SADDSUB			✓			✓	✓
SADDSUB2			✓			✓	✓
SADDSU2		✓	✓			✓	✓
SADDUS2		✓	✓			✓	✓
SADDU4		✓	✓			✓	✓
SAT	✓	✓	✓	✓	✓	✓	✓
SET	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SHFL		✓	✓			✓	✓
SHFL3			✓			✓	✓
SHL	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SHL2							✓
SHLMB		✓	✓			✓	✓
SHR	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SHR2		✓	✓			✓	✓
SHRMB		✓	✓			✓	✓
SHRU	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SHRU2		✓	✓			✓	✓
SMPY	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SMPYH	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SMPYHL	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SMPYLH	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SMPY2		✓	✓			✓	✓
SMPY32		✓				✓	✓
SPACK2		✓	✓			✓	✓
SPACKU4		✓	✓			✓	✓
SPDP				✓	✓	✓	✓
SPINT				✓	✓	✓	✓
SPKERNEL			✓ <sup>1</sup>			✓	✓
SPKERNELR			✓			✓	✓
SPLOOP			✓ <sup>1</sup>			✓	✓
SPLOOPD			✓ <sup>1</sup>			✓	✓
SPLOOPW			✓			✓	✓
SPMASK			✓ <sup>1</sup>			✓	✓
SPMASKR			✓ <sup>1</sup>			✓	✓
SPTRUNC				✓	✓	✓	✓
SSHLL	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SSHVL	✓	✓				✓	✓
SSHVR	✓	✓				✓	✓
SSUB	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SSUB2			✓			✓	✓
STB	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
STB (15-bit offset)	✓	✓	✓	✓	✓	✓	✓
STDW		✓	✓ <sup>1</sup>			✓	✓

**Table A-1 Instruction Compatibility Between C62x, C64x, C64x+, C67x, C67x+, and C674x DSPs (Part 8 of 8)**

<b>Instruction</b>	<b>C62x DSP</b>	<b>C64x DSP</b>	<b>C64x+ DSP</b>	<b>C67x DSP</b>	<b>C67x+ DSP</b>	<b>C674x DSP</b>	<b>C66x DSP</b>
STH	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
STH (15-bit offset)	✓	✓	✓	✓	✓	✓	✓
STNDW	✓	✓ <sup>1</sup>				✓	✓
STNW	✓	✓ <sup>1</sup>				✓	✓
STW	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
STW (15-bit offset)	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SUB	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SUBAB	✓	✓	✓	✓	✓	✓	✓
SUBABS4	✓	✓				✓	✓
SUBAH	✓	✓	✓	✓	✓	✓	✓
SUBAW	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
SUBC	✓	✓	✓	✓	✓	✓	✓
SUBDP				✓	✓	✓	✓
SUBSP				✓	✓	✓	✓
SUBU	✓	✓	✓	✓	✓	✓	✓
SUB2	✓	✓	✓	✓	✓	✓	✓
SUB4	✓	✓				✓	✓
SWAP2	✓	✓				✓	✓
SWAP4	✓	✓				✓	✓
SWE		✓				✓	✓
SWENR		✓				✓	✓
UNPKBU4							✓
UNPKH2							✓
UNPKHU2							✓
UNPKHU4		✓	✓			✓	✓
UNPKLU4		✓	✓			✓	✓
XOR	✓	✓	✓ <sup>1</sup>	✓	✓	✓	✓
XORMPY		✓				✓	✓
XPND2	✓	✓				✓	✓
XPND4	✓	✓				✓	✓
ZERO	✓	✓	✓	✓	✓	✓	✓

1. Instruction also available in compact form. See section [Section 3.10](#) on page 3-29.

## Mapping Between Instruction and Functional Unit

**Table B-1 Instruction to Functional Unit Mapping**

Instruction	Functional Unit			
	.L Unit	.M Unit	.S Unit	.D Unit
ABS	✓			
ABS2	✓			
ABSDP		✓		
ABSSP		✓		
ADD	✓	✓	✓	✓
ADDAB			✓	
ADDAD			✓	
ADDAH			✓	
ADDAW			✓	
ADDDP	✓	✓		
ADDK			✓	
ADDKPC			✓ <sup>1</sup>	
ADDSP	✓	✓		
ADDSUB	✓			
ADDSUB2	✓			
ADDU	✓			
ADD2	✓		✓	✓
ADD4	✓			
AND	✓	✓	✓	✓
ANDN	✓		✓	✓
AVG2		✓		
AVGU4		✓		
B displacement			✓	
B register			✓ <sup>1</sup>	
B IRP			✓ <sup>1</sup>	
B NRP			✓ <sup>1</sup>	
BDEC			✓	
BITC4		✓		
BITR		✓		

**Table B-1 Instruction to Functional Unit Mapping**

<b>Instruction</b>	<b>Functional Unit</b>			
	<b>.L Unit</b>	<b>.M Unit</b>	<b>.S Unit</b>	<b>.D Unit</b>
BNOP displacement			✓	
BNOP register			✓	
BPOS			✓	
CALLP			✓	
CCMATMPY		✓		
CCMATMPYR1		✓		
CCMPY32R1		✓		
CLR				✓
CMATMPY		✓		
CMATMPYR1		✓		
CMPEQ	✓			
CMPEQ2			✓	
CMPEQ4			✓	
CMPEQDP			✓	
CMPEQSP			✓	
CMPGT	✓			
CMPGT2			✓	
CMPGTDP			✓	
CMPGTSP			✓	
CMPGTU	✓			
CMPGTU4			✓	
CMPLT	✓			
CMPLT2			✓	
CMPLTDP			✓	
CMPLTSP			✓	
CMPLTU	✓			
CMPLTU4			✓	
CMPY		✓		
CMPY32R1		✓		
CMPYR		✓		
CMPYR1		✓		
CMPYSP		✓		
CROT270	✓			
CROT90	✓			
DADD	✓		✓	
DADD2	✓		✓	
DADDSP	✓		✓	
DAPYS2	✓			
DAVG2		✓		
DAVGNR2		✓		
DAVGNRU4		✓		
DAVGU4		✓		
DCCMPY				

**Table B-1 Instruction to Functional Unit Mapping**

Instruction	Functional Unit			
	.L Unit	.M Unit	.S Unit	.D Unit
DCCMPYR1		✓		
DCMPEQ2			✓	
DCMPEQ4			✓	
DCMPGT2			✓	
DCMPGTU4			✓	
DCMPY		✓		
DCMPYR1		✓		
DCROT270	✓			
DCROT90	✓			
DDOTP4		✓		
DDOTP4H				
DDOTP4H2		✓		
DDOTP4H2R		✓		
DDOTPL2		✓		
DDOTPL2R		✓		
DDOTPSU4H				
DEAL		✓		
DINT	No unit			
DINTHSP (16-bit)	✓		✓	
DINTHSP (32-bit)	✓		✓	
DINTHSPU	✓		✓	
DINTSPU				
DMAX2	✓			
DMAXU4	✓			
DMIN2	✓			
DMINU4	✓			
DMPY2		✓		
DMPYSP		✓		
DMPYSU4		✓		
DMPYU2		✓		
DMPYU4		✓		
DMV	✓		✓	
DMVD	✓		✓	
DOTP2		✓		
DOTP4H		✓		
DOTPN2		✓		
DOTPNRSU2		✓		
DOTPNRUS2		✓		
DOTPRS2		✓		
DOTPRUS2		✓		
DOTPSU4		✓		

**Table B-1 Instruction to Functional Unit Mapping**

Instruction	Functional Unit			
	.L Unit	.M Unit	.S Unit	.D Unit
DOTPSU4H		✓		
DOTPUS4		✓		
DOTPU4		✓		
DPACK2	✓			
DPACKH2	✓		✓	
DPACKH4	✓			
DPACKHL2	✓		✓	
DPACKL2	✓		✓	
DPACKL4	✓			
DPACKLH2	✓		✓	
DPACKLH4	✓			
DPACKX2	✓			
DPINT	✓			
DPSP	✓			
DPTRUNC	✓			
DSADD	✓		✓	
DSADD2	✓		✓	
DSHL			✓	
DSHL2			✓	
DSHR			✓	
DSHR2			✓	
DSHRU			✓	
DSHRU2			✓	
DSMPY2		✓		
DSPACKU4			✓	
DSPINT	✓		✓	
DSPINTH	✓		✓	
DSSUB	✓			
DSSUB2	✓			
DSUB	✓		✓	
DSUB2	✓		✓	
DSUBSP	✓		✓	
DXPNRD2		✓		
DXPNRD4		✓		
EXT			✓	
EXTU			✓	
FADDDP	✓		✓	
FADDSP	✓		✓	
FMPYDP		✓		
FSUBDP	✓		✓	
FSUBSP	✓		✓	
GMPY		✓		
GMPY4		✓		

**Table B-1 Instruction to Functional Unit Mapping**

Instruction	Functional Unit			
	.L Unit	.M Unit	.S Unit	.D Unit
IDLE	No unit			
INTDP	✓			
INTDPU	✓			
INTSP	✓		✓	
INTSPU	✓		✓	
LAND	✓			
LANDN	✓			
LDB and LDB(U)				✓
LDB and LDB(U) (15-bit offset)				✓ <sup>2</sup>
LDDW				✓
LDH and LDH(U)				✓
LDH and LDH(U) (15-bit offset)				✓ <sup>2</sup>
LDNDW				✓
LDNW				✓ <sup>2</sup>
LDW				✓
LDW (15-bit offset)				✓ <sup>2</sup>
LMBD	✓			
LOR	✓			
MAX2	✓		✓	
MAXU4	✓			
MFENCE	No unit			
MIN2	✓		✓	
MINU4	✓			
MPY		✓		
MPY2		✓		
MPY2IR		✓		
MPY32 (32-bit result)		✓		
MPY32 (64-bit result)		✓		
MPY32SU		✓		
MPY32U		✓		
MPY32US		✓		
MPYDP		✓		
MPYH		✓		
MPYHI		✓		
MPYHIR		✓		
MPYHL		✓		
MPYHLU		✓		
MPYHSU		✓		
MPYHU		✓		
MPYHULS		✓		
MPYHUS		✓		

**Table B-1 Instruction to Functional Unit Mapping**

Instruction	Functional Unit			
	.L Unit	.M Unit	.S Unit	.D Unit
MPYI		✓		
MPYID		✓		
MPYIH		✓		
MPYIHR		✓		
MPYIL		✓		
MPYILR		✓		
MPYLH		✓		
MPYLHU		✓		
MPYLI		✓		
MPYLIR		✓		
MPYLSHU		✓		
MPYLUHS		✓		
MPYSP		✓		
MPYSPDP		✓		
MPYSP2DP		✓		
MPYSU		✓		
MPYSU4		✓		
MPYU		✓		
MPYU2		✓		
MPYU4		✓		
MPYUS		✓		
MPYUS4		✓		
MV	✓		✓	✓
MVC			✓ <sup>1</sup>	
MVD		✓		
MVK	✓		✓	✓
MVKH/MVKLH			✓	
MVKL			✓	
NEG	✓		✓	
NOP	No unit			
NORM	✓			
NOT	✓		✓	
OR	✓		✓	✓
PACK2	✓		✓	
PACKH2	✓		✓	
PACKH4	✓			
PACKHL2	✓		✓	
PACKLH2	✓		✓	
PACKL4	✓			
QMPY32	✓			
QMPYSP	✓			
QSMPY32R1	✓			
RCPDP			✓	

**Table B-1 Instruction to Functional Unit Mapping**

<b>Instruction</b>	<b>Functional Unit</b>			
	<b>.L Unit</b>	<b>.M Unit</b>	<b>.S Unit</b>	<b>.D Unit</b>
RCPSP			✓	
RINT	No unit			
ROTL		✓		
RPACK2			✓	
RSQRDP			✓	
RSQRSP			✓	
SADD	✓		✓	
SADD2			✓	
SADDSUB	✓			
SADDSUB2	✓			
SADDSU2			✓	
SADDUS2			✓	
SADDU4			✓	
SAT	✓			
SET			✓	
SHFL		✓		
SHFL3	✓			
SHL			✓	
SHL2			✓	
SHLMB	✓		✓	
SHR			✓	
SHR2			✓	
SHRMB	✓		✓	
SHRU			✓	
SHRU2			✓	
SMPY		✓		
SMPYH		✓		
SMPYHL		✓		
SMPYLH		✓		
SMPY2		✓		
SMPY32		✓		
SPACK2			✓	
SPACK4			✓	
SPDP			✓	
SPINT	✓		✓	
SPKERNEL	No unit			
SPKERNELR	No unit			
SPLOOP	No unit			
SPLOOPD	No unit			
SPLOOPW	No unit			
SPMASK	No unit			
SPMASKR	No unit			
SPTRUNC	✓			

**Table B-1 Instruction to Functional Unit Mapping**

Instruction	Functional Unit			
	.L Unit	.M Unit	.S Unit	.D Unit
SSH <sub>L</sub>			✓	
SSHVL		✓		
SSHVR		✓		
SSUB	✓			
SSUB2	✓			
STB				✓
STB (15-bit offset)				✓ <sup>2</sup>
STDW			✓	
STH			✓	
STH (15-bit offset)			✓ <sup>2</sup>	
STNDW			✓	
STNW			✓	
STW			✓	
STW (15-bit offset)			✓ <sup>2</sup>	
SUB	✓		✓	✓
SUBAB				✓
SUBABS4	✓			
SUBAH				✓
SUBAW				✓
SUBC	✓			
SUBDP	✓		✓	
SUBSP	✓		✓	
SUBU	✓			
SUB2	✓		✓	✓
SUB4	✓			
SWAP2	✓		✓	
SWAP4	✓			
SWE	No unit			
SWENR	No unit			
UNPKBU4	✓		✓	
UNPKH2	✓		✓	
UNPKHU2	✓		✓	
UNPKHU4	✓		✓	
UNPKLU4	✓		✓	
XOR	✓		✓	✓
XORMPY		✓		
XPND2		✓		
XPND4		✓		
ZERO	✓		✓	✓

1. .S2 only.

2. .D2 only.

## .D Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .D functional unit and illustrates the opcode maps for these instructions.

### C.1 Instructions Executing in the .D Functional Unit

Table C-1 lists the instructions that execute in the .D functional unit.

**Table C-1 Instructions Executing in the .D Functional Unit**

Instruction	Instruction
ADD	OR
ADDAB	STB
ADDAD	STB <sup>1</sup> (15-bit offset)
ADDAH	STDW
ADDAW	STH
ADD2	STH <sup>1</sup> (15-bit offset)
AND	STNDW
ANDN	STNW
LDB and LDB(U)	STW
LDB and LDB(U) <sup>1</sup> (15-bit offset)	STW <sup>1</sup> (15-bit offset)
LDW	SUB
LDH and LDH(U)	SUBAB
LDH and LDH(U) <sup>1</sup>	SUBAH
LDNDW	SUBAW
LDNW	SUB2
LDW	XOR
LDW <sup>1</sup> (15-bit offset)	ZERO
MV	
MVK	

1. D2 only

## C.2 Opcode Map Symbols and Meanings

Table C-2 lists the symbols and meanings used in the opcode maps.

**Table C-2 .D Unit Opcode Map Symbol Definitions**

Symbol	Meaning
<i>baseR</i>	base address register
<i>creg</i>	3-bit field specifying a conditional register
<i>dst</i>	destination. For compact instructions, <i>dst</i> is coded as an offset from either A16 or B16 depending on the value of the <i>t</i> bit.
<i>dw</i>	doubleword; 0 = word, 1 = doubleword
<i>ld/st</i>	load or store; 0 = store, 1 = load
<i>mode</i>	addressing mode, see <a href="#">Table C-3</a> on page C-3
<i>na</i>	nonaligned; 0 = aligned, 1 = nonaligned
<i>offsetR</i>	register offset
<i>op</i>	opfield; field within opcode that specifies a unique instruction
<i>p</i>	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
<i>ptr</i>	offset from either A4-A7 or B4-B7 depending on the value of the <i>s</i> bit. The <i>ptr</i> field is the 2 least-significant bits of the <i>src2</i> ( <i>baseR</i> ) field—bit 2 of register address is forced to 1.
<i>r</i>	LDDW/LDNDW/LDNW instruction
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B. For compact instructions, side of base address ( <i>ptr</i> ) register; 0 = side A, 1 = side B.
<i>src</i>	source. For compact instructions, <i>src</i> is coded as an offset from either A16 or B16 depending on the value of the <i>t</i> bit.
<i>src1</i>	source 1
<i>src2</i>	source 2
<i>sz</i>	data size select; 0 = primary size, 1 = secondary size (see “Expansion Field in Compact Header Word” on page 31)
<i>t</i>	side of source/destination ( <i>src/dst</i> ) register; 0 = side A, 1 = side B
<i>ucst<sub>n</sub></i>	bit <i>n</i> of the unsigned constant field
<i>x</i>	cross path for <i>src2</i> ; 0 = do not use cross path, 1 = use cross path
<i>y</i>	.D1 or .D2 unit; 0 = .D1 unit, 1 = .D2 unit
<i>z</i>	test for equality with zero or nonzero

**Table C-3 Address Generator Options for Load/Store**

<b>mode Field</b>	<b>Syntax</b>				<b>Modification Performed</b>
0 0 0 0	*-R[ucst5]				Negative offset
0 0 0 1	*+R[ucst5]				Positive offset
0 1 0 0	*-R[offsetR]				Negative offset
0 1 0 1	*+R[offsetR]				Positive offset
1 0 0 0	*- -R[ucst5]				Predecrement
1 0 0 1	*++R[ucst5]				Preincrement
1 0 1 0	*R- -[ucst5]				Postdecrement
1 0 1 1	*R++[ucst5]				Postincrement
1 1 0 0	*--R[offsetR]				Predecrement
1 1 0 1	*++R[offsetR]				Preincrement
1 1 1 0	*R- -[offsetR]				Postdecrement
1 1 1 1	*R++[offsetR]				Postincrement

### C.3 32-Bit Opcode Maps

The CPU 32-bit opcodes used in the .D unit are mapped in [Figure C-1](#) through [Figure C-7](#).

**Figure C-1 1 or 2 Sources Instruction Format**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>	<i>dst</i>			<i>src2</i>	<i>src1</i>	
3	1	5			5	5	
13	12				7	6	5
<i>src1</i>	<i>op</i>			1	0	0	0
5	6			1	1	<i>s</i>	<i>p</i>
1	1				0	0	1

**Figure C-2 Extended .D Unit 1 or 2 Sources Instruction Format**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>	<i>dst</i>			<i>src2</i>	<i>src1</i>	
3	1	5			5	5	
13	12	11	10	9	6	5	4
<i>src1</i>	x	1	0	<i>op</i>		1	1
5	1	4			0	0	0
1	1				1	1	1

**Figure C-3 Load/Store Basic Operations**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>	<i>src/dst</i>			<i>baseR</i>	<i>offsetR</i>	
3	1	5			5	5	
13	12				9	8	7
<i>offsetR</i>	<i>mode</i>			r	y	<i>op</i>	
5	4			1	1	3	1
1	1				0	1	1

**Figure C-4 Load Nonaligned Doubleword Instruction Format**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>	<i>src/dst</i>			<i>baseR</i>		
3	1	5			5	5	
13	12	<i>mode</i>			1	0	1
5	4	1 y 0 1 0 0 1 s p			1	0	1 1

**Figure C-5 Store Nonaligned Doubleword Instruction Format**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>	<i>src/dst</i>			<i>baseR</i>		
3	1	5			5	5	
13	12	<i>mode</i>			1 y 1 1 1 0 1 s p	1	0
5	4	1 1 1 0 1 s p			1	0	1 1

**Figure C-6 Load/Store Long-Immediate Operations**

31	29	28	27	23	22
<i>creg</i>	<i>z</i>	<i>dst</i>		<i>offsetR</i>	
3	1	5		15	
15	8	7	6	4	3 2 1 0
offsetR	<i>y</i>	<i>op</i>		1 1 s p	1 1

**Figure C-7 ADDA Long-Immediate Operations**

31	30	29	28	27	23	22
0	0	0	1	<i>dst</i>		<i>offsetR</i>
5	15		15		15	
15	8	7	6	4	3 2 1 0	
offsetR	<i>y</i>	<i>op</i>		1 1 s p	1 1	

## C.4 16-Bit Opcode Maps

The CPU 16-bit opcodes used in the .D unit for compact instructions are mapped in [Figure C-8](#) through [Figure C-21](#). See Section 3.10 “[Compact Instructions on the CPU](#)” on page 3-29 for more information about compact instructions.

**Figure C-8 Doff4 Instruction Format**

15	13	12	11	10	9	8	7	6	4	3	2	1	0
$ucst_{2:0}$		<i>t</i>	$ucst_3$	0	<i>sz</i>	<i>ptr</i>		<i>src/dst</i>		<i>ld/st</i>	1	0	<i>s</i>
3		1	1		1	2		3		1			1

DSZ	<i>sz</i>	<i>ld/st</i>	Mnemonic
0 x x	0	0	<b>STW</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
0 x x	0	1	<b>LDW</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
0 0 0	1	0	<b>STB</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
0 0 0	1	1	<b>LDBU</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
0 0 1	1	0	<b>STB</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
0 0 1	1	1	<b>LDB</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
0 1 0	1	0	<b>STH</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
0 1 0	1	1	<b>LDHU</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
0 1 1	1	0	<b>STH</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
0 1 1	1	1	<b>LDH</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
1 0 0	1	0	<b>STW</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
1 0 0	1	1	<b>LDW</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
1 0 1	1	0	<b>STB</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
1 0 1	1	1	<b>LDB</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
1 1 0	1	0	<b>STNW</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
1 1 0	1	1	<b>LDNW</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
1 1 1	1	0	<b>STH</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
1 1 1	1	1	<b>LDH</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>

**Figure C-9 Doff4DW Instruction Format**

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$ucst_{2:0}$		<i>t</i>	$ucst_3$	0	<i>sz</i>	<i>ptr</i>		<i>src/dst</i>	<i>na</i>	<i>ld/st</i>	1	0	<i>s</i>	
3		1	1		1	2		2		1	1		1	

NOTE: *src/dst* register address formed from op6:op5:0 (even registers)

DSZ	<i>sz</i>	<i>ld/st</i>	<i>na</i>	Mnemonic
1 x x	0	0	0	<b>STDW</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i>
1 x x	0	1	0	<b>LDDW</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i>
1 x x	0	0	1	<b>STNDW</b> (.unit) <i>src</i> , * <i>ptr[ucst4]</i> (ucst4 unscaled only)
1 x x	0	1	1	<b>LDNDW</b> (.unit) * <i>ptr[ucst4]</i> , <i>dst</i> (ucst4 unscaled only)

**Figure C-10 Dind Instruction Format**

15	13	12	11	10	9	8	7	6	4	3	2	1	0
<i>src1</i>		<i>t</i>	0	1	<i>sz</i>	<i>ptr</i>		<i>src/dst</i>		<i>ld/st</i>	1	0	<i>s</i>
3		1		1		2		3		1			1

Opcode map field used...	For operand type...
<i>src1</i>	sint

## C.4 16-Bit Opcode Maps

## Appendix C—D Unit Instructions and Opcode Maps

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>Mnemonic</b>
0 x x	0	0	<b>STW</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
0 x x	0	1	<b>LDW</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
0 0 0	1	0	<b>STB</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
0 0 0	1	1	<b>LDBU</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
0 0 1	1	0	<b>STB</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
0 0 1	1	1	<b>LDB</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
0 1 0	1	0	<b>STH</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
0 1 0	1	1	<b>LDHU</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
0 1 1	1	0	<b>STH</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
0 1 1	1	1	<b>LDH</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
1 0 0	1	0	<b>STW</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
1 0 0	1	1	<b>LDW</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
1 0 1	1	0	<b>STB</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
1 0 1	1	1	<b>LDB</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
1 1 0	1	0	<b>STNW</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
1 1 0	1	1	<b>LDNW</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
1 1 1	1	0	<b>STH</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
1 1 1	1	1	<b>LDH</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>

**Figure C-11 DindDW Instruction Format**

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>		<i>t</i>	0	1	<i>sz</i>	<i>ptr</i>		<i>src/dst</i>		<i>na</i>		<i>Id/st</i>	1	0
3		1			1		2		2	1	1		1	1

NOTE: *src/dst* register address formed from op6:op5:0 (even registers)

<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src1</i>	sint

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>na</b>	<b>Mnemonic</b>
1 x x	0	0	0	<b>STDW</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ]
1 x x	0	1	0	<b>LDDW</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i>
1 x x	0	0	1	<b>STNDW</b> (.unit) <i>src</i> , *ptr[ <i>src1</i> ] (src1 unscaled only)
1 x x	0	1	1	<b>LDNDW</b> (.unit) *ptr[ <i>src1</i> ], <i>dst</i> (src1 unscaled only)

**Figure C-12 Dinc Instruction Format**

15	14	13	12	11	10	9	8	7	6	4	3	2	1	0
0	0	<i>ucst<sub>0</sub></i>	<i>t</i>	1	1	<i>sz</i>	<i>ptr</i>		<i>src/dst</i>		<i>Id/st</i>	1	0	<i>s</i>
		1	1			1	2			3	1		1	1

NOTE: *ucst2* = *ucst<sub>0</sub>* + 1

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>Mnemonic</b>
0 x x	0	0	<b>STW</b> (.unit) <i>src</i> , *ptr[ <i>ucst2</i> ]++
0 x x	0	1	<b>LDW</b> (.unit) *ptr[ <i>ucst2</i> ]++, <i>dst</i>
0 0 0	1	0	<b>STB</b> (.unit) <i>src</i> , *ptr[ <i>ucst2</i> ]++

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>Mnemonic</b>
0 0 0	1	1	<b>LDBU</b> (.unit) *ptr[ucst2]++, dst
0 0 1	1	0	<b>STB</b> (.unit) src, *ptr[ucst2]++
0 0 1	1	1	<b>LDB</b> (.unit) *ptr[ucst2]++, dst
0 1 0	1	0	<b>STH</b> (.unit) src, *ptr[ucst2]++
0 1 0	1	1	<b>LDHU</b> (.unit) *ptr[ucst2]++, dst
0 1 1	1	0	<b>STH</b> (.unit) src, *ptr[ucst2]++
0 1 1	1	1	<b>LDH</b> (.unit) *ptr[ucst2]++, dst
1 0 0	1	0	<b>STW</b> (.unit) src, *ptr[ucst2]++
1 0 0	1	1	<b>LDW</b> (.unit) *ptr[ucst2]++, dst
1 0 1	1	0	<b>STB</b> (.unit) src, *ptr[ucst2]++
1 0 1	1	1	<b>LDB</b> (.unit) *ptr[ucst2]++, dst
1 1 0	1	0	<b>STNW</b> (.unit) src, *ptr[ucst2]++
1 1 0	1	1	<b>LDNW</b> (.unit) *ptr[ucst2]++, dst
1 1 1	1	0	<b>STH</b> (.unit) src, *ptr[ucst2]++
1 1 1	1	1	<b>LDH</b> (.unit) *ptr[ucst2]++, dst

**Figure C-13 DincDW Instruction Format**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	ucst <sub>0</sub>	t	1	1	sz	ptr		src/dst	na	Id/st	1	0	s	
			1	1		1	2		2	1	1	1	1		1

NOTES:

$$\text{ucst2} = \text{ucst}_0 + 1$$

**src/dst** register address formed from op6:op5:0 (even registers)

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>na</b>	<b>Mnemonic</b>
1 x x	0	0	0	<b>STDW</b> (.unit) src, *ptr[ucst2]++
1 x x	0	1	0	<b>LDDW</b> (.unit) *ptr[ucst2]++, dst
1 x x	0	0	1	<b>STNDW</b> (.unit) src, *ptr[ucst2]++ (ucst2 scaled only)
1 x x	0	1	1	<b>LDNDW</b> (.unit) *ptr[ucst2]++, dst (ucst2 scaled only)

**Figure C-14 Ddec Instruction Format**

15	14	13	12	11	10	9	8	7	6	4	3	2	1	0
0	1	ucst <sub>0</sub>	t	1	1	sz	ptr		src/dst		Id/st	1	0	s
			1	1		1	2		3	1	1	1	1	

NOTE:  $\text{ucst2} = \text{ucst}_0 + 1$ 

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>Mnemonic</b>
0 x x	0	0	<b>STW</b> (.unit) src, *--ptr[ucst2]
0 x x	0	1	<b>LDW</b> (.unit) *--ptr[ucst2], dst
0 0 0	1	0	<b>STB</b> (.unit) src, *--ptr[ucst2]
0 0 0	1	1	<b>LDBU</b> (.unit) *--ptr[ucst2], dst
0 0 1	1	0	<b>STB</b> (.unit) src, *--ptr[ucst2]
0 0 1	1	1	<b>LDB</b> (.unit) *--ptr[ucst2], dst
0 1 0	1	0	<b>STH</b> (.unit) src, *--ptr[ucst2]
0 1 0	1	1	<b>LDHU</b> (.unit) *--ptr[ucst2], dst

## C.4 16-Bit Opcode Maps

## Appendix C—.D Unit Instructions and Opcode Maps

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>Mnemonic</b>
0	1	1	<b>STH</b> (.unit) <i>src</i> , *--ptr[ucst2]
0	1	1	<b>LDH</b> (.unit) *--ptr[ucst2], <i>dst</i>
1	0	0	<b>STW</b> (.unit) <i>src</i> , *--ptr[ucst2]
1	0	0	<b>LDW</b> (.unit) *--ptr[ucst2], <i>dst</i>
1	0	1	<b>STB</b> (.unit) <i>src</i> , *--ptr[ucst2]
1	0	1	<b>LDB</b> (.unit) *--ptr[ucst2], <i>dst</i>
1	1	0	<b>STNW</b> (.unit) <i>src</i> , *--ptr[ucst2]
1	1	0	<b>LDNW</b> (.unit) *--ptr[ucst2], <i>dst</i>
1	1	1	<b>STH</b> (.unit) <i>src</i> , *--ptr[ucst2]
1	1	1	<b>LDH</b> (.unit) *--ptr[ucst2], <i>dst</i>

**Figure C-15 DdecDW Instruction Format**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	<i>ucst<sub>0</sub></i>	<i>t</i>	1	1	<i>sz</i>	<i>ptr</i>			<i>src/dst</i>	<i>na</i>	<i>ld/st</i>	1	0	<i>s</i>
		1	1			1	2			2	1	1	1		1

NOTES:

$$\text{ucst2} = \text{ucst}_0 + 1$$

*src/dst* register address formed from op6:op5:0 (even registers)

<b>DSZ</b>	<b>sz</b>	<b>Id/st</b>	<b>na</b>	<b>Mnemonic</b>
1	x	x	0	<b>STDW</b> (.unit) <i>src</i> , *--ptr[ucst2]
1	x	x	0	<b>LDDW</b> (.unit) *--ptr[ucst2], <i>dst</i>
1	x	x	0	<b>STNDW</b> (.unit) <i>src</i> , *--ptr[ucst2] (ucst2 scaled only)
1	x	x	1	<b>LDNDW</b> (.unit) *--ptr[ucst2], <i>dst</i> (ucst2 scaled only)

**Figure C-16 Dstk Instruction Format**

15	14	13	12	11	10	9	7	6	4	3	2	1	0
1	<i>ucst<sub>1-0</sub></i>		<i>t</i>	1	1	<i>ucst<sub>4-2</sub></i>		<i>src/dst</i>		<i>ld/st</i>	1	0	<i>s</i>
	2		1			3		3		1			1

NOTE: ptr = B15, s = 1

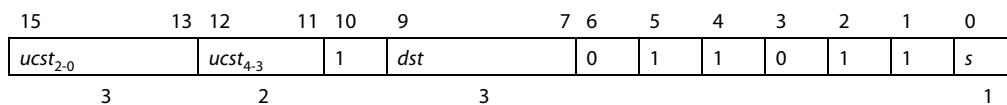
<b>Id/st</b>	<b>Mnemonic</b>
0	<b>STW</b> (.unit) <i>src</i> , *B15[ucst5]
1	<b>LDW</b> (.unit) *B15[ucst5], <i>dst</i>

**Figure C-17 Dx2op Instruction Format**

15	13	12	11	10	9	7	6	5	4	3	2	1	0
<i>src1/dst</i>		x	<i>op</i>	0	<i>src2</i>		0	1	1	0	1	1	<i>s</i>
3		1	1		3								1

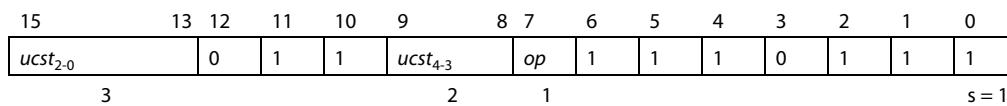
<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src1/dst</i>	sint
<i>src2</i>	xsint

<b>op</b>	<b>Mnemonic</b>
0	<b>ADD</b> (.unit) $src1, src2, dst$ ( $src1 = dst$ )
1	<b>SUB</b> (.unit) $src1, src2, dst$ ( $src1 = dst, dst = src1 - src2$ )

**Figure C-18 Dx5 Instruction Format**NOTE:  $src2 = B15$ 

<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>dst</i>	sint

<b>Mnemonic</b>
<b>ADDAW</b> (.unit)B15, $ucst5, dst$

**Figure C-19 Dx5p Instruction Format**NOTE:  $src2 = dst = B15$ 

<b>op</b>	<b>Mnemonic</b>
0	<b>ADDAW</b> (.unit)B15, $ucst5, B15$
1	<b>SUBAW</b> (.unit)B15, $ucst5, B15$

**Figure C-20 Dx1 Instruction Format**

<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src2/dst</i>	sint

<b>op</b>	<b>Mnemonic</b>
0 0 0	see LSDx1, <a href="#">Figure G-4</a>
0 0 1	see LSDx1, <a href="#">Figure G-4</a>
0 1 0	Reserved
0 1 1	<b>SUB</b> (.unit) $src2, 1, dst$ ( $src2 = dst, dst = src2 - 1$ )
1 0 0	Reserved
1 0 1	see LSDx1, <a href="#">Figure G-4</a>
1 1 0	Reserved
1 1 1	see LSDx1, <a href="#">Figure G-4</a>

**Figure C-21 Dpp Instruction Format**

15	14	13	12	11	10		7	6	5	4	3	2	1	0
<i>dw</i>	<i>dl/st</i>	<i>ucst<sub>0</sub></i>	<i>t</i>	0	<i>src/dst</i>		1	1	1	0	1	1	1	

1 1 1 1 1 4

## NOTES:

*ptr* = B15

$$\mathbf{ucst2} = \mathbf{ucst_0} + 1$$

*src/dst* is from A0-A15, B0-B15

RS header bit is ignored

<b><i>dw</i></b>	<b><i>Id/st</i></b>	<b>Mnemonic</b>
0	0	<b>STW</b> (.unit) <i>src,*B15-[ucst2]</i>
0	1	<b>LDW</b> (.unit)*++B15[ <i>ucst2</i> ], <i>dst</i>
1	0	<b>STDW</b> (.unit) <i>src,*B15-[ucst2]</i>
1	1	<b>LDDW</b> (.unit)*++B15[ <i>ucst2</i> ], <i>dst</i>

<b><i>t</i></b>	<b><i>src/dst</i></b>	<b>Source/Destination</b>	<b><i>t</i></b>	<b><i>src/dst</i></b>	<b>Source/Destination</b>
0	0000	A0	1	0000	B0
0	0001	A1	1	0001	B1
0	0010	A2	1	0010	B2
0	0011	A3	1	0011	B3
0	0100	A4	1	0100	B4
0	0101	A5	1	0101	B5
0	0110	A6	1	0110	B6
0	0111	A7	1	0111	B7
0	1000	A8	1	1000	B8
0	1001	A9	1	1001	B9
0	1010	A10	1	1010	B10
0	1011	A11	1	1011	B11
0	1100	A12	1	1100	B12
0	1101	A13	1	1101	B13
0	1110	A14	1	1110	B14
0	1111	A15	1	1111	B15

## Appendix D

# .L Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .L functional unit and illustrates the opcode maps for these instructions.

### D.1 Instructions Executing in the .L Functional Unit

Table D-1 lists the instructions that execute in the .L functional unit.

**Table D-1 Instructions Executing in the .L Functional Unit**

Instruction	Instruction	Instruction	Instruction
ABS	DPACK2	NORM	SPTRUNC
ABS2	DPACKX2	NOT	SSUB
ADD	DPINT	OR	SSUB2
ADDDP	DPSP	PACK2	SUB
ADDSP	DPTRUNC	PACKH2	SUBABS4
ADDSUB	INTDP	PACKH4	SUBC
ADDSUB2	INTDPU	PACKHL2	SUBDP
ADDU	INTSP	PACKLH2	SUBSP
ADD2	INTSPU	PACKL4	SUBU
ADD4	LMBD	SADD	SUB2
AND	MAX2	SADDSUB	SUB4
ANDN	MAXU4	SADDSUB2	SWAP2
CMPEQ	MIN2	SAT	SWAP4
CMPGT	MINU4	SHFL3	UNPKHU4
CMPGTU	MV	SHLMB	UNPKLU4
CMPLT	MVK	SHRMB	XOR
CMPLTU	NEG	SPINT	ZERO

### D.2 Opcode Map Symbols and Meanings

Table D-2 on page D-2 lists the symbols and meanings used in the opcode maps.

**Table D-2 .L Unit Opcode Map Symbol Definitions**

<b>Symbol</b>	<b>Meaning</b>
<i>creg</i>	3-bit field specifying a conditional register
<i>cstn</i>	n-bit constant field
<i>dst</i>	destination
<i>op</i>	opfield; field within opcode that specifies a unique instruction
<i>op<sub>n</sub></i>	bit n of the opfield
<i>p</i>	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B
<i>scst<sub>n</sub></i>	bit n of the signed constant field
<i>sn</i>	sign
<i>src1</i>	source 1
<i>src2</i>	source 2
<i>ucstn</i>	n-bit unsigned constant field
<i>x</i>	cross path for <i>src2</i> ; 0 = do not use cross path, 1 = use cross path
<i>z</i>	test for equality with zero or nonzero

## D.3 32-Bit Opcode Maps

The CPU 32-bit opcodes used in the .L unit are mapped in [Figure D-1](#) through [Figure D-3](#).

**Figure D-1 1 or 2 Sources Instruction Format****Figure D-2 1 or 2 Sources, Nonconditional Instruction Format**

**Figure D-3 Unary Instruction Format**

31	29	28	27						23	22						18	17
				dst							src2						op
3		1		5					5		5					5	
	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
																1	1
	op	x	0	0	1	1	0	1	0	1	1	0	s	p			
	5		1														

## D.4 16-Bit Opcode Maps

The CPU 16-bit opcodes used in the .L unit for compact instructions are mapped in [Figure D-4](#) through [Figure D-11](#). See Section 3.10 “[Compact Instructions on the CPU](#)” on page 3-29 for more information about compact instructions.

**Figure D-4 L3 Instruction Format**

15	13	12	11	10	9		7	6		4	3	2	1	0		
						src2					dst					
3		1	1			3		3			0	0	0	s		1

Opcode map field used...	For operand type...
src1	sint
src2	xsint
dst	sint

op	SAT	Mnemonic
0	0	<b>ADD</b> (.unit) src1, src2, dst
0	1	<b>SADD</b> (.unit) src1, src2, dst
1	0	<b>SUB</b> (.unit) src1, src2, dst (dst = src1 - src2)
1	1	<b>SSUB</b> (.unit) src1, src2, dst (dst = src1 - src2)

**Figure D-5 L3i Instruction Format**

15	13	12	11	10	9		7	6		4	3	2	1	0		
						src2					dst					
3		1	1			3		3			0	0	0	s		1

Opcode map field used...	For operand type...
src2	xsint

sn	cst3	32-Bit Opcode cst Equivalent		sn	cst3	32-Bit Opcode cst Equivalent	
		scst5	Decimal Value			scst5	Decimal Value
0	000	01000	8	1	000	11000	-8
0	001	00001	1	1	001	11001	-7
0	010	00010	2	1	010	11010	-6
0	011	00011	3	1	011	11011	-5
0	100	00100	4	1	100	11100	-4

32-Bit Opcode cst Equivalent				32-Bit Opcode cst Equivalent			
<i>sn</i>	<i>cst3</i>	<i>scst5</i>	Decimal Value	<i>sn</i>	<i>cst3</i>	<i>scst5</i>	Decimal Value
0	101	00101	5	1	101	11101	-3
0	110	00110	6	1	110	11110	-2
0	111	00111	7	1	111	11111	-1

Mnemonic
<b>ADD (.unit) scst5, src2, dst</b>

**Figure D-6 LtbD Instruction Format**

15	14	13	12	11	10	9				7	6	5	4	3	2	1	0
			x		0	src2								1	0	0	s
					1					3					1		1

Opcode map field used...	For operand type...
src2	xsint

**Figure D-7 L2c Instruction Format**

15	13	12	11	10	9				7	6	5	4	3	2	1	0
src1		x	op <sub>2</sub>	1	src2				op <sub>1-0</sub>		dst	1	0	0	s	
		3		1	1				3		2	1				1

NOTE: dst = A0, A1 or B0, B1 as selected by dst and s

Opcode map field used...	For operand type...
src1	sint
src2	xsint

<i>op</i>	Mnemonic
0 0 0	<b>AND (.unit) src1, src2, dst</b>
0 0 1	<b>OR (.unit) src1, src2, dst</b>
0 1 0	<b>XOR (.unit) src1, src2, dst</b>
0 1 1	<b>CMPEQ (.unit) src1, src2, dst</b>
1 0 0	<b>CMPLT (.unit) src1, src2, dst</b> (dst = src1 < src2 , signed compare)
1 0 1	<b>CMPGT (.unit) src1, src2, dst</b> (dst = src1 > src2 , signed compare)
1 1 0	<b>CMPLTU (.unit) src1, src2, dst</b> (dst = src1 < src2 , unsigned compare)
1 1 1	<b>CMPGTU (.unit) src1, src2, dst</b> (dst = src1 > src2 , unsigned compare)

**Figure D-8 Lx5 Instruction Format**

15	13	12	11	10	9				7	6	5	4	3	2	1	0
scst <sub>2-0</sub>		scst <sub>4-3</sub>	1	dst					0	1	0	0	1	1	s	
		3		2					3							1

Opcode map field used...	For operand type...
dst	sint

Mnemonic
<b>MVK (.unit) scst5, dst</b>

**Figure D-9 Lx3c Instruction Format**

15	13	12	11	10	9	7	6	5	4	3	2	1	0
<i>ucst3</i>		0	<i>dst</i>	0	<i>src2</i>		0	1	0	0	1	1	<i>s</i>
3			1			3							1

NOTE: *dst* = A0, A1 or B0, B1 as selected by *dst* and *s*

Opcode map field used...	For operand type...
<i>src2</i>	sint

Mnemonic
<b>CMPEQ (.unit) ucst3, src2, dst</b>

**Figure D-10 Lx1c Instruction Format**

15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
<i>op</i>		<i>ucst1</i>	1	<i>dst</i>	0	<i>src2</i>		0	1	0	0	1	1	<i>s</i>
2			1		1		3							1

NOTE: *dst* = A0, A1 or B0, B1 as selected by *dst* and *s*

Opcode map field used...	For operand type...
<i>src2</i>	sint

<i>op</i>	Mnemonic
0 0	<b>CMLT (.unit) ucst1, src2, dst</b> ( <i>dst</i> = <i>ucst1</i> < <i>src2</i> , signed compare)
0 1	<b>CMPGT (.unit) ucst1, src2, dst</b> ( <i>dst</i> = <i>ucst1</i> > <i>src2</i> , signed compare)
1 0	<b>CMPLTU (.unit) ucst1, src2, dst</b> ( <i>dst</i> = <i>ucst1</i> < <i>src2</i> , unsigned compare)
1 1	<b>CMPGTU (.unit) ucst1, src2, dst</b> ( <i>dst</i> = <i>ucst1</i> > <i>src2</i> , unsigned compare)

**Figure D-11 Lx1 Instruction Format**

15	13	12	11	10	9	7	6	5	4	3	2	1	0	
<i>op</i>		1	1	0	<i>src2/dst</i>		1	1	0	0	1	1	<i>s</i>	
3						3								1

<i>op</i>	Mnemonic
0 0 0	see LSDx1, <a href="#">Figure G-4</a>
0 0 1	see LSDx1, <a href="#">Figure G-4</a>
0 1 0	<b>SUB (.unit)0, src2, dst</b> ( <i>src2</i> = <i>dst</i> ; <i>dst</i> = 0 - <i>src2</i> )
0 1 1	<b>ADD (.unit)-1, src2, dst</b> ( <i>src2</i> = <i>dst</i> )
1 0 0	Reserved
1 0 1	see LSDx1, <a href="#">Figure G-4</a>
1 1 0	Reserved
1 1 1	see LSDx1, <a href="#">Figure G-4</a>



## Appendix E

# .M Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .M functional unit and illustrates the opcode maps for these instructions.

### E.1 Instructions Executing in the .M Functional Unit

Figure E-1 lists the instructions that execute in the .M functional unit.

**Table E-1 Instructions Executing in the .M Functional Unit**

Instruction	Instruction	Instruction	Instruction
AVG2	DOTPUS4	MPYIL	MPY32 (32-bit result)
AVGU4	DOTPU4	MPYILR	MPY32 (64-bit result)
BITC4	GMPY	MPYLH	MPY32SU
BITR	GMPY4	MPYLHU	MPY32U
CMPY	MPY	MPYLI	MPY32US
CMPYR	MPYDP	MPYLIR	MVD
CMPYR1	MPYH	MPYLSHU	ROTL
DDOTP4	MPYHI	MPYLUHS	SHFL
DDOTPH2	MPYHIR	MPYSP	SMPY
DDOTPH2R	MPYHL	MPYSPDP	SMPYH
DDOTPL2	MPYHLU	MPYSP2DP	SMPYHL
DDOTPL2R	MPYHSLU	MPYSU	SMPYLN
DEAL	MPYHSU	MPYSU4	Multiply Signed by Signed, 16 LSB × 16 LSB and 16 MSB × 16 MSB With Left Shift and Saturation
DOTP2	MPYHU	MPYU	SMPY32
DOTPN2	MPYHULS	MPYU4	SSHVL
DOTPNRSU2	MPYHUS	MPYUS	SSHVR
DOTPNRUS2	MPYI	MPYUS4	XORMPY
DOTPRS2	MPYID	MPY2	XPND2
DOTPRUS2	MPYIH	MPY2IR	XPND4
DOTPSU4	MPYIHR		

## E.2 Opcode Map Symbols and Meanings

Figure E-2 on page E-2 lists the symbols and meanings used in the opcode maps.

**Table E-2 .M Unit Opcode Map Symbol Definitions**

Symbol	Meaning
<i>creg</i>	3-bit field specifying a conditional register
<i>dst</i>	destination
<i>op</i>	opfield; field within opcode that specifies a unique instruction
<i>p</i>	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B
<i>src1</i>	source 1
<i>src2</i>	source 2
<i>x</i>	cross path for <i>src2</i> ; 0 = do not use cross path, 1 = use cross path
<i>z</i>	test for equality with zero or nonzero

## E.3 32-Bit Opcode Maps

The CPU 32-bit opcodes used in the .M unit are mapped in Figure E-1 through Figure E-3.

**Figure E-1 Extended M-Unit with Compound Operations**

31	29	28	27	23	22	18	17
<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>	
3	1		5			5	5
13	12	11	10	6	5	4	3
<i>src1</i>	<i>x</i>	0		<i>op</i>	1	1	0
5	1			5			1
							1
							0

**Figure E-2 Extended .M Unit 1 or 2 Sources, Nonconditional Instruction Format**

31	30	29	28	27	23	22	18	17
0	0	0	1	<i>dst</i>			<i>src2</i>	
				5			5	5
13	12	11	10	6	5	4	3	2
<i>src1</i>	<i>x</i>	0		<i>op</i>	1	1	0	0
5	1			5			1	1
								0

**Figure E-3 Extended .M-Unit Unary Instruction Format**

31	30	29	28	27	dst					src2					18	17
0	0	0	1		5					5					op	
13	12	11	10	9	8	7	6	5	4	3	2	1	0			
op	x	0	0	0	0	1	1	1	1	0	0	s	p			
															1	1

## E.4 16-Bit Opcode Maps

The CPU 16-bit opcodes used in the .M unit for compact instructions are mapped in [Figure E-4](#). See Section 3.10 “[Compact Instructions on the CPU](#)” on page 3-29 for more information about compact instructions.

**Figure E-4 M3 Instruction Format**

15	13	12	11	10	9		7	6	5	4	3	2	1	0
src1	x	dst		src2			op		1	1	1	1	s	
3	1	2		3			2							1

NOTE: RS = 0: dst from [A0, A2, A4, A6], [B0, B2, B4, B6]; RS = 1: dst from [A16, A18, A20, A22], [B16, B18, B20, B22]

Opcode map field used...	For operand type...
src1	sint
dst	sint
src2	xsint

SAT	op	Mnemonic
0	0	<b>MPY</b> (.unit) src1, src2, dst
0	0	<b>MPYH</b> (.unit) src1, src2, dst
0	1	<b>MPYLH</b> (.unit) src1, src2, dst
0	1	<b>MPYHL</b> (.unit) src1, src2, dst
1	0	<b>SMPY</b> (.unit) src1, src2, dst
1	0	<b>SMPYH</b> (.unit) src1, src2, dst
1	1	<b>SMPYLH</b> (.unit) src1, src2, dst
1	1	<b>SMPYHL</b> (.unit) src1, src2, dst



## Appendix F

# .S Unit Instructions and Opcode Maps

This appendix lists the instructions that execute in the .S functional unit and illustrates the opcode maps for these instructions.

### F.1 Instructions Executing in the .S Functional Unit

Table F-1 lists the instructions that execute in the .S functional unit.

**Table F-1 Instructions Executing in the .S Functional Unit**

Instruction	Instruction	Instruction	Instruction
ABSDP	CMPEQ2	MVKH/MVKLH	SET
ABSSP	CMPEQ4	MVKL	SHL
ADD	CMPEQDP	MVKH/MVKLH	SHLMB
ADDDP	CMPEQSP	NEG	SHR
ADDK	CMPGT2	NOT	SHR2
ADDKPC <sup>1</sup>	CMPGTD <sup>P</sup>	OR	SHRMB
ADDSP	CMPGTSP	PACK2	SHRU
ADD2	CMPGTU4	PACKH2	SHRU2
AND	CMPLT2	PACKHL2	SPACK2
ANDN	CMPLTD <sup>P</sup>	PACKLH2	SPACKU4
B displacement	CMPLTP	RCPDP	SPDP
B register <sup>1</sup>	CMPLTU4	RCPSP	SSH <sup>L</sup>
B IRP <sup>1</sup>	DMPYU4	RPACK2	SUB
B NRP <sup>1</sup>	EXT	RSQRDP	SUBDP
BDEC	EXTU	RSQRSP	SUBSP
BNOP displacement	MAX2	SADD	SUB2
BNOP register	MIN2	SADD2	SWAP2
BPOS	MV	SADDSU2	UNPKHU4
CALLP	MVC <sup>1</sup>	SADDUS2	UNPKLU4
CLR	MVK	SADDU4	XOR
			ZERO

1. S2 only

### F.2 Opcode Map Symbols and Meanings

Table F-2 lists the symbols and meanings used in the opcode maps.

## F.3 32-Bit Opcode Maps

## Appendix F—.S Unit Instructions and Opcode Maps

**Table F-2 .S Unit Opcode Map Symbol Definitions**

<b>Symbol</b>	<b>Meaning</b>
<i>creg</i>	3-bit field specifying a conditional register
<i>csta</i>	constant a
<i>cstb</i>	constant b
<i>cstn</i>	n-bit constant field
<i>dst</i>	destination
<i>h</i>	MVK or MVKH/MVKLH instruction
<i>N3</i>	3-bit field
<i>op</i>	opfield; field within opcode that specifies a unique instruction
<i>p</i>	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B
<i>scstn</i>	n-bit signed constant field
<i>src1</i>	source 1
<i>src2</i>	source 2
<i>ucstn</i>	n-bit unsigned constant field
<i>ucst<sub>n</sub></i>	bit n of the unsigned constant field
<i>x</i>	cross path for <i>src2</i> ; 0 = do not use cross path, 1 = use cross path
<i>z</i>	test for equality with zero or nonzero

**F.3 32-Bit Opcode Maps**

The CPU 32-bit opcodes used in the .S unit are mapped in [Figure F-1](#) through [Figure F-11](#).

**Figure F-1 1 or 2 Sources Instruction Format**

31	29	28	27	23	22	18
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>	
3	1		5		5	
17	13	12	11	6	5	4
<i>src1</i>	<i>x</i>		<i>op</i>	1	0	3
5	1		6		0	2
					1	1
					<i>s</i>	<i>p</i>

**Figure F-2 Extended .S Unit 1 or 2 Sources Instruction Format**

31	29	28	27	23	22	18
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>	
3	1		5		5	
17	13	12	11	10	9	6
<i>src1</i>	<i>x</i>	1	1	<i>op</i>	1	1
5	1			4	0	0
					<i>s</i>	<i>p</i>

**Figure F-3 Extended .S Unit 1 or 2 Sources, Nonconditional Instruction Format**

31	30	29	28	27		23	22		18
0	0	0	<i>z</i>		<i>dst</i>			<i>src2</i>	
			1		5			5	
17	13	12	11	10	9				18
<i>src1</i>	x	1	1		<i>op</i>	1	1	0	<i>s</i>
5	1				4			0	<i>p</i>
								1	1

**Figure F-4 Unary Instruction Format**

31	29	28	27		23	22		18	
	<i>creg</i>	<i>z</i>		<i>dst</i>			<i>src2</i>		
3		1		5			5		
17	13	12	11	10	9	8	7	6	18
<i>op</i>	x	1	1	1	1	0	0	1	<i>s</i>
5	1							0	<i>p</i>
								1	1

**Figure F-5 Extended .S Unit Branch Conditional, Immediate Instruction Format**

31	29	28	27		23	22		18	
	<i>creg</i>	<i>z</i>			<i>cst21</i>				
3		1			21				
			<i>cst21</i>		7	6	5	4	18
					0	0	1	0	<i>s</i>
					0	0	0	0	<i>p</i>
								1	1

**Figure F-6 Call Unconditional, Immediate with Implied NOP 5 Instruction Format**

31	30	29	28	27		23	22		18
0	0	0	<i>z</i>			<i>cst21</i>			
			1			21			
					<i>cst21</i>	7	6	5	18
						0	0	1	<i>s</i>
						0	0	0	<i>p</i>
								1	1

**Figure F-7 Branch with NOP Constant Instruction Format**

31	29	28	27		23	22		18	
	<i>creg</i>	<i>z</i>			<i>src2</i>				
3		1			12				
15	13	12	11	10	9	8	7	6	18
<i>src1</i>	0	0	0	0	1	0	0	1	<i>s</i>
3								0	<i>p</i>
								1	1

**F.4 16-Bit Opcode Maps****Appendix F—.S Unit Instructions and Opcode Maps****Figure F-8 Branch with NOP Register Instruction Format**

31	29	28	27	26	25	24	23	22	18	17	16
<i>creg</i>	<i>z</i>	0	0	0	0	1		<i>src2</i>	0	0	0
3	1						5				

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>src1</i>	<i>x</i>	0	0	1	1	0	1	1	0	0	0	0	<i>s</i>	<i>p</i>
3	1							1				1	1	

**Figure F-9 Branch Instruction Format**

31	29	28	27	26	25	24	23	22	18	17	16
<i>creg</i>	<i>z</i>	0	0	0	0	0	<i>src2</i>	0	0	0	0
3	1					5					
15	14	13	12	11	10	9	8	7	6	5	4
0	0	0	<i>x</i>	0	0	1	1	0	1	1	0
			1						0	0	<i>s</i>
									1	0	

**Figure F-10 MVK Instruction Format**

31	29	28	27	23	22	18
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>cst16</i>	
3	1		5		16	
17	13	12		7	6	5
<i>csta</i>		<i>cstb</i>		<i>op</i>	1	0
5		5		2		
					1	1

**Figure F-11 Field Operations**

31	29	28	27	23	22	18
<i>creg</i>	<i>z</i>		<i>dst</i>		<i>src2</i>	
3	1		5		5	
17	13	12		8	7	6
<i>csta</i>		<i>cstb</i>		<i>op</i>	0	1
5		5		2	0	<i>s</i>
					1	1

**F.4 16-Bit Opcode Maps**

The CPU 16-bit opcodes used in the .S unit for compact instructions are mapped in [Figure F-12](#) through [Figure F-27](#). See “Compact Instructions on the CPU” on page 29 for more information about compact instructions.

**Figure F-12 Sbs7 Instruction Format**

15	13	12		6	5	4	3	2	1	0
N3			scst7	0	0	1	0	1	s	
3			7							1

NOTE: N3 = 0, 1, 2, 3, 4, or 5

BR	Mnemonic
1	<b>BNOP</b> (.unit) scst7, N3

**Figure F-13 Sbu8 Instruction Format**

15	14	13		6	5	4	3	2	1	0
1	1		ucst8	0	0	1	0	1	s	
			8							1

BR	Mnemonic
1	<b>BNOP</b> (.unit) ucst8, 5

**Figure F-14 Scs10 Instruction Format**

15			6	5	4	3	2	1	0	
		scst10	0	1	1	0	1	s		
		10								1

NOTE: NextPC &gt; B3, A3

BR	Mnemonic
1	<b>CALLP</b> (.unit) scst10, 5

**Figure F-15 Sbs7c Instruction Format**

15	13	12		6	5	4	3	2	1	0
N3			scst7	1	z	1	0	1	s	
3			7							1

NOTE: N3 = 0, 1, 2, 3, 4, or 5

BR	s	z	Mnemonic
1	0	0	<b>[A0] BNOP</b> .S1 scst7, N3
1	0	1	<b>[!A0] BNOP</b> .S1 scst7, N3
1	1	0	<b>[B0] BNOP</b> .S2 scst7, N3
1	1	1	<b>[!B0] BNOP</b> .S2 scst7, N3

**Figure F-16 Sbu8c Instruction Format**

15	14	13		6	5	4	3	2	1	0
1	1		ucst8	1	z	1	0	1	s	
			8							1

BR	s	z	Mnemonic
1	0	0	<b>[A0] BNOP</b> .S1 ucst8, 5

<b>BR</b>	<b>s</b>	<b>z</b>	<b>Mnemonic</b>
1	0	1	<b>[!A0] BNOP</b> .S1 ucst8, 5
1	1	0	<b>[B0] BNOP</b> .S2 ucst8, 5
1	1	1	<b>[!B0] BNOP</b> .S2 ucst8, 5

**Figure F-17 S3 Instruction Format**

15	13	12	11	10	9	7	6	4	3	2	1	0
<i>src1</i>	x	<i>op</i>	0		<i>src2</i>		<i>dst</i>	1	0	1		<i>s</i>
3	1	1			3		3					1

<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src1</i>	sint
<i>src2</i>	xsint
<i>dst</i>	sint

<b>BR</b>	<b>SAT</b>	<b>op</b>	<b>Mnemonic</b>
0	0	0	<b>ADD</b> (.unit) <i>src1,src2,dst</i>
0	1	0	<b>SADD</b> (.unit) <i>src1,src2,dst</i>
0	x	1	<b>SUB</b> (.unit) <i>src1,src2,dst</i> ( <i>dst</i> = <i>src1 - src2</i> )

**Figure F-18 S3i Instruction Format**

15	13	12	11	10	9	7	6	4	3	2	1	0
<i>cst3</i>	x	<i>op</i>	1		<i>src2</i>		<i>dst</i>	1	0	1		<i>s</i>
3	1	1			3		3					1

<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src2</i>	xsint

<b>32-Bit Opcode cst Translation</b>		
<b>cst3</b>	<b>ucst5</b>	<b>Decimal Value</b>
000	10000	16
001	00001	1
010	00010	2
011	00011	3
100	00100	4
101	00101	5
110	00110	6
111	01000	8

<b>BR</b>	<b>op</b>	<b>Mnemonic</b>
0	0	<b>SHL</b> (.unit) <i>src2,ucst5,dst</i>
0	1	<b>SHR</b> (.unit) <i>src2,ucst5,dst</i>

**Figure F-19 Smvk8 Instruction Format**

15	13 12	11	10	9	7 6	5	4	3	2	1	0
$ucst_{2-0}$	$ucst_{4-3}$	$ucst_7$		$dst$	$ucst_{6-5}$	1	0	0	1	1	0

3                    2                    1                    3                    2                    1                    0

Opcode map field used...	For operand type...
$dst$	sint

Mnemonic
<b>MVK (.unit) <math>ucst8, dst</math></b>

**Figure F-20 Ssh5 Instruction Format**

15	13 12	11	10	9	7 6	5	4	3	2	1	0
$ucst_{2-0}$	$ucst_{4-3}$		1		$src2/dst$	$op$	0	0	0	1	0

3                    2                    1                    3                    2                    1                    0

NOTE: x = 0, src and dst on the same side.

Opcode map field used...	For operand type...
$src2/dst$	sint

SAT	op	Mnemonic
x	0 0	<b>SHL (.unit) <math>src2, ucst5, dst</math> (<math>src2 = dst</math>)</b>
x	0 1	<b>SHR (.unit) <math>src2, ucst5, dst</math> (<math>src2 = dst</math>)</b>
0	1 0	<b>SHRU (.unit) <math>src2, ucst5, dst</math> (<math>src2 = dst</math>)</b>
1	1 0	<b>SSHLL (.unit) <math>src2, ucst5, dst</math> (<math>src2 = dst</math>)</b>
x	1 1	see S2sh, <a href="#">Figure F-21</a>

**Figure F-21 S2sh Instruction Format**

15	13 12	11	10	9	7 6	5	4	3	2	1	0
$src1$	$op$	1		$src2/dst$	1	1	0	0	0	1	0

3                    2                    1                    3                    2                    1                    0

NOTE: x = 0, src and dst on the same side.

Opcode map field used...	For operand type...
$src2/dst$	sint

op	Mnemonic
0 0	<b>SHL (.unit) <math>src2, src1, dst</math> (<math>src2 = dst, dst = src2 &lt;&lt; src1</math>)</b>
0 1	<b>SHR (.unit) <math>src2, src1, dst</math> (<math>src2 = dst, dst = src2 &gt;&gt; src1</math>)</b>
1 0	<b>SHRU (.unit) <math>src2, src1, dst</math> (<math>src2 = dst, dst = src2 &lt;&lt; src1</math>)</b>
1 1	<b>SSHLL (.unit) <math>src2, src1, dst</math> (<math>src2 = dst, dst = src2 sshl src1</math>)</b>

**F.4 16-Bit Opcode Maps****Appendix F—.S Unit Instructions and Opcode Maps****Figure F-22 Sc5 Instruction Format**

15	13 12	11	10	9	7 6	5	4	3	2	1	0
$ucst_{2-0}$	$ucst_{4-3}$	0		$src2/dst$		$op$	0	0	0	1	s

3                    2                    3                    2                    1

NOTES:

x = 0, src and dst on the same side

s = 0, dst = A0; s = 1, dst = B0

Opcode map field used...	For operand type...
$src2/dst$	sint

<b>op</b>	<b>Mnemonic</b>
0 0	<b>EXTU</b> (.unit) $src2, ucst5, 31, A0/B0$
0 1	<b>SET</b> (.unit) $src2, ucst5, ucst5, dst$ ( $src = dst, ucst5 = ucst5$ )
1 0	<b>CLR</b> (.unit) $src2, ucst5, ucst5, dst$ ( $src = dst, ucst5 = ucst5$ )
1 1	see S2ext, Figure F-23

**Figure F-23 S2ext Instruction Format**

15	13 12	11	10	9	7	6	5	4	3	2	1	0
$dst$	$op$	0		$src2$	1	1	0	0	0	0	1	s

3                    2                    3                    1

NOTE: x = 0, src and dst on the same side.

Opcode map field used...	For operand type...
$dst$	sint
$src2$	sint

<b>op</b>	<b>Mnemonic</b>
0 0	<b>EXT</b> (.unit) $src, 16, 16, dst$
0 1	<b>EXT</b> (.unit) $src, 24, 24, dst$
1 0	<b>EXTU</b> (.unit) $src, 16, 16, dst$
1 1	<b>EXTU</b> (.unit) $src, 24, 24, dst$

**Figure F-24 Sx2op Instruction Format**

15	13 12	11	10	9	7	6	5	4	3	2	1	0
$src1/dst$	x	$op$	0		$src2$	0	1	0	1	1	1	s

3                    1                    1                    3                    1

Opcode map field used...	For operand type...
$src1/dst$	sint
$src2$	xsint

<b>op</b>	<b>Mnemonic</b>
0	<b>ADD</b> (.unit) $src1, src2, dst$ ( $src1 = dst$ )
1	<b>SUB</b> (.unit) $src1, src2, dst$ ( $src1 = dst, dst = src1 - src2$ )

## **Figure F-25 Sx5 Instruction Format**

<b>Opcode map field used...</b>	<b>For operand type...</b>
src2/dst	sint

**Mnemonic**  
**ADDK** (.unit) ucst5, dst

## **Figure F-26 Sx1 Instruction Format**

15	13	12	11	10	9	7	6	5	4	3	2	1	0
<i>op</i>	1	1	0		<i>src2/dst</i>	1	1	0	1	1	1	1	5
3					3								1

<b>Opcode map field used...</b>	<b>For operand type...</b>
src2/dst	sint

<b>op</b>	<b>Mnemonic</b>
0 0 0	see LSDx1, <a href="#">Figure G-4</a>
0 0 1	see LSDx1, <a href="#">Figure G-4</a>
0 1 0	<b>SUB</b> (.unit)0, <i>src2</i> , <i>dst</i> ( <i>src2</i> = <i>dst</i> , <i>dst</i> = 0 - <i>src2</i> )
0 1 1	<b>ADD</b> (.unit)-1, <i>src2</i> , <i>dst</i> ( <i>src2</i> = <i>dst</i> )
1 0 0	Reserved
1 0 1	see LSDx1, <a href="#">Figure G-4</a>
1 1 0	<b>MVC</b> (.unit) <i>src</i> , ILC (s = 1)
1 1 1	see LSDx1, <a href="#">Figure G-4</a>

## **Figure F-27 Sx1b Instruction Format**

NOTE: src2 from B0-B15

<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src2</i>	uint

Mnemonic
<b>BNOP</b> (.unit) <i>src2, N3</i>



# .D, .L, or .S Unit Opcode Maps

This appendix illustrates the opcode maps that execute in the .D, .L, or .S functional units.

For a list of the instructions that execute in the .D functional unit, see Appendix C “[.D Unit Instructions and Opcode Maps](#)” on page C-1. For a list of the instructions that execute in the .L functional unit, see Appendix D “[.L Unit Instructions and Opcode Maps](#)” on page D-1. For a list of the instructions that execute in the .S functional unit, see Appendix F “[.S Unit Instructions and Opcode Maps](#)” on page F-1.

- G.1 “[Opcode Map Symbols and Meanings](#)” on page G-1
- G.2 “[32-Bit Opcode Maps](#)” on page G-2
- G.3 “[16-Bit Opcode Maps](#)” on page G-2

### G.1 Opcode Map Symbols and Meanings

Table G-1 lists the symbols and meanings used in the opcode maps.

**Table G-1 .D, .L, and .S Units Opcode Map Symbol Definitions**

Symbol	Meaning
<i>CC</i>	
<i>dst</i>	destination
<i>dstms</i>	
<i>op</i>	opfield; field within opcode that specifies a unique instruction
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B
<i>src</i>	source
<i>src2</i>	source 2
<i>srcms</i>	
<i>ucstn</i>	n-bit unsigned constant field
<i>unit</i>	unit decode
<i>x</i>	cross path for <i>src2</i> ; 0 = do not use cross path, 1 = use cross path

## G.2 32-Bit Opcode Maps

### Appendix G—.D, .L, or .S Unit Opcode Maps

## G.2 32-Bit Opcode Maps

For the CPU 32-bit opcodes used in the .D functional unit, see Appendix C “[.D Unit Instructions and Opcode Maps](#)” on page C-1. For the CPU 32-bit opcodes used in the .L functional unit, see Appendix D “[.L Unit Instructions and Opcode Maps](#)” on page D-1. For the CPU 32-bit opcodes used in the .S functional unit, see Appendix F “[.S Unit Instructions and Opcode Maps](#)” on page F-1.

## G.3 16-Bit Opcode Maps

The CPU 16-bit opcodes used in the .D, .L, or .S units for compact instructions are mapped in [Figure G-1](#) through [Figure G-4](#). See Section 3.10 “[Compact Instructions on the CPU](#)” on page 3-29 for more information about compact instructions.

**Figure G-1 LSDmvto Instruction Format**

15	13	12	11	10	9	7	6	5	4	3	2	1	0
dst	x	srcms		src2		0	0	unit		1	1	s	
3	1	2		3					2			1	1

Opcode map field used...	For operand type...
dst	sint
src2	xsint

unit	Mnemonic
0 0	<b>MV (.Ln) src, dst</b>
0 1	<b>MV (.Sn) src, dst</b>
1 0	<b>MV (.Dn) src, dst</b>

**Figure G-2 LSDmvfr Instruction Format**

15	13	12	11	10	9	7	6	5	4	3	2	1	0
dst	x	dstms		src2		1	0	unit		1	1	s	
3	1	2		3					2			1	1

Opcode map field used...	For operand type...
dst	sint
src2	xsint

unit	Mnemonic
0 0	<b>MV (.Ln) src, dst</b>
0 1	<b>MV (.Sn) src, dst</b>
1 0	<b>MV (.Dn) src, dst</b>

**Figure G-3 LSDx1c Instruction Format**

15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
CC	ucst1	0	1	0		dst	1	1	unit		1	1	s	
2	1				3				2			1	1	1

Opcode map field used...	For operand type...
dst	sint

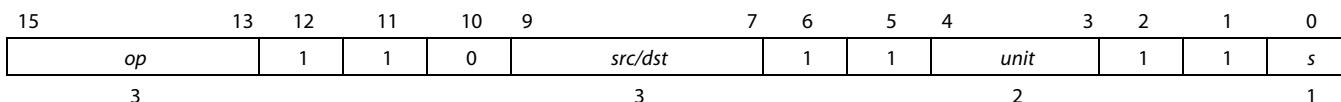
<b>CC</b>	<b>Mnemonic</b>	
0 0	[A0] MVK (.unit) ucst1, dst	
0 1	[!A0] MVK (.unit) ucst1, dst	
1 0	[B0] MVK (.unit) ucst1, dst	
1 1	[!B0] MVK (.unit) ucst1, dst	

<b>CC</b>	<b>unit</b>	<b>Mnemonic</b>
0 0	0 0	[A0] MVK (.Ln) ucst1, dst
	0 1	[A0] MVK (.Sn) ucst1, dst
	1 0	[A0] MVK (.Dn) ucst1, dst

<b>CC</b>	<b>unit</b>	<b>Mnemonic</b>
0 1	0 0	[!A0] MVK (.Ln) ucst1, dst
	0 1	[!A0] MVK (.Sn) ucst1, dst
	1 0	[!A0] MVK (.Dn) ucst1, dst

<b>CC</b>	<b>unit</b>	<b>Mnemonic</b>
1 0	0 0	[B0] MVK (.Ln) ucst1, dst
	0 1	[B0] MVK (.Sn) ucst1, dst
	1 0	[B0] MVK (.Dn) ucst1, dst

<b>CC</b>	<b>unit</b>	<b>Mnemonic</b>
1 1	0 0	[!B0] MVK (.Ln) ucst1, dst
	0 1	[!B0] MVK (.Sn) ucst1, dst
	1 0	[!B0] MVK (.Dn) ucst1, dst

**Figure G-4 LSDx1 Instruction Format**


<b>Opcode map field used...</b>	<b>For operand type...</b>
<i>src/dst</i>	sint

<b><i>op</i></b>	<b>Mnemonic</b>
0 0 0	<b>MVK (.unit)0, dst</b>
0 0 1	<b>MVK (.unit)1, dst</b>
0 1 0	See Dx1, <a href="#">Figure C-20</a> ; Lx1, <a href="#">Figure D-11</a> ; and Sx1, <a href="#">Figure F-26</a>
0 1 1	See Dx1, <a href="#">Figure C-20</a> ; Lx1, <a href="#">Figure D-11</a> ; and Sx1, <a href="#">Figure F-26</a>
1 0 0	See Dx1, <a href="#">Figure C-20</a> ; Lx1, <a href="#">Figure D-11</a> ; and Sx1, <a href="#">Figure F-26</a>
1 0 1	<b>ADD (.unit) src, 1, dst (src = dst)</b>
1 1 0	See Dx1, <a href="#">Figure C-20</a> ; Lx1, <a href="#">Figure D-11</a> ; and Sx1, <a href="#">Figure F-26</a>
1 1 1	<b>XOR (.unit) src, 1, dst (src = dst)</b>

<b><i>op</i></b>	<b><i>unit</i></b>	<b>Mnemonic</b>
0 0 0	0 0	<b>MVK</b> (.Ln)0, dst
	0 1	<b>MVK</b> (.Sn)0, dst
	1 0	<b>MVK</b> (.Dn)0, dst

<b><i>op</i></b>	<b><i>unit</i></b>	<b>Mnemonic</b>
0 0 1	0 0	<b>MVK</b> (.Ln)1, dst
	0 1	<b>MVK</b> (.Sn)1, dst
	1 0	<b>MVK</b> (.Dn)1, dst

<b><i>op</i></b>	<b><i>unit</i></b>	<b>Mnemonic</b>
1 0 1	0 0	<b>ADD</b> (.Ln) src, 1, dst
	0 1	<b>ADD</b> (.Sn) src, 1, dst
	1 0	<b>ADD</b> (.Dn) src, 1, dst

<b><i>op</i></b>	<b><i>unit</i></b>	<b>Mnemonic</b>
1 1 1	0 0	<b>XOR</b> (.Ln) src, 1, dst
	0 1	<b>XOR</b> (.Sn) src, 1, dst
	1 0	<b>XOR</b> (.Dn) src, 1, dst

# No Unit Specified Instructions and Opcode Maps

This appendix lists the instructions that execute with no unit specified and illustrates the opcode maps for these instructions.

For a list of the instructions that execute in the .D functional unit, see Appendix C “[.D Unit Instructions and Opcode Maps](#)” on page C-1. For a list of the instructions that execute in the .L functional unit, see Appendix D “[.L Unit Instructions and Opcode Maps](#)” on page D-1. For a list of the instructions that execute in the .M functional unit, see Appendix E “[.M Unit Instructions and Opcode Maps](#)” on page E-1. For a list of the instructions that execute in the .S functional unit, see Appendix F “[.S Unit Instructions and Opcode Maps](#)” on page F-1.

- H.1 ["Instructions Executing With No Unit Specified"](#) on page H-2
- H.2 ["Opcode Map Symbols and Meanings"](#) on page H-2
- H.3 ["32-Bit Opcode Maps"](#) on page H-3
- H.3 ["32-Bit Opcode Maps"](#) on page H-3

## H.1 Instructions Executing With No Unit Specified

Table H-1 on page H-2 lists the instructions that execute with no unit specified.

**Table H-1 Instructions Executing With No Unit Specified**

Instruction
DINT
IDLE
NOP
RINT
SPKERNEL
SPKERNELR
SPLOOP
SPLOOPD
SPLOOPW
SPMASK
SPMASKR
SWE
SWENR

## H.2 Opcode Map Symbols and Meanings

Table H-2 lists the symbols and meanings used in the opcode maps.

**Table H-2 No Unit Specified Instructions Opcode Map Symbol Definitions**

Symbol	Meaning
<i>creg</i>	3-bit field specifying a conditional register
<i>csta</i>	constant a
<i>cstb</i>	constant b
<i>cstn</i>	n-bit constant field
<i>ii<sub>n</sub></i>	bit n of the constant <i>ii</i>
<i>N3</i>	3-bit field
<i>op</i>	opfield; field within opcode that specifies a unique instruction
<i>p</i>	parallel execution; 0 = next instruction is not executed in parallel, 1 = next instruction is executed in parallel
<i>s</i>	side A or B for destination; 0 = side A, 1 = side B.
<i>stg<sub>n</sub></i>	bit n of the constant <i>stg</i>
<i>z</i>	test for equality with zero or nonzero

## H.3 32-Bit Opcode Maps

The CPU 32-bit opcodes used in the no unit instructions are mapped in [Figure H-1](#) and [Figure H-2](#).

**Figure H-1 Loop Buffer Instruction Format**

31	29	28	27		23	22		18	17					
<i>creg</i>	<i>z</i>		<i>cstb</i>				<i>csta</i>		1					
3	1		5				5							
16	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>op</i>	0	0	0	0	0	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>
4													1	1

**Figure H-2 NOP and IDLE Instruction Format**

31	30	29	28	27		18	17							
0	0	0	1		Reserved (0)		0							
10														
16	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>op</i>	0	0	0	0	0	0	0	0	0	0	0	0	<i>s</i>	<i>p</i>
4													1	1

## H.4 16-Bit Opcode Maps

The CPU 16-bit opcodes used in the no unit instructions for compact instructions are mapped in [Figure H-3](#) through [Figure H-7](#). See “[Compact Instructions on the CPU](#)” on page 3-29 for more information about compact instructions.

**Figure H-3 Uspl Instruction Format**

15	14	13	12	11	10	9		7	6	5	4	3	2	1	0
0	<i>ii</i> <sub>3</sub>	0	0	1	1		<i>ii</i> <sub>2-0</sub>	1	1	0	0	1	1	<i>op</i>	
1							3							1	

**NOTE: Supports ii of 1-16**

<b><i>op</i></b>	<b>Mnemonic</b>
0	<b>SPLLOOP ii</b> (ii = real ii - 1)
1	<b>SPLLOOPD ii</b>

**H.4 16-Bit Opcode Maps****Appendix H—No Unit Specified Instructions and Opcode Maps****Figure H-4 Uspldr Instruction Format**

15	14	13	12	11	10	9	7	6	5	4	3	2	1	0	
1	$ii_3$	0	0	1	1		$ii_{2-0}$		1	1	0	0	1	1	<i>op</i>
1							3							1	

**NOTE: Supports ii of 1-16**

<b><i>op</i></b>	<b>Mnemonic</b>
0	[A0] SPLOOPD <i>ii</i> ( <i>ii</i> = real <i>ii</i> - 1)
1	[B0] SPLOOPD <i>ii</i>

**Figure H-5 Uspk Instruction Format**

15	14	13	12	11	10	9	7	6	5	4	3	2	1	0
$ii/stg_{4-3}$	0	1	1	1		$ii/stg_{2-0}$		1	1	0	0	1	1	<i>ii/stg<sub>5</sub></i>
2						3								1

<b>Mnemonic</b>
SPKERNEL <i>ii/stage</i>

**Figure H-6 Uspm Instruction Format****a) SPMASK Instruction**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>D2</i>	<i>D1</i>	1	0	1	1	<i>S2</i>	<i>S1</i>	<i>L2</i>	1	1	0	0	1	1	<i>L1</i>
1	1				1		1	1							1

**NOTE: Supports masking of D1, D2, L1, L2, S1, and S2 instructions (not M1 or M2)**

<b>Mnemonic</b>
SPMASK <i>unitmask</i>

**b) SPMASKR Instruction**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>D2</i>	<i>D1</i>	1	1	1	1	<i>S2</i>	<i>S1</i>	<i>L2</i>	1	1	0	0	1	1	<i>L1</i>
1	1				1		1	1							1

**NOTE: Supports masking of D1, D2, L1, L2, S1, and S2 instructions (not M1 or M2)**

<b>Mnemonic</b>
SPMASKR <i>unitmask</i>

**Figure H-7 Unop Instruction Format**

15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<i>N3</i>	0	1	1	0	0	0	1	1	0	1	1	1	1	0
3														

<b>Mnemonic</b>
NOP <i>N3</i>

## **IMPORTANT NOTICE**

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<b>Products</b>		<b>Applications</b>	
Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>	Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>	Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>	Communications and Telecom	<a href="http://www.ti.com/communications">www.ti.com/communications</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>	Computers and Peripherals	<a href="http://www.ti.com/computers">www.ti.com/computers</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>	Consumer Electronics	<a href="http://www.ti.com/consumer-apps">www.ti.com/consumer-apps</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>	Energy	<a href="http://www.ti.com/energy">www.ti.com/energy</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>	Industrial	<a href="http://www.ti.com/industrial">www.ti.com/industrial</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>	Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>	Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>	Space, Avionics & Defense	<a href="http://www.ti.com/space-avionics-defense">www.ti.com/space-avionics-defense</a>
RF/IF and ZigBee® Solutions	<a href="http://www.ti.com/lprf">www.ti.com/lprf</a>	Video and Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
		Wireless	<a href="http://www.ti.com/wireless-apps">www.ti.com/wireless-apps</a>