

Black Iron E804 User Manual

April 24, 2024

Copyright © 2023 Hangzhou C-SKY MicroSystems Co., Ltd. All rights reserved.

This document is the property of Hangzhou C-SKY MicroSystems Co., Ltd. and its affiliates ("C-SKY"). This document may only be distributed to: (i) a C-SKY party having a legitimate business need for the information contained herein, or (ii) a non-C-SKY party having a legitimate business need for the information contained herein.

No license, expressed or implied, under any patent, copyright or trade secret right is granted or implied by the conveyance of this document.

Trademarks and Permissions

The C-SKY Logo and all other trademarks indicated as such herein (including XuanTie) are trademarks of Hangzhou C-SKY MicroSystems Co., Ltd. All other products or service names are the property of their respective owners.

Notice

The purchased products, services and features are stipulated by the contract made between C-SKY and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Hangzhou C-SKY MicroSystems Co., LTD

Address: Room 201, 2/F, Building 5, No.699 Wangshang Road, Hangzhou, Zhejiang, China

Website: www.xrvm.cn

Copyright © 2023 Hangzhou C-SKY MicroSystem Co., Ltd. All rights reserved.

The ownership and intellectual property rights of this document belong to Hangzhou C-SKY Microsystems Co., Ltd. and its affiliates (hereinafter referred to as "C-SKY"). This document can only be distributed to: (i) C-SKY employees who have a legal employment relationship and need the information in this document, or (ii) partners who are not C-SKY organizations but have a legal cooperation relationship and need the information in this document. This document cannot be used without the express consent of Hangzhou C-SKY Microsystems Co., Ltd. No part of this document may be copied, disseminated, transcribed, stored in a retrieval system or translated into any language or computer language without the written permission of C-SKY.

Trademark Statement

Zhongtian Micro's LOGO and all other trademarks (such as XuanTie) are owned by Hangzhou Zhongtian Microsystems Co., Ltd. and its affiliated companies. No legal entity may use Zhongtian Micro's trademarks or commercial logos without the written consent of Hangzhou Zhongtian Microsystems Co., Ltd.

Notice

The products, services or features you purchase are subject to the commercial contracts and terms of Zhongtian Micro. All or part of the products, services or features described in this document may not be within the scope of your purchase or use. Unless otherwise agreed in the contract, Zhongtian Micro does not make any express or implied statements or warranties regarding the contents of this document.

Due to product version upgrades or other reasons, the content of this document will be updated from time to time. Unless otherwise agreed, this document is only used as a guide, and all statements, information and suggestions in this document do not constitute any express or implied warranty. Hangzhou Zhongtian Microsystem Co., Ltd. does not assume any legal responsibility for any losses caused by the use of this document by any third party.

Hangzhou C-SKY MicroSystems Co., LTD

Address: Room 201, 2F, Building 5, No. 699 Wangshang Road, Hangzhou, Zhejiang, China

Website: www.xrvm.cn

Revision History

Version	Description	date
01	First official release.	2021.07.31
02	Updated templates.	2024.04.24

Black Iron E804 User Manual

Chapter 1	1
Overview 1.1 Introduction	1
Configurable Options	2
Debuggability	3
Symbols	3
1.2 Features	1
1.4 Design for Testability	1
1.6 Naming Conventions	1
1.6.1	1
1.6.2 Terminology	1

Chapter 2	6
Microarchitecture	
2.1 Block Diagram	6
2.3 Floating Point Processing Unit	8
2.5 Trusted Protection Technology	9
2.2 Pipeline Introduction	2
2.4 DSP Extension Unit	2
2.6 Tightly Coupled IP Architecture	2

Chapter 3 Programming Model	11
3.1 Operating Mode and Register View	11
3.2 General Registers	12
3.2.1 Condition Code/Carry Flag	13
3.3 System Control Registers	13
3.3.1 Processor Status Register (PSR, CR<0,0>)	14
3.3.2 Update	14
3.3.3 Vector Base Register (VBR, CR<1,0>)	16
3.3.4 Exception Reserved Registers (CR<2,0>, CR<4,0>)	17
3.3.5 Product Identification Register (CPUIDR, CR<13,0>)	17
3.3.6 Implicit Operation Register (CHR, CR<31,0>)	17
3.3.7 Other Control Registers	18
3.3.8 General User Mode Register (R14(User SP), CR<14,1>)	18
3.3.9 Interrupt Pointer Register (R14(Int_SP), CR<15,1>)	18
3.4 Data Endianness	19

3.5 Data Alignment Access	19	3.6 System Address Mapping	19	3.7
Memory Access Order	20			

Chapter 4 Exception Handling

twenty two

4.1 Exception Handling Overview	22	4.2 Exception Types																					
24 4.2.1 Restart Exception (Vector Offset 0X0)	24	4.2.2 Unaligned Access Exception (Vector Offset 0X4)	24	4.2.3 Access Error Exception (Vector Offset 0X8)	25	4.2.4 Divide by Zero Exception (Vector Offset 0X0C)	25	4.2.5 Illegal Instruction Exception (Vector Offset 0X10)	25	4.2.6 Privilege Violation Exception (Vector Offset 0X14)	25	4.2.7 Trace Exception (Vector Offset 0X18)	25	4.2.8 Breakpoint Exception (Vector Offset 0X1C)	26	4.2.9 Unrecoverable Error Exception (Vector Offset 0X20)	26	4.2.10 4.2.11 Trap Instruction Exception (Vector Offset 0X40 - 0X4C)	26	4.2.12 TSPEND Interrupt (Vector Offset 0X58)	26	4.2.13 Floating Point Exception (Vector Offset 0X78)	
26 4.3 Interrupt Exceptions	26	4.4 Exception Priority	28	4.5 Exception Return	29																		

Chapter 5 Tightly Coupling IP

30

5.1 Tightly Coupled IP Introduction	30	5.2 System Timer											
31 5.2.1 Introduction	31	5.2.2 Register Definitions											
31 5.2.2.1 Control and Status Register (CORET_CSR)	32	5.2.2.2 Backfill Value Register (CORET_RVR)	33	5.2.2.3 Current Value Register (CORET_CVR)	34	5.2.2.4 Calibration Register (CORET_CALIB)	34	5.2.3 Operation Procedure	35	5.3 Vectored Interrupt Controller	36		
5.3.1 Introduction	36	5.3.2 Register Definitions	37	5.3.2.1 Interrupt Enable Set Register (VIC_ISER)	38	5.3.2.2 Interrupt Low Power Wake-up Setting Register (VIC_IWER)	39	5.3.2.3 Interrupt Enable Clear Register (VIC_ICER)	40	5.3.2.4 Interrupt Low Power Wake-up Clear Register (VIC_IWDR)	41	5.3.2.5 Interrupt Wait Setting Register (VIC_ISPR)	41

5.3.2.6 Interrupt Pending Clear Register (VIC_ICPR)	42	5.3.2.7 Interrupt Response Status Register (VIC_IABR)	42
5.3.2.8 Interrupt Priority Setting Register (VIC_IPR0 - VIC_IPR31)	43	5.3.2.9 Interrupt Status Register (VIC_ISR)	45
5.3.2.10 Interrupt Priority Threshold Register (VIC_IPTR)	45	5.3.2.11 Tspend Interrupt Enable Setting Register (VIC_TSPEND)	46
5.3.2.12 Tspend Interrupt Response Status Register (VIC_TSABR)	47	5.3.2.13 Tspend Interrupt Priority Setting Register (VIC_TSPPR)	47
5.3.3 Interrupt Handling Mechanism	48	5.3.3.1 Interrupt Status Bit	48
5.3.3.2 Interrupt Priority	49	5.3.3.3 Interrupt Vector Number	49
5.3.3.4 Interrupt Handling Procedure	50	5.3.3.5 Interrupt Nesting	54
5.3.3.6 Tspend Interrupt Operation Steps	54	5.3.6 Interface Signal	54
5.3.7 Interrupt Setting Example	57		

Chapter 6 59

Instruction Set	6.1 Overview	59	6.2 32-bit Instructions	
6.2.1 32-bit Instruction Function Classification	59	6.2.1.1 Data Operation Instructions		
6.2.1.2 Branch Jump Instructions	63	6.2.1.3		
6.2.1.4 Privileged Memory Access Instructions	64	6.2.1.5 Special Function Instructions		
6.2.1.6 DSP Extension Instructions	66	6.3 16-bit Instructions		
6.3.1 16-bit Instruction Function Classification	72	6.3.1.1 Data Operation Instructions	72	
6.3.1.2 Branch Jump Instructions	74	6.3.1.3 Memory Access Instructions	74	
6.3.1.4 Multiple Register Access Instructions	75	6.4 Instruction Set List	90	

Chapter 7 Memory 100

Protection	7.1 Introduction to Memory Protection Unit	100	7.2 Related System Control Registers	
	7.2.1 Cache Configuration Register (CCR, CR<18,0>)	100		

7.2.2 Bufferable Access Permission Configuration Register (CAPR, CR<19,0>)	101	7.2.3	
Protection Zone Control Register (PACR, CR<20,0>)	102	7.2.4 Protection Zone Select	
Register (PRSR, CR<21,0>)	103	7.3 Memory Access Processing	
	104		
Chapter 8 On-Chip Cache		105	
8.1 Cache Introduction	105	8.2 Related System Control	
Registers	105	8.2.1 Cache Enable Register (CER)	
Cache Invalidate Register (CIR)	107	8.2.3 Cacheable Configuration Registers	
0~3 (CRCR)	107	8.2.4 Cache Performance Analysis Control Register (CPFCR)	
109	8.2.5 Cache Access Count Register (CPFATR)	109	8.2.6 Cache Miss Count Register (CPFMTR)
	109		
Chapter 9 Bus Matrix and Bus Interface		111	
9.1 Introduction	111	9.2 System Bus Interface	
9.2.1 Features	113	9.2.2 Protocol Contents	
Supported Transfer Types	113	9.2.2.1	
Behaviors under Different Bus Responses	114	9.2.2.2 Supported Response Types	
9.2.3		9.2.4 Interface Signals of AHB	
Protocol	114	9.2.5 Interface Signals of AHB-Lite Protocol	
Instruction Bus Interface	117	9.3.1 Features	
117	9.3.2 Protocol Contents	117	9.3.2.1 Supported Transfer
Types	117	9.3.2.2	
117	9.3.3 Behavior under different bus responses ...	118	9.3.4 Command bus
interface signals	119	9.4 Data bus interface	
120	9.4.1		
Features	120	9.4.2 Protocol content	
Supported transfer types	120	9.4.2.1	
120	9.4.2.2 Supported response types		
121	9.4.3 Behavior under different bus responses	121	9.4.4 Data bus interface
signals	122	9.5 Access order of instructions and data	
	123		
Chapter 10 Debug		124	
10.1 Overview	124		

10.2 External Interface	125
Chapter 11 Working Mode and Conversion	127
11.1 Normal Operation Mode	127
11.2 Low Power Mode	128
11.3 Debug Mode	128
11.3.1 Debug Mode	128
11.3.2 Exit Debug Mode	129
Enter Debug Mode	128
Chapter 12 Floating Point Processing Unit	130
12.1 Overview	130
12.1.1 Introduction	130
12.1.2 Features	130
12.2 Microarchitecture	
131 12.2.1 Introduction to the Floating Point Unit	131
12.3 Programming Model	
131 12.3.1 Introduction	131
12.3.2 Data Formats	131
12.3.2.1 Integer Data Formats	131
12.3.2.2 Single Precision Data Formats	132
12.3.3 Floating Point Register Descriptions	132
12.3.3.1 Transferring Data to and from General Purpose Registers	132
12.3.3.2 Maintaining Register Precision Consistency	132
12.3.4 Reading and Writing Floating Point System Registers	134
12.3.4.1 Floating Point Version Register (FPU ID Register, FID)	134
12.3.4.2 Floating Point Control Register (FPU Control Register, FCR)	135
12.3.4.3 Floating Point Exception Status Handler (FPU Exception Status Register, FESR)	137
12.3.4.4 Data Endianness	138
12.4 Exception Handling	138
12.4.1 Introduction to Floating Point Exceptions	139
12.4.2 Non-Normal Number Input Exception	139
12.4.2.1 Exception Enable	139
12.4.2.2 Exception Disable	140
12.4.3 Illegal Operation Exception	140
12.4.3.1 Exception Enabled	140
12.4.3.2 Exception Disabled	141
12.4.4 Divide by Zero Exception	141
12.4.4.1 Exception Enabled	141
12.4.4.2 Exception Disabled	141
12.4.5 Overflow Exception	141

12.4.5.1 Exception Enabled	141	12.4.5.2 Exception Disabled	142
12.4.6 Underflow Exception	142	12.4.6.1 Exception Enabled.....	142
12.4.6.2 Exception Disabled	142		
12.4.7 Inexact Exception	142	12.4.7.1 Exception Enabled	
12.4.7.2 Exception Disabled.....	143	12.4.8 Exception Priority	143
12.4.9 Exceptions for Special Instructions	143	12.5 Instruction Set	143
12.5.1 Data Operation Instructions	144	12.5.1.1 Floating Point Instruction List	
12.5.1.2 Memory Access Instructions	145	12.5.1.3	
12.5.2 Floating Point Instruction Execution Delay	145		
Chapter 13 Initialization Reference Code	147		
13.1 MPU Setting Example	147	13.2 Cache Setting Example	149
13.3 Interrupt Enable Initialization	150	13.4 General Register Initialization Example	
13.5 Stack Pointer Initialization Example	151	13.6 Exception and Interrupt Service Routine Entry Address Setting Example	151
13.7 FPU Initialization Example	152		
Chapter 14 Appendix A Basic Command Glossary	154		
14.1 ABS – Absolute Value Instruction	154	14.2 ADDC – Unsigned Add with Carry Instruction	
14.3 ADDI – Unsigned Add Immediate Instruction	156	14.4 ADDI(SP) – Unsigned Add Immediate (Stack Pointer) Instruction	160
14.5 ADDU – Unsigned Add Instruction	162	14.6 AND – Bitwise AND Instruction	164
14.7 ANDI – Bitwise AND Immediate Instruction	165	14.8 ANDN – Bitwise AND Not Instruction	167
14.9 ANDNI – Bitwise AND Not Immediate Instruction	168	14.11 ASRC – Arithmetic Shift Right Immediate to C Instruction	170
14.12 ASRI - Arithmetic Shift Right Immediate Instruction	172	14.13 BCLRI - Bit Clear Immediate Instruction	173
14.14 BEZ - Branch If Register Equals Zero Instruction	175	14.15 BF - Branch If C Is Zero Instruction	176

14.16 BGENI – Generate Bit Immediately Instruction	178	14.17 BGEND – Generate Bit Register Instruction	179
14.18 BHSZ – Branch If Register Greater Than Zero Instruction	180		
14.19 BHZ – Branch If Register Greater Than Zero Instruction	181	14.20 BKPT – Breakpoint Instruction	182
14.21 BLSZ – Branch If Register Less Than Zero Instruction	183		
14.22 BLZ – Branch If Register Less Than Zero Instruction	185	14.23 BMASKI – Generate Bit Immediate Mask Instruction	186
14.24 BNEZ – Branch If Register Not Equal To Zero Instruction	188	14.25 BNEZAD – Branch If Register Decrement Greater Than Zero Instruction	189
14.26 BR – Unconditional Jump Instruction	191	14.27 BREV – Bit Reverse Instruction	192
14.28 BSETI – Set Bit Immediately Instruction	193		
14.29 BSR – Jump to Subroutine Instruction	195	14.30 BT – Branch If C is 1 Instruction	198
14.31 BTSTI – Test Bit Immediately Instruction	199	14.32 CLRF – Clear If C is 0 Instruction	200
14.33 CLRT – Clear If C is 1 Instruction	201	14.34 CMPHS – Compare Unsigned Greater Than or Equal Instruction ..	202
14.35 CMPHSI – Compare Unsigned Greater Than or Equal Immediately Instruction ..	204	14.36 CMPLT – Compare Signed Less Than Instruction	207
14.37 CMPLTI – Compare Signed Less Than Immediate Instruction	209		
14.38 CMPNE – Compare Not Equal Instruction	212	14.39 CMPNEI – Compare Not Equal Immediate Instruction	214
14.40 DECF – Subtract Immediate If C is Zero Instruction	217	14.42 DECLT – Subtract Less Than Zero Set C Instruction	218
14.41 DEC GT – Subtract Greater Than Zero Set C Instruction	218	14.43 DECNE – Subtract Not Equal To Zero Set C Instruction	219
14.44 DECT – Subtract Immediate If C is One Instruction	220	14.45 DIVS – Signed Divide Instruction	221
14.46 DIVU – Unsigned Divide Instruction	222	14.47 DOZE – Enter Low Power Sleep Mode Instruction	223
14.48 FF0 – Fast Find Zero Instruction	224	14.49 FF1 – Fast Find One Instruction	225
14.50 GRS – Generate Sign Instruction	226		
14.51 IDLY – Interrupt Identification Disable Instruction	227	14.52 INCF – Add Immediate Value If C Is 0 Instruction	229
14.53 INCT – Add Immediate Value If C Is 1 Instruction	230		
14.54 INS – Insert Bit Instruction	231	14.55 IPOP – Interrupt Pop Instruction	233
14.56 IPUSH – Interrupt Push Instruction	234		
14.57 IXH – Index Halfword Instruction	235		

14.58 IXW – Index Word Instruction	236	14.59 IxD – Index Double Word Instruction	236
14.60 JMP – Jump From Register Instruction	237	14.61 JMPI – Jump From Indirect Instruction	239
14.62 JSR – Jump From Register To Subroutine Instruction	240	14.63 JSRI – Jump From Indirect To Subroutine Instruction	242
14.64 LD.B – Load Unsigned Extended Byte Instruction	240	14.65 LD.BS – Load Sign-Extended Byte Instruction	244
14.66 LD.H – Load Unsigned-Extended Halfword Instruction	248	14.67 LD.HS – Load Sign- Extended Halfword Instruction	251
14.68 LD.W – Load Word Instruction	251	14.69 LDM – Continuous Load Multiple Words Instruction ..	255
14.70 LDQ – Continuous Load Quad Words Instruction ..	257	14.71 LDR.B – Register Shift Addressed Unsigned Extended Byte Load Instruction .	259
14.72 LDR.BS – Register Shift Addressed Signed Extended Byte Load Instruction .	260	14.73 LDR.H – Register Shift Addressed Unsigned Extended Half Word Load Instruction .	262
14.74 LDR.HS – Register Shift Addressed Signed Extended Half Word Load Instruction .	264	14.75 LDR.W – Register Shift Addressed Word Load Instruction ..	266
14.76 LRS.B – Byte Signed Load Instruction ..	268	14.77 LRS.H – Half Word Signed Load Instruction ..	270
14.78 LRW – Memory Read Instruction	273	14.80 LSL – Logical Shift Left Instruction	275
14.81 LSLC – Logical Shift Left Immediate to C Instruction	276	14.82 LSLI – Logical Shift Left Immediate to C Instruction	278
14.83 LSR – Logical Shift Right Instruction	280	14.84 LSRC – Logical Shift Right Immediate to C Instruction	281
14.85 LSRI – Logical Shift Right Immediate to C Instruction	283		
14.86 MFCR – Control Register Read Move Instruction	284	14.87 MOV – Data Move Instruction	285
14.88 MOVF – Data Move with C = 0 Instruction	285		
14.89 MOVI – Move Immediate Data Instruction	287	14.90 MOVIH – Move Immediate Data High Instruction ..	288
14.91 MOVT – Move Data with C = 1 Instruction	289	14.92 MTCR – Write Control Register Transfer Instruction	290
14.93 MULA.32.I – 32-Bit Signed Multiply Accumulate Low 32 Instruction ..	290	14.94 MULALL.S16.S – 16-Bit Signed Multiply Accumulate Low Halfword Instruction with Saturation Operation ..	291
14.95 MULA.(U/S)32 – 32-Bit (Unsigned/Yes) Signed Multiply Accumulate Instruction ..	291	14.96 MULT – Multiplication Instruction	293
14.97 MUL.(U/S)32 – 32-Bit (Unsigned/Yes) Signed Multiply Accumulate Instruction ..	294	14.98 MVC - C Bit Move Instruction	295
14.99 MVCV - C Bit Negative Move	297		

14.100 NIE – Nested Interrupt Enable Instruction ..	298	14.101 NIR – Nested Interrupt Return Instruction ..
300 14.102 NOR – Bitwise Not OR Instruction	302	14.103 NOT – Bitwise Not Instruction
303 14.104 OR – Bitwise Or Instruction	304	14.105 ORI – Bitwise Or Immediate
Instruction	305	14.106 POP – Pop – Stack Instruction
– Clear – PSR Instruction	311	14.108 PSRSET – Set – PSR Instruction
PUSH – Push – Stack Instruction	315	14.110 REVB – Reverse – Byte
Instruction	319	14.111 REVH – Reverse Halfword Instruction
14.112 ROTL – Rotate Left Instruction	322	14.113 ROTLI – Rotate Left Immediate Instruction
323 14.114 RSUB – Reverse Subtraction Instruction	324	14.115 RTS – Return from Subroutine
Instruction	325	14.116 RTE – Return from Exception and Normal Interrupt Instruction ..
14.117 SCE – Set Conditional Execution Instruction	327	14.118 SEXT – Extract Bit and Sign-
Extend Instruction	330	14.119 SEXTB – Extract Byte and Sign-Extend Instruction
332 14.120 SEXTH – Extract Halfword and Sign-Extend Instruction	333	14.121 SRS.B – Store
Byte Signed Instruction	335	14.122 SRS.H – Store Half-Word Signed Instruction
336 14.123 SRS.W – Store Word Signed Instruction	338	14.124 ST.B – Store Byte Signed
Instruction	339	14.125 ST.H – Store Half-Word Signed Instruction
342 14.126 ST.W – Store Word Signed Instruction	344	14.127 STM – Store Multiple Continuous Words
Instruction	347	14.128 STOP – Enter Low Power Halt Mode Instruction
349 14.129 STQ – Store Quad Continuous Words Instruction	350	14.130 STR.B – Shift Register Addressed
Byte Store Instruction	352	14.131 STR.H – Shift Register Store Halfword Instruction ..
353 14.132 STR.W – Shift Register Store Word Instruction ..	355	14.133 SUBC – Unsigned Subtract with Borrow
Instruction ..	357	14.134 SUBI – Unsigned Subtract Immediate Instruction ..
359 14.135 SUBI(SP) – Unsigned Subtract (Stack Pointer) Immediate Instruction ..	362	14.136 SUBU – Unsigned Subtract
Instruction ..	363	14.137 SYNC – CPU Synchronize Instruction ..
365 14.138 TRAP – Operating System Trap Instruction ..	366	14.139 TST – Test for Zero Instruction ..
367 14.140 TSTNBZ – No Byte Equal to Zero Register Test Instruction ..	369	14.141 WAIT——Enter Low Power Wait Mode
Instruction	371	

14.142 XOR – Bitwise Exclusive OR Instruction	372	14.143 XORI – Bitwise Exclusive OR Immediate Instruction ..	373
14.144 XSR – Extended Shift Right Instruction ..	374	14.145 XTRB0 – Extract Byte 0 Without Sign-Extension Instruction ..	376
14.146 XTRB1 – Extract Byte 1 Without Sign-Extension Instruction ..	377	14.147 XTRB2 – Extract Byte 2 Without Sign-Extension Instruction ..	378
14.148 XTRB3 – Extract Byte 3 Without Sign-Extension Instruction ..	379	14.149 ZEXT – Bit Extract Without Sign-Extension Instruction ..	380
14.150 ZEXTB – Byte Extract Without Sign-Extension Instruction ..	382	14.151 ZEXTH – Extract Halfword Without Sign-Extend Instruction	383

Chapter 15 Appendix B DSP Instruction 385

Set Glossary	
15.1 PADD.8 —— 8-bit Parallel Add Instruction	386
15.2 PADD.16 —— 16-bit Parallel Add Instruction	387
15.3 PADD.(U/S)8.S —— 8-bit Parallel (Unsigned/Signed) Add Instruction with Saturation Operation ..	388
15.4 PADD.(U/S)16.S —— 16-bit Parallel (Unsigned/Signed) Add Instruction with Saturation Operation ..	390
15.5 ADD.(U/S)32.S —— 32-bit (Unsigned/Signed) Add Instruction with Saturation Operation ..	392
15.6 PSUB.8 —— 8-bit Parallel Subtraction Instruction	394
15.7 PSUB.16 —— 16-bit Parallel Subtraction Instruction ..	395
15.8 15.14 PADDH.(U/S)8.S – 8-Bit Parallel (Un/Yes) Signed Subtract Instruction with Saturation ..	396
15.9 PSUB.(U/S)16.S – 16-Bit Parallel (Un/Yes) Signed Subtract Instruction with Saturation ..	398
15.10 SUB.(U/S)32.S – 32-Bit (Un/Yes) Signed Subtract Instruction with Saturation ..	400
15.11 PADDH.(U/S)8 – 8-Bit Parallel (Un/Yes) Signed Add Average Instruction ..	402
15.12 PADDH.(U/S)16 – 16-Bit Parallel (Un/Yes) Signed Add Average Instruction ..	403
15.13 ADDH.(U/S)32 – 32-Bit (Un/Yes) Signed Add Average Instruction ..	404
15.14 15.15 PSUBH.(U/S)16 – 16-Bit Parallel (Unsigned/Signed) Subtract Averaging Instruction ..	406
15.16 SUBH.(U/S)32 – 32-Bit (Unsigned/Signed) Subtract Averaging Instruction ..	407
15.17 PASX.16 – 16-Bit Parallel Interleaved Add/Subtract Instruction ..	407
15.18 PSAX.16 – 16-Bit Parallel Interleaved Subtract/Add Instruction ..	408
15.19 PASX.(U/S)16.S – 16-Bit Parallel (Unsigned/Signed) Interleaved Add/Subtract with Saturation Instruction ..	409
15.20 PSAX.(U/S)16.S – 16-Bit Parallel (Unsigned/Signed) Interleaved Subtract/Add with Saturation Instruction ..	411
15.21 PASXH.(U/S)16 – 16-Bit Parallel (Un/Signed) Signed Interleaved Add/Subtract Average Instruction ..	412
15.22 PSAXH.(U/S)16 – 16-Bit Parallel (Un/Signed) Signed Interleaved Add/Subtract Average Instruction ..	413
15.23 ADD.64 – 64-Bit Add Instruction	414
15.24 ADD.(U/S)64.S – 64-Bit (Un/Signed) Add with Saturation Instruction ..	416
15.25 SUB.64 – 64-Bit Subtract Instruction	417
15.26 SUB.(U/S)64.S – 64-Bit (Un/Signed) Subtract with Saturation Instruction ..	418
15.27 ASRI.S32.R – Immediate Arithmetic Shift Right with Rounding Instruction ..	419
15.28 ASR.S32.R – Arithmetic Shift Right with Rounding Instruction ..	420
15.29 LSRI.U32.R – Immediate Logical Shift Right with Rounding Instruction ..	421
15.30 LSR.U32.R – Logical Shift Right with Rounding Instruction ..	422

15.31 LSLI.(U/S)32.S – Signed Logical Shift Left Immediate (Without/With) Saturation Instruction ..	423
15.32 LSL.(U/S)32.S – Signed Logical Shift Left Immediate (Without/With) Saturation Instruction ..	425
15.33 PASRI.S16 – 16-Bit Parallel Immediate Arithmetic Shift Right Instruction ..	426
15.34 PASR.S16 – 16 -Bit Parallel Arithmetic Shift Right Instruction ..	427
15.35 PASRI.S16.R – 16-Bit Parallel Immediate Arithmetic Shift Right Instruction with Rounding ..	428
15.36 PASR.S16.R – 16-Bit Parallel Arithmetic Shift Right Instruction with Rounding ..	430
15.37 PLSRI.U16 – 16-Bit Parallel Immediate Logical Shift Right Instruction ..	431
15.38 PLSR.U16 – 16-Bit Parallel Logical Shift Right Instruction ..	432
15.39 PLSRI.U16.R – 16-Bit Parallel Immediate Logical Shift Left Instruction ..	432
15.40 PLSL.16 – 16-Bit Parallel Logical Shift Left Instruction ..	437
15.41 PLSLI.(U/S)16.S – 16-Bit Parallel Immediate (No/Yes) Signed Logical Shift Left Instruction with Saturation ..	438
15.42 PLSL.(U/S)16.S – 16-Bit Parallel (No/Yes) Signed Logical Shift Left Instruction with Saturation ..	440
15.43 PLSLI.(U/S)16 – 16-Bit Parallel Immediate (No/Yes) Signed Logical Shift Left Instruction with Saturation ..	440
15.44 SEL – Bit Select Instruction	441
15.45 PCMPNE.8 – 8-Bit Parallel Compare Not Equal Instruction ..	442
15.46 PCMPNE.16 – 16-Bit Parallel Compare Not Equal Instruction ..	443
15.47 PCMPHS.(U/S)8 – 8-Bit Parallel (Un/Signed) Signed Greater Than Equal Comparison Instruction ..	445
15.48 PCMPHS.(U/S)16 – 16-Bit Parallel (Un/Signed) Signed Greater Than Equal Comparison Instruction ..	446
15.49 PCMPLT.(U/S)8 – 8-Bit Parallel (Un/Signed) Signed Less Than Comparison Select Instruction ..	447
15.50 PCMPLT.(U/S)16 – 16-Bit Parallel (Un/Signed) Signed Less Than Comparison Instruction ..	449
15.51 PCMAX.(U/S)8 – 8 15.53 PCMAX.(U/S)16 — 16-Bit Parallel (Unsigned/With) Signed Large Instruction .	451
15.54 MAX.(U/S)32 — 32-Bit (Unsigned/With) Signed Large Instruction .	452
15.55 PMIN.(U/S)8 — 8-Bit Parallel (Unsigned/With) Signed Small Instruction .	453
15.56 PMIN.(U/S)16 — 16-Bit Parallel (Unsigned/With) Signed Small Instruction .	454
15.57 MIN .(U/S)32 — 32-Bit (Unsigned/With) Signed Small Instruction .	455
15.58 DEXTI — Immediate Digital Truncation Instruction .	456
15.59 EXT — Word Truncation Instruction .	457
15.60 PKG – Pack Immediate Shift Instruction	458
15.61 PKGLL – Pack Low Halfword Instruction	459
15.62 PKGHH – Pack High Halfword Instruction	460
15.63 PEXT.U8.E – 8-Bit Parallel Unsigned Extend Instruction	461
15.64 PEXT.S8.E – 8-Bit Parallel Signed Extend Instruction	463
15.65 PEXTX.U8.E – 8-Bit Parallel Unsigned Interleave Extend Instruction	465
15.66 PEXTX.S8.E – 8-Bit Parallel Signed Interleave Extend Instruction	467
15.67 NARL – Truncate and Assemble Low Instruction	468
15.68 NARH – Truncate and Assemble High Instruction	469
15.69 NARLX – Interleave and Concatenate Low Instruction	470
15.70 NARHX – Interleave and Concatenate High Instruction ..	471
15.71 CLIP.(U/S)32 – Signed Clip and Saturate Immediate Instruction (Un/With) ..	473
15.72 CLIP.(U/S)32 – Signed Clip and Saturate Immediate Instruction (Un/With) ..	475

15.73 PCLPI.(U/S)16 – 16-Bit Parallel Immediate (Without/With) Sign Clipping Saturation Instruction	477
15.74 PCLIP.(U/S)16 – 16-Bit Parallel (Without/With) Sign Clipping Saturation Instruction	480 15.75
PABS.S8.S – 8-Bit Parallel Absolute Value Instruction with Saturation ..	482 15.76 PABS.S16.S – 16-Bit
Parallel Absolute Value Instruction with Saturation ..	483 15.77 ABS.S32.S – 32-Bit Absolute Value
Instruction with Saturation ..	484 15.78 PNEG.S8.S – 8-Bit Parallel Negate Instruction with Saturation ..
485	15.79 PNEG.S16.S – 16-Bit Parallel Negate Instruction with Saturation ..
486 15.80 NEG.S32.S – Negate	with Saturation Instruction ..
487 15.81 DUP.8 – 8-Bit Operand Copy Instruction ..	488 15.82 DUP.16 – 16-
Bit Operand Copy Instruction ..	489 15.83 MULS.(U/S)32 – 32-Bit (Un/Yes) Signed Multiply-Accumulate-
Subtract Instruction ..	490 15.84 MULA.(U/S)32.S – 32-Bit (Un/Yes) Signed Multiply-Accumulate-Add with
Saturation Instruction ..	491 15.85 MULS.(U/S)32.S – 32-Bit (Un/Yes) Signed Multiply-Accumulate-Subtract
with Saturation Instruction ..	493 15.86 MUL.S32.H – 32-Bit Signed Multiply High 32 Instruction ..
494 15.87	MUL.S32.RH – 32-Bit Signed Multiply Upper 32 -Bit Instruction with Rounding ..
495 15.88 RMUL.S32.H – 32-Bit Signed	Fractional Multiply Upper 32-Bit Instruction with Rounding ..
496 15.89 RMUL.S32.RH – 32-Bit Signed	Multiply Accumulate Upper 32 -Bit Instruction with Saturation ..
497 15.90 MULA.S32.HS – 32-Bit Signed	Multiply Accumulate Upper 32-Bit Instruction with Saturation ..
499 15.91 MULS.S32.HS – 32-Bit Signed	Signed Subtract Multiply Accumulate Upper 32-Bit Instruction with Saturation ..
500 15.92 MULA.S32.RHS – 32-Bit	Signed Multiply Accumulate Upper 32-Bit Instruction with Rounding and Saturation ..
502 15.93	504 15.94 MULXL.S32 – 32-Bit Signed Lower Halfword Unaligned Multiply Instruction ..
505	15.95 MULXL.S32.R – 32-Bit Signed Lower Halfword Unaligned Multiply Instruction with Rounding ..
506	15.96 MULXH.S32 – 32-Bit Signed Upper Halfword Unaligned Multiply Instruction ..
507 15.97 MULXH.S32.R – 32-Bit Signed Upper Halfword Unaligned Multiply Instruction with Rounding ..	508 15.98 RMULXL.S32 – 32-Bit Signed Lower Halfword Unaligned Fractional Multiply Instruction ..
509 15.99 RMULXL.S32.R – 32-Bit Signed Lower Halfword Unaligned Fractional Multiply Instruction with Rounding ..	511 15.100
511 15.101	RMULXH.S32 – 32-Bit Signed Upper Halfword Unaligned Fractional Multiply Instruction ..
513 15.101	RMULXH.S32.R – 32-Bit Signed Upper Halfword Unaligned Fractional Multiply Instruction with Rounding ..
515 15.102 MULAXL.S32.S – 32-Bit Signed Lower Halfword Unaligned Multiply Accumulate Instruction with Saturation ..	516 15.103 MULAXL.S32.RS – 32-Bit Signed Lower Halfword Unaligned Multiply Accumulate Instruction with Rounding and Saturation ..
518 15.104 MULAXH.S32.S – 32-Bit Signed Lower Halfword Unaligned Multiply Accumulate Instruction with Saturation ..	519 15.105 MULAXH.S32.RS – 32-Bit Signed Upper Halfword Unaligned Multiply Accumulate Instruction with Rounding and Saturation ..
521 15.106 MULLL.S16 – 16-Bit Signed Lower Halfword Multiply Instruction ..	522 15.107 MULHH.S16 – 16-Bit Signed Upper Halfword Multiply Instruction ..
523 15.108 MULHL.S16 – 16-Bit Signed Upper and Lower Halfword Multiply Instruction ..	524 15.109 RMULLL.S16 – 16-Bit Signed Lower Halfword Fractional Multiply Instruction ..
525 15.110 RMULHH.S16 – 16-Bit Signed Upper Halfword Fractional Multiply Instruction ..	526 15.111
527 15.112	RMULHL.S16 – 16-Bit Signed Upper and Lower Halfword Fractional Multiply Instruction ..
528 15.112	MULAHH.S16.S – 16-Bit Signed Upper Halfword Multiply and Accumulate Instruction with Saturation ..
529	15.113 MULAHL.S16.S – 16-Bit Signed Upper and Lower Halfword Multiply and Accumulate Instruction with Saturation ..
531 15.114 MULALL.S16.E – 16-Bit Signed Lower Halfword Multiply and Accumulate Instruction with Extend ..	532

15.115 MULAHH.S16.E – 16-Bit Signed Upper Halfword Multiply Accumulate Instruction with Extend Operation ..	533
15.116 MULAHL.S16.E – 16-Bit Signed Upper and Lower Halfword Multiply Accumulate Instruction with Extend Operation ..	534
15.117 PMUL.(U/S)16 – 16-Bit Parallel (Un/Signed) Multiply Instruction ..	535
15.118 PMULX.(U/S)16 – 16-Bit Parallel (Un/Signed) Interleaved Multiply Instruction ..	537
15.119 PRMUL.S16 – 16-Bit Parallel Signed Fractional Multiply Instruction ..	538
15.120 PRMULX.S16 – 16-Bit Parallel Signed Interleaved Fractional Multiply Instruction ..	541
15.121 15.125 MULCA.S16.S – 16-Bit Signed Multiply Chain Add with Saturation Instruction ..	553
15.126 MULCAX.S16.S – 16-Bit Signed Multiply Chain Add with Saturation Instruction ..	554
15.127 MULCS.S16 – 16-Bit Signed Multiply Chain Subtract Instruction ..	555
15.128 MULCSR.S16 – 16-Bit Signed Reverse Multiply Chain Subtract Instruction ..	556
15.129 MULCSX.S16 – 16-Bit Signed Cross Multiply Chain Subtract Instruction ..	557
15.130 MULACA.S16.S – 16-Bit Signed Multiply Chain Add Accumulate with Saturation Instruction ..	558
15.131 MULACAX.S16.S – 16-Bit Signed Cross Multiply Chain Add Accumulate with Saturation Instruction ..	559
15.132 MULACS.S16.S – 16-Bit Signed Multiply Chain Subtract Accumulate with Saturation Instruction ..	560
15.133 MULACSX.S16.S – 16-Bit Signed Cross-Multiply -Chain-Subtract-Accumulate Instruction with Saturation ..	564
15.134 MULSCA.S16.S – 16-Bit Signed Cross-Multiply-Chain-Add-Accumulate Instruction with Saturation ..	566
15.135 MULSCAX.S16.S – 16-Bit Signed Cross-Multiply-Chain -Add-Accumulate Instruction with Saturation ..	567
15.136 MULACA.S16.E – 16-Bit Signed Cross-Multiply-Chain-Add-Accumulate Instruction with Extend ..	569
15.137 MULACAX.S16.E – 16-Bit Signed Cross-Multiply-Chain-Add-Accumulate Instruction with Extend ..	570
15.138 MULACS.S16.E – 16-Bit Signed Cross-Multiply-Chain-Subtract-Accumulate Instruction with Extend ..	571
15.139 MULACSR.S16.E – 16-Bit Signed Reverse Multiply Chain Subtract Accumulate Instruction with Extend Operation ..	572
15.140 MULACSRX.S16.E – 16-Bit Signed Cross Multiply Chain Subtract Accumulate Instruction with Extend Operation ..	573
15.141 MULACS.RX.S16.E – 16-Bit Signed Cross-Multiply-Chain-Subtract Instruction with Extend Operation ..	574
15.142 MULSCA.S16.E – 16-Bit Signed Cross Multiply Chain Add Accumulate Subtract Instruction with Extend Operation ..	575
15.143 MULSCAX.S16.E – 16-Bit Signed Cross Multiply Chain Add Accumulate Subtract Instruction with Extend Operation ..	576
15.144 PSABSA.U8 – 8-Bit Parallel Unsigned Subtract Absolute Value Chain Add Instruction ..	577
15.145 PSABSAA.U8 – 8-Bit Parallel Signed Unsigned Subtract Absolute Value Chain Add Accumulate Instruction ..	578
15.146 DIVSL – Signed Long Divide Instruction	578
15.147 DIVUL – Unsigned Long Divide Instruction	579
15.148 MULACA.S8 – 8-Bit Parallel Signed Multiply Chain Add Accumulate Instruction ..	580
15.149 BLOOP – Loop Body Acceleration Instruction	581
15.150 LDBI.W – Load Word with Incrementing Address Instruction	582
15.151 LDBI.H – Load Unsigned Halfword with Incrementing Address Instruction	583
15.152 LDBI.HS – Load Signed Halfword with Incrementing Address Instruction	584
15.153 LDBI.B – Load Unsigned Byte with Incrementing Address Instruction	584
15.154 LDBI.BS – Load Signed Byte with Incrementing Address Instruction	585
15.155 PLDBI.D – Load Double Word with Incrementing Address	586
15.156 STBI.W – Store Word with Incrementing Address	587
15.157 LDBI.L – Load Long Word with Incrementing Address	589

15.157 STBI.H – Store Halfword with Increment Address Instruction .. 589 15.158 STBI.B – Store Byte with Increment Address Instruction .. 590 15.159 LDBIR.W – Load Word with Increment Register Address Instruction .. 591 15.160 LDBIR.H – Load Unsigned Halfword with Increment Register Address Instruction .. 592 15.161 LDBIR.HS – Load Signed Halfword with Increment Register Address Instruction .. 593 15.162 LDBIR.B – Load Unsigned Byte with Increment Register Address Instruction .. 594 15.163 LDBIR.BS – Load Signed Byte with Increment Register Address Instruction .. 594 15.164 PLDBIR.D – Load Doubleword with Increment Register Address Instruction .. 595 15.165 STBIR.W - WORD STORE INSTRUCTION WITH AUTO-INCREMENT REGISTER ADDRESS .. 597 15.166 STBIR.H - HALF-WORD STORE INSTRUCTION WITH AUTO-INCREMENT REGISTER ADDRESS .. 597 15.167 STBIR.B - BYTE STORE INSTRUCTION WITH AUTO-INCREMENT REGISTER ADDRESS .. 598

Chapter 1 Overview

1.1 Introduction

E804 is a 32-bit high-efficiency embedded CPU core for the control field, with many characteristics such as low cost, low power consumption, and high code density.

E804 adopts a 16/32-bit mixed encoding instruction system and is designed with a streamlined and efficient 3-stage pipeline.

E804 provides a variety of configurable functions, including hardware floating-point unit, on-chip cache, DSP acceleration unit, trusted protection technology, on-chip tightly coupled IP, etc. Users can configure according to application needs. In addition, E804 provides multiple bus interfaces and supports flexible configuration of system bus, instruction bus, and data bus. E804 has made special optimizations for memory copy applications to achieve the ultimate memory copy performance. In addition, E804 has made special acceleration for interrupt response, and the interrupt response delay only takes 13 cycles.

1.2 Features

The main features of the E804 architecture and programming model are:

- Reduced Instruction Set Processor Architecture (RISC);
- 32-bit data, 16-bit/32-bit mixed encoding instructions;
- 16 32-bit general-purpose registers;
- 3-stage assembly line;
- Launch in sequence, execute in sequence, and retire in sequence;
- Configurable multi-bus interface;
- Configurable cache;
- Configurable hardware floating-point unit;
- Configurable DSP processing unit;
- Configurable trusted protection technology;
- Configurable memory protection units (0-8);
- Configurable hardware multiplier that supports fast multiplication results in 1 cycle;
- Configurable tightly coupled IP, including vectored interrupt controller and timers;

- Supports various processor clock to system clock ratios;
- Interrupt response latency is only 13 processor cycles.

The main features of the E804 microarchitecture are:

- Static branch prediction;
- Support hardware division;
- Supports continuous memory access;
- Supports big endian and little endian.

1.3 Configurable Options

E804 configurable options are shown in Table 1.1 .

Table 1.1: E804 configurable options

Configurable Unit	Configuration	detailed
Hardware Multiplier	options None/Yes	If configured, the multiplication result is generated in 1 cycle, otherwise it takes 1-32 cycles Period completed.
Memory protection unit 0 to 8 entries		It can be configured as 0-8 entries, where 0 means no memory preservation. Protection unit.
Cache No/2K/4K/8K /16K Floating point unit No/Yes		It can be configured as 2KB, 4KB, 8KB, and 16KB. Single-precision floating-point hardware processing.
DSP acceleration unit No/Yes		Hardware acceleration for applications such as audio and motor. The processor is configured with 32 32-bit general purpose registers.
Trusted protection technology No/Yes		Configure this technology, combined with Zhongtian Micro's SoC platform technology/system System software will provide system security protection functions.
Instruction bus	Fixed inclusion	Only supports AHB-Lite, direct output (Non-Flop-out) mode.
Data bus System	Fixed includes supports only AHB-Lite, direct output (Non-Flop-out) mode.	
bus	AHB/AHB-Lite compatible. Can be configured as AHB compatible or AHB-Lite compatible. Discuss.	Only supports direct output (Non-Flop-out) mode.
Vectored interrupt controller None/INT8/INT16 /INT24/INT32		Supports nested processing of hardware interrupts. Supports 8/16/24/32 interrupts source.
System timer	No/Yes	Used for timing.

1.4 Design for Testability

E804 supports scan chain test (SCAN) and built-in self test (BIST). The scan chain test is used to test whether there are manufacturing errors in the combinational and sequential logic inside the processor, and the built-in self test is used to test whether there are manufacturing errors in the cache.

The number of scan chains of E804 can be specified by the customer.

1.5 Debuggability Design

E804 uses the JTAG standard (2-wire) to design the hardware debug interface. E804 supports all common debug functions, including soft breakpoints, memory Breakpoints, register check and memory check and modification, instruction single-step tracing and multi-step tracing, program flow tracing, etc. For details, see the debug interface.

1.6 Naming conventions

1.6.1 Symbols

The standard symbols and operators used in this document are shown in the following table.

1.6.2 Terminology

- Logic 1 refers to the level value that corresponds to Boolean logic true.
- Logic 0 refers to the level value that corresponds to Boolean logic false.
- Setting means making one or several bits reach the level value corresponding to logic 1.
- Clearing means making one or more bits reach the level corresponding to logic 0.
- Reserved bits are reserved for function expansion and their value is 0 unless otherwise specified.
- A signal is an electrical value that conveys information through its state or transitions between states.
- A pin represents an external electrical physical connection, and the same pin can be connected to multiple signals.
- Enabling means making a discrete signal valid:
An active-low signal switches from a high level to a low level; an active-high signal switches from a low level to a high level.
- Disabling means changing the state of a signal that is in the enabled state:
An active-low signal switches from a low level to a high level; an active-high signal switches from a high level to a low level.
- LSB stands for Least Significant Bit and MSB stands for Most Significant Bit.
- Signals, bit fields, and control bits all use a common convention for representation.

符号	功能
+	加
-	减
*	乘
/	除
>	大于
<	小于
=	等于
\geq	大于或等于
\leq	小于或等于
\neq	不等于
.	与
+	或
\oplus	异或
NOT	取反
:	连接
\Rightarrow	传输
\Leftrightarrow	交换
\pm	误差
0b0011	二进制数
0x0F	十六进制数

Figure 1.1: Symbol list

- The identifier is followed by a number indicating the range, from high to low, indicating a group of signals, for example, `addr[4:0]` indicates a group of address buses, The highest bit is `addr[4]` and the lowest bit is `addr[0]`.

• A single identifier refers to a single signal, for example `pad_cpu_reset_b` refers to a single signal.

Adding a number indicates a certain meaning, for example, `addr15` means the 16th bit in a bus.

Chapter 2 Microarchitecture

2.1 Structure diagram

The E804 structure block diagram is shown in Figure 2.1.

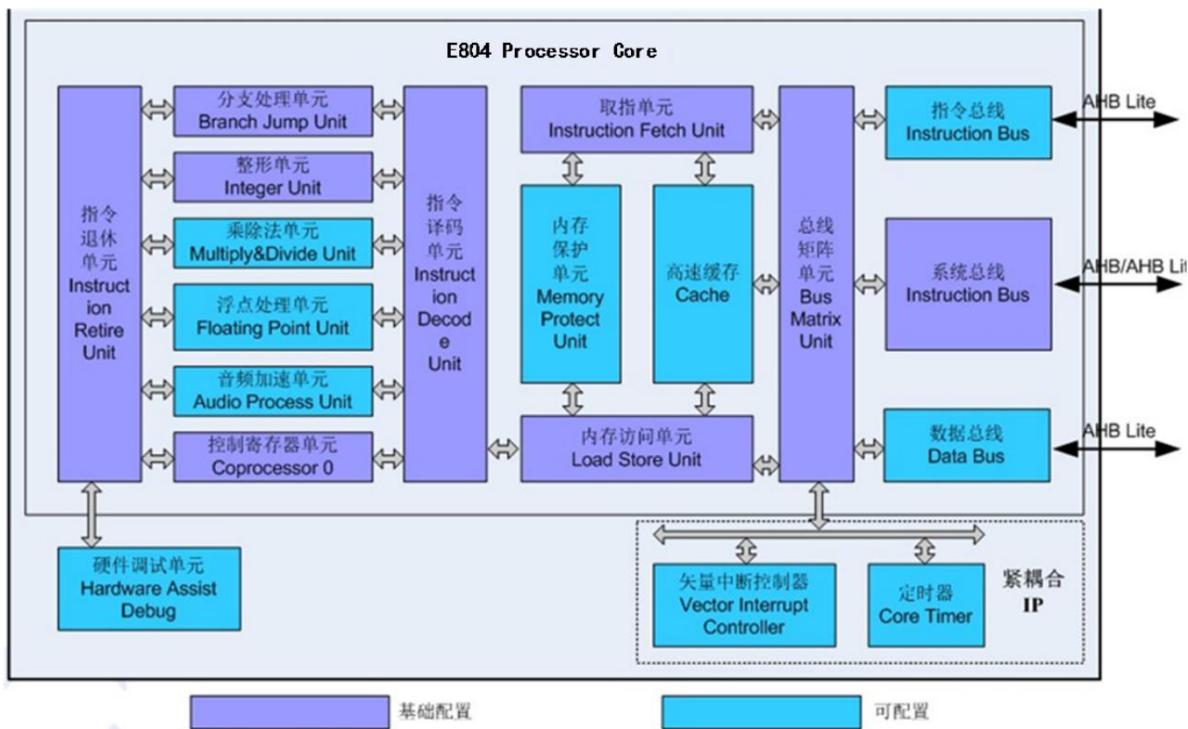


Figure 2.1: E804 block diagram

The instruction fetch unit is responsible for accessing and extracting instructions. It can obtain 32 bits of data per clock cycle, that is, it can extract one 32-bit instruction per cycle.

In addition, the instruction fetch unit predicts branch instructions and splits complex instructions.

The instruction decoding unit decodes the instruction, accesses the general registers and completes the issuance of the instruction.

The branch processing unit checks the branch prediction instructions and processes the register jump instructions. The shaping unit is responsible for the execution of ALU instructions. The control register unit mainly processes the control register related instructions (MTCR/MFCR). The execution delay of branch instructions, ALU instructions and control register instructions is 1 cycle.

The multiplication and division unit is responsible for the operation of multiplication, multiplication accumulation and division instructions. In terms of multiplication implementation, users can configure two methods: slow multiplication and fast multiplication.

Among them, slow multiplication has a low cost, but requires 1-34 cycles to produce a multiplication result; fast multiplication supports one cycle to produce a multiplication result, but the cost is high. Division requires 3-35 cycles to produce an operation result.

The memory access unit is responsible for the sequential execution of load/store instructions and supports signed/unsigned byte/halfword/word access.

Q. Load and store instructions can be executed consecutively to optimize performance.

The retirement unit collects and processes the execution results of instructions, completes the write-back of results, and handles interrupts and exceptions.

Responsible for interaction with the hardware debugging unit.

E804 implements a configurable memory protection unit (MPU) to protect memory access to sensitive data, supporting 1-8 configurable entries.

The user can define the access attributes of each protection zone by setting the memory protection unit.

E804 implements a configurable cache unit, supporting 2KB-32KB configurable. The cache adopts a four-way set associative structure, supporting

There are two working modes: write-back and write-through.

E804 has designed a multi-level bus interface, supporting three bus interfaces: instruction bus, data bus and system bus. The instruction/data bus realizes high-performance instruction/data access. Users can design the local memory subsystem through the instruction/data bus to improve the overall performance of the system. The instruction/data bus supports the AHB-Lite interface and only supports direct output (Non-Flop-out, which has certain requirements for interface timing). The system bus also only supports direct output, and supports both AMBA2 AHB protocol and AHB-Lite protocol. The CPU clock and the system peripheral bus clock must maintain a 1:1 relationship.

In order to facilitate system integration and development, E804 has designed a tightly coupled IP, including a vector interrupt controller and a system timer, which can be configured by the user according to the application needs. The vector interrupt controller supports up to 32 interrupt sources, supports both level and pulse interrupt modes, and supports interrupt nesting in hardware. The system timer provides a 24-bit cyclic down counter, which can be timed according to the CPU clock or external reference clock, and supports the system timer to generate interrupts. For more information about tightly coupled IP, please refer to [Tightly Coupled IP](#)

E804 is designed for floating-point applications and has a configurable floating-point processing unit. The floating-point processing unit only implements single-precision floating-point operations.

E804 is designed with a configurable DSP acceleration unit for applications such as audio and motors. The DSP acceleration unit implements DSP acceleration instruction subset.

E804 is designed for security protection and has configurable trusted protection technology. It cooperates with our company's SoC platform and system software to provide system For details, please refer to the Trusted Protection Technical Manual.

2.2 Introduction to the pipeline

E804 is designed with a streamlined and efficient 3-stage pipeline, which includes instruction fetch, decoding and execution. Most integer instructions (including branch instructions, integer operation instructions, load and store instructions and DSP acceleration instructions) can be completed in the 3-stage pipeline. For some floating-point instructions, an additional execution cycle is required. The E804 pipeline is shown below.

Table 2.1: E804 pipeline introduction

Pipeline name	abbreviation	Pipeline effect
	IF	<ol style="list-style-type: none"> 1. Initiate an instruction fetch request and process the returned instruction data; 2. Instruction pre-decoding; 3. Static branch prediction; 4. Complex instruction splitting.
Decoding	ID	<ol style="list-style-type: none"> 1. Instruction decoding; 2. Instruction correlation analysis; 3. Command transmission; 4. Execute branch jump instruction; 5. Calculate the address of the load-store instruction and initiate an access request.
implement	EX	<ol style="list-style-type: none"> 1. Execute and complete integer instructions; 2. Execute floating-point instructions; 3. Execute DSP acceleration instructions; 4. Complete the load and store instructions; 5. Write back the instruction execution result; 6. Order to retire; 7. Exception and interrupt handling.
Execution 2	EX2	Complete some floating-point instructions.

2.3 Floating Point Processing Unit

E804 is designed for floating-point applications with a low-cost, energy-efficient floating-point processing unit. Its main features include:

- Fully compatible with ANSI/IEEE Std 754 floating point standard (with system software support);
- Only supports single-precision floating-point operations;
- Supports four rounding modes: rounding towards zero, rounding towards positive infinity, rounding towards negative infinity and rounding to the nearest;
- Supports both trapping and non-trapping processing modes for floating-point exceptions;
- Support accurate handling of floating point exceptions;
- Support floating-point hardware division and square root;
- 16 independent single-precision floating-point registers;
- Single-issue architecture, processing one floating-point instruction per cycle;

- Support in-order issuance, in-order execution, and in-order write-back of floating-point arithmetic instructions;
- Contains three independent execution pipelines, namely floating-point ALU, floating-point multiplication and floating-point division square root;
- Optimized execution delay technology, all instructions except floating-point division square root can be executed within 1-2 clock cycles.

2.4 DSP Expansion Unit

E804 is designed with low-cost, high-efficiency DSP processing unit for applications such as audio and motor. Its main design features include:

- Basically covers all operations defined by the existing ARM/Andes DSP instruction subset;
- Extreme acceleration technology for short cycles;
- Data pre-fetching techniques for arrays/matrices.

Key features include:

- A subset of DSP instructions operates on general-purpose registers and expands them to 32;
- Supported operation types include multiplication, multiply-accumulate, shift, addition, subtraction, data packing, comparison, storage access, etc.;
- Supports single instruction multiple data operations. The supported data types for single instruction multiple data operations are word/half word, and the supported operation types are multiplication, multiplication and accumulation, shift, add, subtract;
- The maximum operation width is 64 bits, supporting a single instruction to access up to 4 general-purpose registers and write back 2 general-purpose registers;
- Some operations support both rounding and non-rounding value acquisition modes;
- Some operations support both saturated and non-saturated value acquisition modes.
- Some operations support both integer and decimal operations;
- Single-issue architecture, processing one DSP extension instruction per cycle;
- Support in-order issuance, in-order execution, and in-order write-back of DSP extended instructions;
- Optimized execution delay technology, all instructions can be completed in 1 clock cycle.

2.5 Trusted Protection Technology

E804 is designed for security and uses trusted protection technology that can be used for system security protection. Its main features include:

- Based on the same physical processor core, two worlds are virtualized, namely the secure world and the non-secure world;
- Support software-based spatial partitioning of memory and I/O space into two worlds;
- Support trusted interrupts;
- Support trusted debugging;
- Support trusted boot;

For more information, see the Trusted Protection Technical Manual.

2.6 Tightly coupled IP architecture

In order to improve the system integration of E804 and facilitate user integration and development, E804 implements a series of systems closely related to the CPU core.

These IPs are collectively referred to as Tightly Coupled IP (TCIP). The Tightly Coupled IP of E804 includes the system timer CoreTim and the vector interrupt controller VIC.

Key features of the vectored interrupt controller include:

- The number of interrupts is hardware configurable, supporting 1/2/4/8/16/32 interrupt sources;
- Interrupt priority is software definable, 4 levels of priority can be defined;
- Supports hardware interrupt nesting;
- Supports both level and pulse interrupt source signals.

The main characteristics of the system timer include:

- 1 24-bit counter;
- Supports selectable input clock, you can choose CPU clock or external input clock;
- Support interrupt generation;

For more information, see Tightly Coupled [IP](#).

Chapter 3 Programming Model

3.1 Working Mode and Register View



Figure 3.1: Programming model

E804 defines two processor operating modes: normal user mode and super user mode. The two operating modes correspond to different operating permissions.

It is mainly reflected in the following aspects:

1. Access to registers;
2. Use of privileged instructions;

3. Access to memory space.

Programs access registers according to their permissions. Ordinary user programs are only allowed to access registers designated for ordinary user mode; programs working in super System software in super user mode can access all registers and use control registers to perform super user operations.

The management of access permissions can prevent user programs from accessing privileged information, and the operating system coordinates the behavior of ordinary user programs to provide services to ordinary users. Program provides management and services.

Most instructions can be executed in both modes, but some privileged instructions that have a significant impact on the system can only work in supervisor mode.

Next. Privileged instructions include STOP, DOZE, WAIT, MFCR, MTCR, PSRSET, PSRCLR, and RTE.

E804 is designed with a memory protection unit. The operating system can manage and control memory access by setting the memory protection unit.

Ordinary user programs are only allowed to access memory spaces that are open to ordinary user mode.

The processor's operating mode is controlled by the S bit in the Processor Status Register (PSR). When the S bit in the PSR is set, the processor operates in In supervisor mode, when the S bit is cleared to 0, the processor operates in normal user mode.

During exception handling, the processor switches the mode from normal user mode to supervisor mode and stores the current PSR value in the exception

The processor status register (EPSR) is always preserved, and then the S bit in the PSR is set to force the processor into supervisor mode.

After the processing is completed, in order to return to the previous working mode, the system function executes RTE (return from exception) and re-fetches instructions from the place where the exception occurred.

As a system call instruction, TRAP#n instruction provides a controllable interface for ordinary user programs to access operating system service programs.

The program can generate a system call exception through TRAP#n and force the processor to enter supervisor mode.

The registers that can be operated in normal user mode include 16 32-bit general-purpose registers, a 32-bit program counter (PC), a conditional/carry bit (C) and other registers (determined by configuration). Among them, the C bit is located at the lowest bit of the PSR and is the only bit in the PSR that can be used in normal user mode.

In addition to the registers accessible in normal user mode, supervisor mode also includes registers containing processor control and status information.

A set of exception shadow registers EPSR and EPC used to save PSR and PC when an exception occurs, and a set of exception shadow registers EPSR and EPC used to save interrupt

The base address register VBR of the vector table and other related control registers (determined by configuration).

3.2 General Registers

The basic configuration of E804 includes 18 32-bit general registers: R15~R0, R14', R28.

The number of general registers is expanded to 33: R31~R0, R14'.

Table 3.1: Normal User Programming Mode Registers

name	Function
R0	Undefined, the first parameter in the function call.
R1	Unsure, the second parameter when calling the function.
R2	Unsure, the third parameter when calling the function.
R3	Unsure, the fourth parameter in the function call.
R4	uncertain.
R5	uncertain.
R6	uncertain.
R7	uncertain.

Continue on next page

Table 3.1 – Continued from previous page

R8	uncertain.
R9	uncertain.
R10	uncertain.
R11	uncertain.
R12	uncertain.
R13	uncertain.
R14 (User SP) Stack pointer (normal user mode).	
R14'(Spv SP) Stack pointer (Supervisor mode).	
R15	Link register.
R16	uncertain.
R17	uncertain.
R18	uncertain.
R19	uncertain.
R20	uncertain.
R21	uncertain.
R22	uncertain.
R23	uncertain.
R24	uncertain.
R25	uncertain.
R26	uncertain.
R27	uncertain.
R28	uncertain.
R29	uncertain.
R30	uncertain.
R31	uncertain.

General registers are used to store instruction operands, instruction execution results, and address information.

The DSP instruction set and DSP instruction subset can operate the general registers R31~R0 at the same time.

E804 has designed stack pointer R14 for normal user mode and supervisor mode. Normal user mode can only access the user program R14 (User SP); In super user mode, system software can access not only R14 (Spv SP) of system programs, but also R14 (Spv SP) of ordinary R14 (User SP) of the user program. In supervisor mode, the index of general register R14 will use the register of supervisor mode. R14 (Spv SP). If the user wants to access R14 (User SP) in supervisor mode, he can access CR<14,1> through MFCR/MTCR.

become.

3.2.1 Condition Code/Carry Flag

The condition code/carry flag represents the result of an operation. The condition code/carry flag can be set as a result of a comparison operation instruction, or as a result of some other high-precision arithmetic or logical instruction.

3.3 System Control Registers

System programmers use supervisor mode to set up system operating functions, I/O control, and other restricted operations.

The system control registers that can be operated in supervisor mode are as follows:

- Processor Status Register (PSR);
- Vector Base Register (VBR);
- Exception Preservation Program Counter (EPC);
- Exception Preserved Processor Status Register (EPSR);
- Product ID register (CPUID);
- Cache Configuration Register (CCR);
- Highly configurable access rights configuration register (CAPR)*;
- Protected Area Control Register (PACR)*;
- Protection Zone Select Register (PRSR)*;
- Implicit Operation Register (HINT).

Note: * Only available in certain configurations.

Table 3.2: Supervisor Programming Model Control Registers

name	Type	Control Register Number/Address	Description
PSR	Read/write	CR<0,0>	Processor Status Register.
VBR	Read/write	CR<1,0>	Vector Base Address Register.
EPSR	Read/write	CR<2,0>	Exception Preservation Processor Status Register.
EPC	Read/write	CR<4,0>	Exception preserves the program counter.
CPUID	Read	CR<13,0>	product serial number register.
CCR	Read/write	CR<18,0>	Cache Configuration Register.
CAPR	Reading/writing	CR<19,0>	can be used to configure the high-level access rights configuration register.
PACR	Read/write	CR<20,0>	Protection Zone Control Register.
PRSR	Read/write	CR<21,0>	Protection Zone Select Register.
CHR	Reading/writing	CR<31,0>	implicitly operates the register.
R14(User SP)	Read/write	CR<14,1>	Stack Pointer Register.

3.3.1 Processor Status Register (PSR, CR<0,0>)

The Processor Status Register (PSR) stores the current processor status and control information, including the C bit, interrupt valid bit, and other control bits.

In Supervisor mode, software can access the Processor Status Register (PSR). The control bits indicate the following states for the processor:

mode (TM_{y1:0}), supervisor mode or normal user mode (S bit). They also specify whether the interrupt request is valid.

31	30	24	23	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	-		VEC[7:0]	TM[1:0]	0	0	0	MM	EE	IC	IE	0	0	0	0	0	0	0	C	
1	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Figure 3.2: Processor Status Register

S - Superuser mode setting bit:

- When S is 0, the processor operates in normal user mode;
- When S is 1, the processor operates in supervisor mode;

The S bit is set by hardware when the processor is restarted and enters an exception handler.

VEC[7:0] - Exception vector value:

When an exception occurs, VEC[7:0] is used to calculate the exception service routine entry address and will be cleared when reset.

TM[1:0] - Tracking mode bits:

In instruction tracking mode, after each instruction is executed, E804 will enter the tracking exception service routine; in jump tracking mode, when an instruction containing a jump (whether it is a jump or not) is executed, E804 will enter the tracking exception service routine. These bits are cleared by hardware when reset and when entering the exception service routine. The following table shows the TM_{y1:0} encoding and the corresponding working mode.

Table 3.3: TM[1:0] encoding and corresponding operating mode

Value	
00	Description 00 Normal
01	execution mode 01 Instruction
10	trace mode 10
11	Undefined 11 Jump trace mode

For specific operations in tracking mode, please refer to exception handling .

MM-Misaligned exception mask bit:

- When MM is 0, the address misalignment exception of load or store will occur normally and will not be masked, that is, the exception will be responded to;
- When MM is 1, the address misalignment exception of load or store will be masked and the data access request will be split into multiple aligned operations;

The bit will be cleared by reset.

EE-Exception valid control bit:

- When EE is 0, exceptions are invalid. Once any exception other than a normal interrupt occurs, it will be considered unrecoverable by E804.

Complex abnormalities:

- When EE is 1, exceptions are valid and all exceptions will respond normally and use EPSR and EPC.

IC-interrupt control bit in instruction:

- When IC is 0, interrupts can only be responded to between instructions;
- When IC is 1, it indicates that the interrupt can be serviced during the execution of a long, multi-cycle* instruction;

It will be reset and will not be affected by other exceptions.

IE-interrupt valid control bit:

- When IE is 0, the interrupt is invalid;
- When IE is 1, the interrupt is valid.

This bit is cleared by reset and is also cleared when entering an exception service routine.

C-Condition code/Carry bit:

This bit is used as a conditional bit for some instructions. It is undefined after reset and after being copied to EPSR.

Note: Some multi-cycle instructions including LDM, STM, PUSH, POP, IPUSH, IPOP, LDQ32, STQ32 can be interrupted without waiting for them to complete, thus shortening the interrupt response delay. Multi-cycle instruction NIE cannot respond to interrupts, and NIR only responds to interrupts at the end of instruction execution and cannot be interrupted by the PSR (IC) bit.

3.3.2 Update PSR

The PSR can be updated in several different ways, and the effects of changes to the control bits in the PSR vary. The PSR can usually be modified by exception responses, exception handling, and execution of PSRSET, PSRCLR, RTE, MTCR instructions. These modifications are implemented in four ways:

1. Update PSR for exception response and exception handling:

Updating the PSR is part of the exception response and will update the S, TM, VEC, IE and EE bits in the PSR.

2. RTE instruction updates PSR:

Updating the PSR As part of the execution of the RTE instruction, all bits in the PSR are modified.

3. MTCR instruction updates PSR:

If the destination register is CR<0,0>, the PSR will be updated as part of the MTCR instruction execution. This update may change the value of all bits in the PSR, and the subsequent instructions, exception events and interrupt responses will use the new PSR value.

4. PSRCLR, PSRSET instructions update PSR:

Updating the PSR As part of the execution of the PSRCLR and PSRSET instructions, subsequent instructions, exceptions, and interrupt responses will use the new PSR value.

3.3.3 Vector Base Register (VBR, CR<1,0>)

The VBR register is used to store the base address of the exception entry address table. The reset value of VBR is 0X00000000.

31		10 9	0
VECTOR BASE		Reserved	
0		0	

Figure 3.3: Base vector register

3.3.4 Exception Reserved Registers (CR<2,0>, CR<4,0>)

The EPSR, EPC, registers are used to save the current processor state when an exception occurs. For more detailed information, please refer to Exception Handling .

3.3.5 Product ID Register (CPUIDR, CR<13,0>)

This register is used to record the CPU information, including the CPU model, configuration, etc. The product serial number register is read-only, and its reset value is determined by the product itself.

3.3.6 Implicit Operation Register (CHR, CR<31,0>)

The processor implicit operation register is a collection of some implicit operations of the processor, including soft reset operations and the use of some acceleration functions of the processor. able.

31	16 15 14		5 4 3 2 1 0		
Reset value[15:0]		0	IAE RPE IPE BE 0		
HS_EXP —					
0	0	0	0 0 0 0 0 0		

Figure 3.4: Implicit Operation Register

Reset_value-soft reset enable domain:

When Reset_Value is written to 16'hABCD, the processor reset operation will be triggered. The processor resets itself and indicates that a software reset has occurred to the processor through the processor pin sysio_pad_srst. Sysio_pad_srst maintains one system clock cycle.

HS_EXP-Hardware stack exception indicator:

When the processor supports Trusted Execution Technology, indicates that an exception occurred in the hardware push operation of the trusted world.

IAE-interrupt response acceleration control bit:

- When IAE is 0, the interrupt response instructions NIE/IPUSH will be executed sequentially;
- When IAE is 1, the interrupt response instruction NIE/IPUSH will be executed speculatively to speed up the interrupt response.

This bit will be cleared by reset.

The RPE function returns the acceleration control bits:

- When RPE is 0, the function return instruction RTS will be executed sequentially;
- When RPE is 1, the function return instruction RTS will be executed speculatively, speeding up the function return.

This bit will be reset to 1.

IPE-instruction prefetch acceleration control bit:

- When IPE is 0, the instruction prefetch acceleration function is not enabled;
- When IPE is 1, the instruction prefetch acceleration function is enabled, which improves the efficiency of instruction prefetching.

This bit will be cleared by reset.

BE-bus Burst transmission control bit:

- When BE is 0, the bus transfer type only supports SINGLE;
- When BE is 1, the system bus supports the Burst transfer type.

This bit will be cleared by reset.

3.3.7 Other Control Registers

Other control registers of E804 include:

- Cache Configuration Register (CCR)*;
- Highly configurable access rights configuration register (CAPR)*;
- Protected Area Control Register (PACR)*;
- Protection Zone Select Register (PRSR)*;

Among them, the cache configuration register (CCR), the highly cacheable access permission configuration register (CAPR)*, the protection zone control register (PACR)*, and the protection zone selection register (PRSR)* are control registers related to the memory protection unit settings. They are effective when the CPU configures the memory protection unit. For specific control register definitions, please refer to Memory Protection .

3.3.8 Normal user mode general register 14 (R14 (User SP), CR<14,1>)

The general register 14 of normal user mode is mapped to the control register CR<14,1>.

3.3.9 Interrupt Pointer Register (R14(Int_SP),CR<15,1>)

E804 can optionally configure the interrupt pointer (Int_SP) to share the interrupt stack space under different threads to save the total stack overhead. If configured and ISE is enabled, the pointer is only used in the interrupt (excluding tspending), that is, when the vector number is greater than or equal to 32, and the original pointer is used at hardware stacking and other times.

In the non-interrupt supervisor mode, this register is mapped to the control register CR<15,1>, that is, the supervisor can access CR<15,1>

Access the interrupt stack pointer register.

3.4 Data size

The data endianness is proposed relative to the format of data storage in the memory. The high address byte is stored in the low bit of the physical memory and is defined as the big endian; the high address byte is stored in the high bit of the physical memory and is defined as the little endian.

The high order bits of address bytes stored in physical memory are defined as little endian.

E804 supports the little-endian format and the V1 version big-endian format (CSKY CPU has designed two versions of big-endian and small-endian formats, V1 and V2), and does not support

In little-endian mode, no matter the access size of the access instruction is byte/halfword/word, the least significant byte data always corresponds to the address.

The lowest bit of the address; in the big-endian mode of version V1, the data needs to be determined according to the access size (byte/half word/word) of the access instruction.

Table 3.4: Little-endian/big-endian memory access modes

Access size	Access address	Little	V1 Big Endian
	A	D3D2D1D0 D3D2D1D0	
endian word Half		D1D0	D3D2
word A Half word A+2		D3D2	D1D0
Byte A Byte A+1		D0	D3
Byte A+2 Byte A+3		D1	D2
		D2	D1
		D3	D0

3.5 Data Alignment Access

The data does not support unaligned access. When the CPU detects that an unaligned access occurs, it will enter an unaligned access exception. For details, refer to Unaligned Access exception (vector offset 0X4).

E804 supports masking unaligned access exceptions by configuring the MM bit in the processor status register PSR. For details, refer to the processor status register Status Register (PSR, CR<0,0>).

3.6 System Address Mapping

In order to facilitate system integration and development, E804 divides the address and function of the 4GB memory space.

The access (instruction or data access) address is arbitrated and distributed to different buses. The system address division and function recommended by E804 are shown in the following table.

In order to make SOC design more flexible, chip manufacturers can also define the division of bus address space by configuring pad_bmu_iahbl_base

, pad_bmu_iahbl_mask, pad_bmu_dahbl_base, pad_bmu_dahbl_mask to achieve the purpose. For specific configuration methods, refer to

See bus matrix and bus interface .

Table 3.5: Address division diagram

Name	Memory address space functions	
Command Bus	0x00000000~0x1FFFFFFF stores instructions	
Data Bus	0x20000000~0x3FFFFFFF stores data	
System Bus	0x40000000~0xDFFFFFFF Function is defined by the system developer	Can store instructions, data and system IP
Tightly coupled IP bus 0xE0000000~0XEFFFFFFF	Tightly coupled IP access address space	
System bus 0xF0000000~0xFFFFFFFF	Function is defined by the system developer	Can store instructions, data and system IP

The bus matrix unit of E804 only arbitrates based on the memory access address, regardless of the memory access type (instruction access/data access/closed Coupled IP access). Both instruction fetch requests and data fetch requests can access any bus and memory space. The programmer must ensure that memory access The correctness of the address ensures successful access to the target memory. For example, in the E804 with the instruction bus and data bus, if the instruction access If the address falls on [0x20000000, 0x40000000], the bus matrix unit will distribute the instruction access request to the data bus and Access in memory.

The E804 is highly configurable, and users can choose to implement any of the command bus, data bus, and tightly coupled IP bus according to the application. The impact of one or more bus configurations on memory access is shown in the following table.

Table 3.6: Address division diagram

Name	Configurability	Configuration Effects on Memory Access	
Command Bus	fixed inclusion	Memory accesses at [0x00000000~0x1FFFFFFF] will be dispatched to the instruction On the bus	
Data Bus	Fixed inclusion	Memory accesses at [0x20000000~0x3FFFFFFF] will be distributed to the data On the bus	
System bus	Fixed inclusion	Access to the system bus space is distributed to the system bus	
Tightly coupled to IP bus	configurable	To achieve memory access at [0xE000 0000~0XEFFFFFFF] will Distributed to a tightly coupled IP bus	
		Memory access at [0xE0000000~0XEFFFF FFF] is not implemented Will be distributed to the system bus	
System Bus	Fixed inclusion	Access to the system bus space is distributed to the system bus	

The above accesses will only occur in non-cacheable areas or in case of cache misses.

3.7 Memory Access Order

E804 is designed with multiple bus interfaces, so system integrators can connect different memories to the bus.

To facilitate user development, the E804 hardware design ensures that memory access instructions are executed in strict accordance with the order of assembly instructions.

This is done in one go, avoiding the need for users to guarantee the order of memory accesses in software.

For example, in the following two instruction sequences, Ins A accesses the memory on the system bus, and Ins B accesses the memory on the data bus. Assume that the system bus access delay of the memory is much longer than that of the data bus memory. In order to ensure that the instructions are completed in the order of the program, the hardware ensures that the Ins A instruction is executed. After the completion of the process, the Ins B instruction is allowed to execute.

Ins A: ld r4, (r7)

Ins B: ld r5, (r14)

Chapter 4 Exception Handling

Exception handling (including instruction exceptions and external interrupts) is an important technology of the processor. When certain exception events occur, the processor will stop the execution of the current instruction and switch to the processing of the exception event. These events include external interrupts, instruction execution errors, and system call requests. This chapter mainly describes the exception types, exception priorities, exception vector table, exception return, and bus error recovery.

4.1 Exception Handling Overview

Exception handling is the process of the processor switching from normal program processing to a specific exception handling program based on internal or external exception events. External events that cause exceptions include: interrupt requests from external devices, read/write access errors, and hardware restarts; internal events that cause exceptions include: illegal instructions, misalignment errors, privileged exceptions, and instruction tracing. TRAP and BKPT instructions will also generate exceptions when they are executed normally. In addition, illegal instructions, misaligned addresses accessed by LD and ST, and privileged instructions executed in user mode will all generate exceptions. Exception handling uses the exception vector table to jump to the entrance of the exception service program.

The key to exception handling is to save the state of the current instruction of the CPU when an exception occurs, and restore the state before exception handling when exiting the exception handling. Exceptions can be identified at each stage of the instruction pipeline, and the following instructions will not change the state of the CPU. Exceptions are handled at the boundary of the instruction, that is, the CPU responds to the interrupt when the instruction retires, and saves the address of the next instruction to be executed when exiting the exception handling. Even if the exception is identified in the instruction fetch or decode stage, the exception will not be processed until the corresponding instruction retires. E804 determines which instruction address the exception address register stores based on whether the instruction is completed when the exception is identified. For example, if the exception event is an external interrupt service request, the interrupted instruction will retire normally and change the state of the CPU, and the address of its next instruction will be saved in the exception address register as the entry of the instruction when the interrupt returns; if the exception event is generated by a division instruction by zero, because this instruction cannot be completed, it will retire abnormally but will not change the state of the CPU (that is, will not change the value of the register), the address of this division instruction will be saved in the exception address register, and the CPU will continue to execute this division instruction when it returns from the interrupt service routine.

The exception is handled as follows:

In the first step, the processor saves the PSR and PC into shadow registers (EPSR and EPC).

The second step is to set the supervisor mode setting bit S in the PSR to 1 (regardless of the operating mode the processor is in when the exception occurs), so that the processor enters supervisor mode.

The third step is to update the exception vector number VEC field in the PSR to the currently occurring exception vector number, identifying the exception category and the situation in which the shared exception service is supported.

The fourth step is to clear the exception enable bit EE in PSR to disable exception response. When EE is zero, any exception (except ordinary interrupt) will be treated as an unrecoverable error exception by the processor. When an unrecoverable error exception occurs, EPSR and EPC will also be updated.

The fifth step is to clear the interrupt enable bit IE in PSR to disable the interrupt response.

The second, third and fourth steps above occur simultaneously.

In the sixth step, the processor first calculates the exception entry address based on the exception vector number in the PSR, and then uses this address to obtain the exception service routine.

The address of the first instruction in the sequence. Multiply the exception vector by 4 and add it to the exception vector base address (stored in the vector base address register VBR,

When VBR does not exist, the value is always zero) to get the abnormal entry address, read a word from the memory with the abnormal entry address, and

[31:1] is transferred to the program counter as the address of the first instruction of the exception service routine (the lowest bit of the PC is always 0, which is the same as the exception vector

The lowest bit of the exception entry address value obtained in the table is irrelevant). For vectored interrupts, the exception vector is provided by the external interrupt controller; for other

The processor determines the exception vector based on internal logic.

In the final step, the processor starts executing from the first instruction of the exception service routine and transfers the control of the CPU to the exception service routine.

Start exception handling.

All exception vectors are stored in the address space that the supervisor is allowed to access. In the processor address map, only the restart vector is fixed.

Once the processor has been initialized, the VBR allows the base address of the exception vector table to be reloaded.

E804 supports a maximum vector table size of 1024 bytes, which means it supports 256 exception vectors (see [Table 4.1 for details](#)). The first 31 vectors are used

The 32nd vector is reserved for the vectors recognized inside the processor, and the remaining 224 vectors are reserved for external devices.

The interrupt vector and interrupt request enable the processor to respond to the interrupt service. The processor latches this interrupt vector when responding to the interrupt request.

Interrupt vector devices, the processor provides automatic vectors for general interrupts.

Table 4.1: Exception vector allocation

Vector Number	Vector Offset (Hexadecimal)	Vector Allocation
0	000	Restart abnormality.
1	004	Unaligned access exception.
2	008	Access error exception.
3	00C	Division by zero exception.
4	010	Illegal instruction exception.
5	014	Privilege violation exception.
6	018	Trapping exceptions.
7	01C	Breakpoint exception.
8	020	Unrecoverable error exception.
9	024	Idly abnormal.
10	028	Normal interrupt. (auto vectored)
11 - 15	02C - 03C	reserve.
16 - 19	040 - 04C	Trap instruction exceptions (TRAP #0-3).
20 - 21	050 - 054	reserve.
22	058	TSPEND Interrupt
23-29	05C - 074	reserve
30	078	Floating point exception
31	07C	reserve.
32 - 255	080 - 3FC	Reserved for use by the vectored interrupt controller.

4.2 Exception Types

This section describes external interrupt exceptions and exceptions generated inside the E804. The exceptions handled by the E804 are of the following types:

- Restart abnormality;
- Unaligned access exception;
- AccessErrorException;
- Division by zero exception;
- Illegal instruction exception;
- Privilege violation exceptions;
- Tracking anomalies;
- Breakpoint exceptions;
- Unrecoverable error exception;
- Idly abnormal;
- General interrupts;
- Trap instruction exception;
- TSPEND interrupt.

4.2.1 Restart exception (vector offset 0X0)

The restart exception has the highest priority among all exceptions. It is used for system initialization and system recovery after a major failure. Restarting will terminate the process.

All operations of the processor are suspended and cannot be resumed.

Restarting an exception sets PSR (S) to a high level to make the processor work in supervisor mode, and clears PSR (TM) to disable tracking exceptions. Restarting an exception will also clear PSR (IE) to disable interrupt response. At the same time, VBR (vector reference register) is also cleared, and the reference address of the exception vector table is 0X00000000. The CPU reads the exception service program entry address from the exception vector table with offset address 0X0 as the offset address and loads it into the program counter (PC).

4.2.2 Unaligned access exception (vector offset 0X4)

An unaligned access exception occurs when the processor attempts to perform an access operation on an address boundary that is inconsistent with the access size.

PSR (MM), this exception can be masked, the processor will ignore the alignment check on the data, and access the address boundary that is smaller than this unaligned address and closest to it. EPC points to the instruction that attempted the unaligned access.

Misaligned access exceptions occur only on data accesses.

4.2.3 Access Error Exception (Vector Offset 0X8)

If a bus access results in an incorrect reply, an access error exception occurs. An access error exception also occurs when an access error occurs in a protected memory area. The EPC points to the instruction that initiated the incorrect access.

4.2.4 Division by zero exception (vector offset 0x0C)

When the processor finds that the divisor of a division instruction is zero, the processor performs an exception process and does not execute the division instruction. EPC points to the division instruction. make.

4.2.5 Illegal instruction exception (vector offset 0X10)

If the processor finds an illegal instruction or an unimplemented instruction during decoding, the instruction will not be executed and exception processing will be performed.

This is an illegal instruction.

4.2.6 Privilege Violation Exception (Vector Offset 0X14)

To protect system security, some instructions are granted privileges and can only be executed in supervisor mode. Attempting to execute the following privileged instructions in user mode will generate a privilege violation exception: MFCR, MTCR, PSRSET, PSRCLR, RTE, STOP, WAIT, DOZE.

If the processor finds a privilege violation exception, it will handle the exception before executing the instruction. The EPC points to the privileged instruction.

4.2.7 Tracing Exception (Vector Offset 0X18)

To facilitate program development and debugging, the E804 tracks each instruction or instruction that changes the control flow. In instruction tracking mode, each instruction will generate a tracking exception after execution, so that the debugger can monitor the execution of the program. In jump tracking mode, each instruction that changes the control flow (including POP) will generate a tracking exception. For conditional jump instructions, the tracking exception will occur regardless of whether the program jumps or not.

The TM bit in the PSR controls the trace mode. The state of the TM determines whether a trace exception is generated when the instruction retires.

Definition of TM bit.

Trace exception processing starts after the traced instruction retires and before the next instruction retires. The EPC points to the next instruction.

The following control-related instructions cannot be traced: RTE, TRAP, STOP, WAIT, DOZE, and BKPT.

When enabled, trace exceptions are handled as part of the normal execution of the RTE, regardless of the state of the TM bit in the PSR or EPSR.

If the instruction being traced is not completed due to other exceptions, the trace exception will not be processed. If the processor finds that there is an interrupt waiting to be processed when the traced instruction retires, the interrupt has a higher priority than the trace exception, and the TP (trace exception waiting to be processed) of the shadow register of PSR will be valid, and the processor responds to the interrupt. When the RTE of the interrupt service routine retires, the trace exception waiting to be processed will be processed. The trace exception waiting to be processed is used to track the previous instruction. When responding to the interrupt, this instruction has been retired. In order to avoid the situation where the trace exception is lost after the interrupt is processed, the trace exception needs to be processed immediately after the interrupt is processed. In this case, the trace exception priority is the highest and second only to restart.

4.2.8 Breakpoint exception (vector offset 0X1C)

E804 provides a breakpoint instruction BKPT, which generates a breakpoint exception when it is retired. When a breakpoint exception occurs, EPC points to the instruction.

4.2.9 Unrecoverable Error Exception (Vector Offset 0X20)

When PSR (EE) is zero, an exception will generate an unrecoverable exception because the information used for exception recovery (stored in EPC and EPSR) Overwritten due to an unrecoverable error.

Since the software is written to exclude the possibility of abnormal events by default when PSR (EE) is zero, if the CPU has an exception in this case, this error generally means a system error. In the service routine of the unrecoverable error exception, the exception type that causes the unrecoverable error exception is uncertain.

4.2.10 IDLY exception (exception offset 0X24)

The IDLY exception is used to indicate that a transmission error has occurred in the IDLY instruction sequence. In the exception service routine, the EPC points to the instruction that caused the transmission error. The exception service routine should analyze the cause of the transmission error and back up the value of the EPC so that the IDLY instruction sequence can be re-executed if necessary.

4.2.11 Trap instruction exception (vector offset 0X40 - 0X4C)

Some instructions can be used to explicitly generate a trap exception. The TRAP#n instruction can force an exception to occur. It is used by the user program to generate a trap exception. In the exception service routine, EPC points to the TRAP instruction.

4.2.12 TSPEND interrupt (vector 0X58)

When configuring the vector interrupt controller, set the VIC_TSPEND register through software to generate a TSPEND interrupt. For details, please refer to [Tightly Coupled IP](#).

4.2.13 Floating point exception (vector offset 0X78)

Floating-point exceptions may occur during the operation of floating-point instructions.

4.3 Interrupt Exception

When an external device needs to request a service from the processor or send information required by the processor, it can use an interrupt request signal and a corresponding interrupt vector signal to request an interrupt exception from the processor.

E804 provides a tightly coupled vector interrupt controller. Users can choose to integrate the system by adding an E804 CPU to the vector interrupt controller. In this case, they only need to connect the external IP interrupt source signal to the vector interrupt controller. Users can also select the CPU core and integrate it in the system.

The E804 CPU core provides two interrupt request signals, supporting automatic interrupt vector number provision and interrupt vector number provision explicitly by peripherals.

Figure 4.1 shows the interface signals of the processor core related to interrupts. When VIC is not configured, in order to support vectored interrupts, the peripheral uses an 8-bit interrupt vector signal to provide the interrupt vector number when requesting an interrupt, or sets pad/intc_cpu_avec_b to use the automatic interrupt vector number. If PSR (IE) or PSR (EE) is cleared, the pad/intc_cpu_int_b input signal is masked, and the processor does not respond to the interrupt. When PSR (IE) or PSR (EE) is valid at the same time, the CPU responds to the interrupt. At this time, if pad/intc_cpu_avec_b is valid, the processor uses the automatic vector number, and its vector offset is 0X28, otherwise the processor uses the vector number provided on pad/intc_cpu_vec_b[7:0]. cpu_intc/pad_int_ack indicates that the CPU has responded to the interrupt. pad/intc_cpu_int_b and pad/intc_cpu_avec_b are both valid at low level, and pad/intc_int_ack is valid at high level. When VIC is configured, the corresponding interrupt signals are generated by the control signals, and the processing and response of these signals are consistent with the above description.

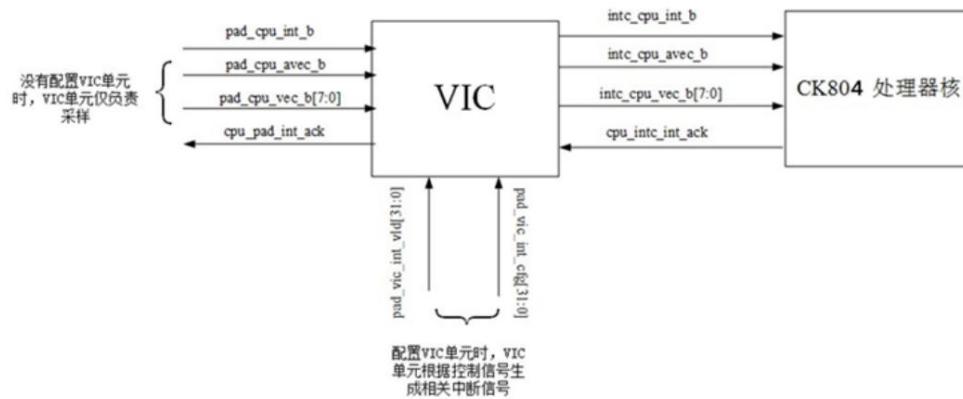


Figure 4.1: Interrupt interface signals

In Figure 4.2 , when the interrupt vector number is ready, the pad_cpu_int_b interrupt signal line is pulled low.

- If the VIC unit is not configured, sys_clk pulls intc_cpu_int_b low after sampling ($T1=1\text{cyc}$) to send an interrupt request to the CPU;
- If VIC is configured, cpu_clk needs to be used for two cycles to sample and determine the priority ($T1=2\text{cyc}$) before sending

The CPU sends an interrupt request.

After the signal is sampled by the rising edge of the CPU internal clock cpu_clk, the CPU receives an interrupt and obtains the interrupt vector according to the vector signal. Under the condition that the external system can respond within one cycle, after responding to the interrupt, the CPU needs five cycles ($T2=5\text{cyc}$) to issue an instruction fetch request for the first instruction of the interrupt service program and enter the interrupt service program. In the interrupt service program, the software should clear the external interrupt source, that is, pull up the interrupt valid signal. This signal also needs to be sampled by the CPU and the external clocks in sequence before exiting the interrupt.

E804 can be configured with interrupt acceleration function. After the CPU responds to the interrupt, it starts speculative execution of NIE and IPUSH instructions while taking the exception entry address. If the interrupt service program also executes these two instructions first, it can save six cycles of time under the condition that the external system can respond within one cycle. If a speculative prediction error, access exception or debug request occurs, the interrupt acceleration function is terminated.

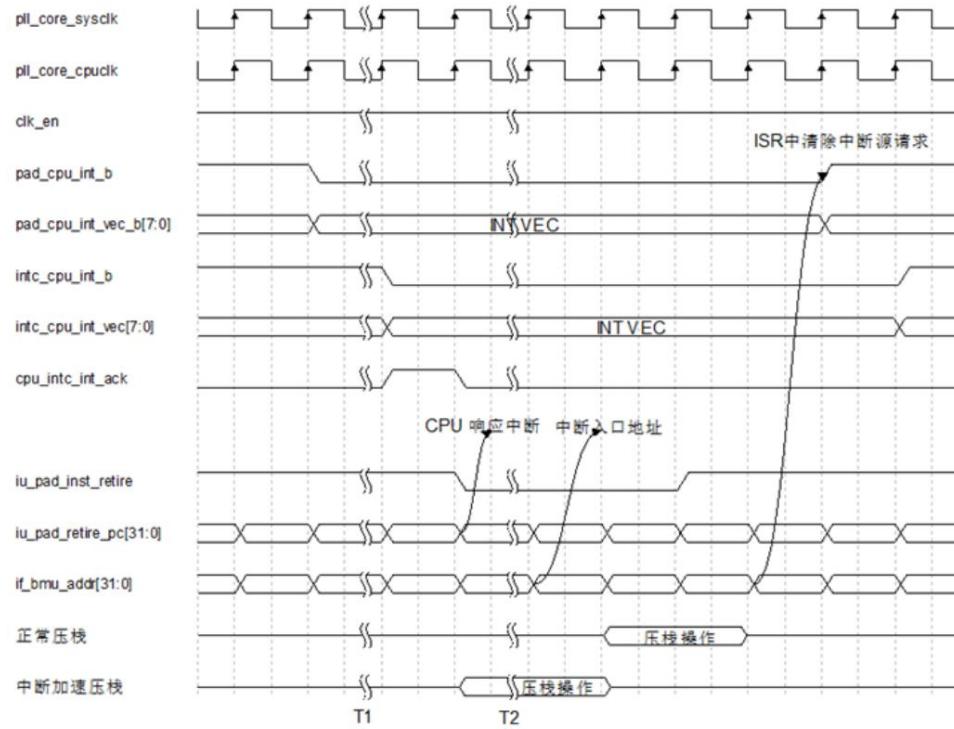


Figure 4.2: Processor interrupt response timing diagram

If the corresponding interrupt controller is configured, multiple interrupt sources are supported, and their corresponding interrupt priorities can be set separately and the interrupt nesting function can be realized.

For more detailed interrupt mechanism and interface signal description, please refer to [Tightly Coupled IP](#).

4.4 Exception Priority

As shown in [Table 4.2](#), E804 divides the priority into 10 levels according to the characteristics of the exceptions and the order in which they are processed. In [Table 4.2](#),

1 represents the highest priority and 10 represents the lowest priority. It is worth noting that in Group 9, several exceptions share the same priority because they are mutually exclusive.

In E804, multiple exceptions can occur simultaneously. The restart exception is very special and has the highest priority. All other exceptions are handled according to the priority relationship in [Table 4.2](#).

If multiple exceptions occur at the same time, the exception with the highest priority is handled first. After the exception is returned, the processor re-executes the exception

When the instruction is executed, the remaining exceptions can be reproduced in sequence.

Table 4.2: Exception priorities

Priority	Exceptions with their associated	feature
1	priorities Restart	The processor stops all program execution and initializes the system.
2	exceptions Pending trace exceptions	If TP=1 in EPSR, after RTE instruction is retired, the processing Handler handles pending tracking exceptions.
3	IDLY Error	After the associated instruction retires, the processor saves the context and handles the exception. often.
4	Misalignment Error	After the associated instruction retires, the processor saves the context and handles the exception. often.
6	Normal interrupt	If IC=0, the interrupt is responded after the instruction retires; If IC = 1, the processor enables interrupts before the instruction is completed. response.
7.0	Unrecoverable error exception	After the associated instruction retires, the processor saves the context and handles the exception. often.
7.1	Access Error	
8	Illegal instruction Privileged Exceptions Division by zero Trap Instructions Breakpoint instructions Floating point exceptions	After the associated instruction retires, the processor saves the context and handles the exception. often.
9	Tracking Exceptions	After the associated instruction retires, the processor saves the context and handles the exception. often.

4.5 Exception Return

The processor returns from the exception service routine by executing the RTE instruction. The RTE instruction uses the registers stored in the EPSR and EPC shadow registers.

The context is returned from the exception service routine.

Chapter 5 Tightly Coupling IP

5.1 Introduction to Tightly Coupled IP

In order to improve the system integration of E804 and facilitate user integration and use, E804 implements a series of system key functions that are closely related to the processor. IP, these IP are collectively called tightly coupled IP (TCIP). The tightly coupled IP of E804 includes system timer CoreTim, Vector interrupt controller VIC, on-chip cache control register unit CRU. These tightly coupled IPs are matched with E804, plus a small amount of external memory, etc. resources, a minimum-function SoC system can be formed, which improves the convenience of using E804 and reduces the development and application of E804. cost.

The main functions of the E804's tightly coupled IP are shown in [Table 5.1](#), and the system structure diagram is shown in [Figure 5.1](#).

Table 5.1: Tightly coupled IP main functions

IP Name	Key Features
System Timer	Complete the timing function of the system and wake up the CPU when the power consumption is low.
Vectorized Interrupt	Complete interrupt collection, arbitration, hardware nesting, and interaction with the processor.
Controller On-chip Cache Control Register Unit	Sets the E804 on-chip cache, such as switching the cache, configuring the cacheable area, etc.

Unlike traditional IP, tightly coupled IP is connected to the processor through a dedicated tightly coupled IP bus interface, without the need to access it through the system bus. The tightly coupled IP bus interface is directly connected to the E804 Bus Matrix Unit (BMU), supporting a single CPU. The clock cycle of tightly coupled IP access transmission not only improves the access efficiency of tightly coupled IP, but also improves the efficiency of system integration. Other system IPs share a unified memory address space and perform register access and functions through transfer instructions (Load) and storage instructions (Store) Control. The memory address allocation of the tightly coupled IP is shown in [Table 5.2](#).

Table 5.2: Memory address allocation for tightly coupled IP

IP name	Memory address space
System timer	0xE000E010~0xE000E0FF
vector interrupt controller	0xE000E100~0xE000ECFF
On-chip cache control register unit	0xE000F000~0xE000FFFF

In addition to communicating with the core through a dedicated bus interface, the tightly coupled IP also functions directly with the processor. The vector interrupt controller transmits the interrupt information after arbitration to the processor and receives the interrupt response signal from the processor; the on-chip high-speed cache is located between the core bus interconnect unit (BMU) and the instruction bus interface unit (I-AHBL).

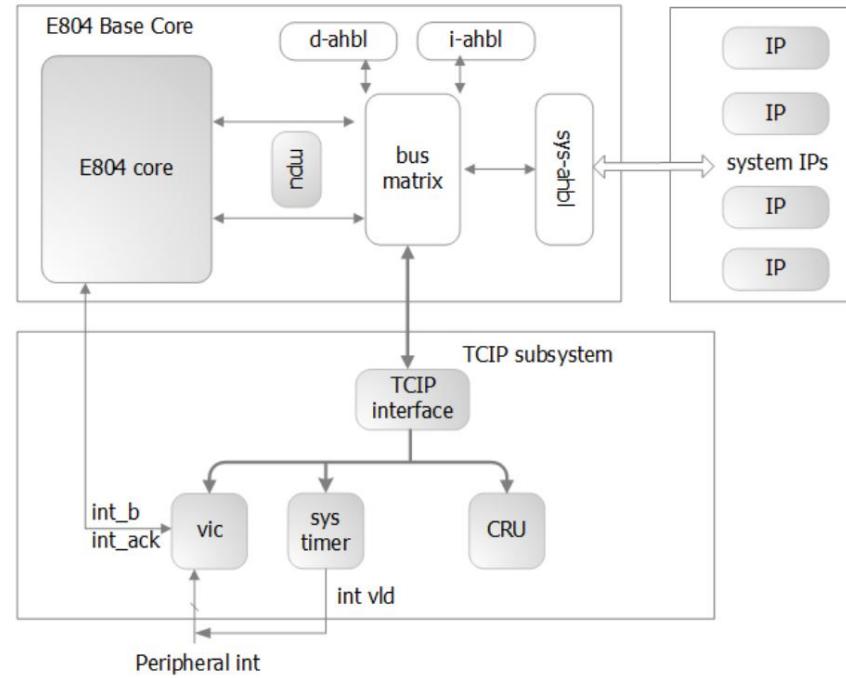


Figure 5.1: System structure diagram of tightly coupled IP

All control registers of the tightly coupled IP are 32-bit registers, so they can only be accessed and transmitted in word units. Any access in half-word or byte units will cause unexpected errors, and the tightly coupled IP registers only support the little-endian format. All accesses to the tightly coupled IP registers must be performed in the supervisor mode. Access in the normal user mode will generate an access error exception.

5.2 System Timer

5.2.1 Introduction

The system timer is an optional module of E804, which is mainly used for timing. The system timer provides a simple and easy-to-use 24-bit circular decrement counter. When the system timer is enabled, the counter starts working. When the counter decrements to 0, it will initiate an interrupt request to the vector interrupt controller to request a processor response and handle the system timer's affairs. The structural block diagram of the system timer is shown in Figure 5.2.

5.2.2 Register Definition

Each register of the system timer is 32 bits wide, and the register address space is shown in Table 5.3 .

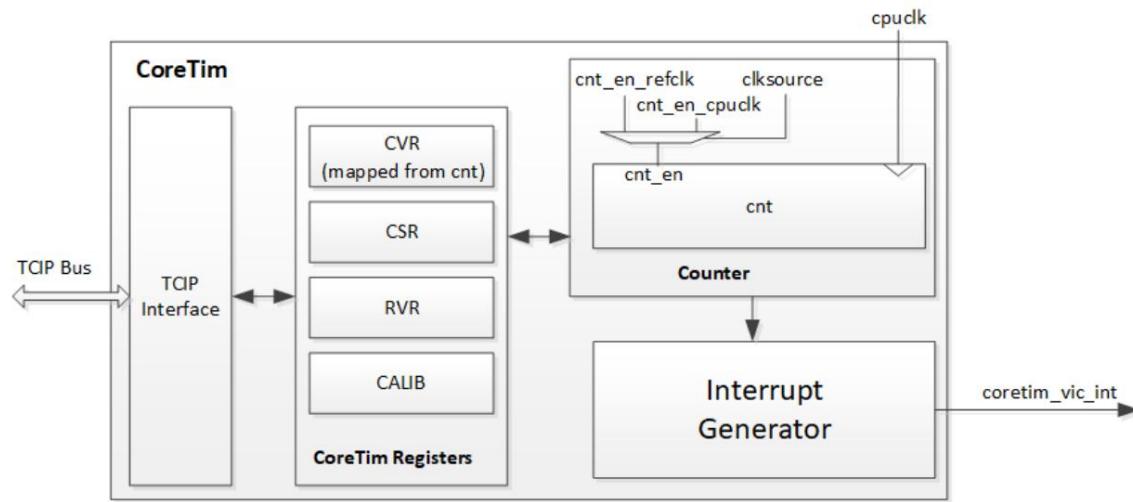


Figure 5.2: System timer structure diagram

Table 5.3: System timer register definition

address	Name	Type	Initial Value	Description		
0xE000E010	CORET_CSR	Read/write	0x00000004	Control status register.		
0xE000E014	CORET_RVR	Read/Write	-	Backfill Value Register.		
0xE000E018	CORET_CVR	Read/Write	-	Current Value Register.		
0xE000E01C	CORET_CALIB	Read only	-	Calibration register.		
0xE000E020~	-	-	-	-	reserve.	
0xE000E0FF						

5.2.2.1 Control and Status Register (CORET_CSR)

CORET_CSR is the control and status register of the system timer as shown in Figure 5.3. The bit description of the CORET_CSR register is shown in Table 5.4. As shown in 5.4 .

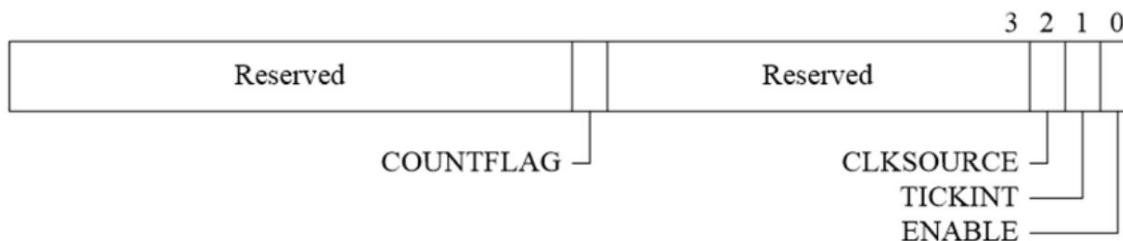


Figure 5.3: System Timer Control and Status Registers

Table 5.4: System Timer Control and Status Register Field Description

Bit Type Name			describe
31:17	-	-	reserve.
16	Read-only COUNTFLAG	Indicates whether the counter has counted to 0 since the last read of this register:	<p>0 The counter has not yet counted to 0;</p> <p>1 The counter has counted to 0.</p> <p>When the counter value changes from 1 to 0, COUNTFLAG will be set.</p> <p>Reading the CSR register and any write to the CVR register will cause COUNTFLAG Cleared.</p>
15:3	-	-	reserve.
2	Read/write CLKSOURCE	Indicates the clock source of the system timer:	<p>0 Use the optional external reference clock as the clock source of the counter;</p> <p>1 Use the internal clock as the clock source of the counter;</p> <p>If no external clock is present, reading this bit will return 1 and writing this bit will have no effect.</p> <p>The external reference clock frequency must be less than or equal to half the internal clock frequency.</p>
1	Read/Write TICKINT	Indicates whether the interrupt status bit of the system timer will be changed when the count reaches 0:	<p>0 When counting to 0, it will not affect the system timer interrupt status bit;</p> <p>1 Counting to 0 changes the interrupt status bit of the system timer.</p> <p>Writing to the CVR register will clear the counter, but this method will not cause the system to The interrupt status bit of the timer changes.</p>
0	Read/write ENABLE	Indicates the enable status bit of the system timer:	<p>0 The counter is not enabled;</p> <p>1 Counter enabled.</p>

5.2.2.2 Backfill Value Register (CORET_RVR)

The CORET_RVR register is used to assign a value to the CORET_CVR register at the beginning of each counting cycle.

The registers and their bit descriptions are shown in Figure 5.4 and Table 5.5 .



Figure 5.4: System timer backfill value register

Table 5.5: System Timer Backfill Value Register Field Description

Bit	name	describe
31:24	-	reserve.
23:0	RELOAD	When the counter reaches 0, the RELOAD value will be assigned to the CORET_CVR register. Writing 0 to the CORET_RVR register will stop the counter at the next cycle. After that, the counter value will remain at 0. When the counter is enabled using an external reference clock, Wait until the counter starts counting normally (that is, when CORET_CVR becomes a non-zero value), then Set CORET_RVR to 0 to stop the counter at the next cycle. The counter cannot start the first count.

How to calculate RELOAD value:

The normal value range of RELOAD is between 0x1 and 0x00FFFFFF. RELOAD can be assigned a value of 0, but it has no effect.

The system timer interrupt and COUNTFLAG bit are only effective when the count value changes from 1 to 0.

For a timer with N count clock cycles, the value of RELOAD needs to be assigned to N-1. For example, if you want to generate a CoreTim interrupt, need to assign 99 to RELOAD.

5.2.2.3 Current Value Register (CORET_CVR)

CORET_CVR contains the current value of the system timer. The CORET_CVR register and its bit description are shown in Figure 5.5 and Table 5.6 .



Figure 5.5: System timer current value register

Table 5.6: System timer current value register field description

Bit	name	describe
31:24	-	reserve.
23:0	CURRENT	This indicates the value of the counter when it is read. Writing to the CORET_CVR register will cause both this register and the COUNTFLAG status to be set. If the bit is cleared, it causes the system timer to fetch the register at the beginning of the next clock cycle. The value in CORET_RVR is assigned to CORET_CVR. Note that writing to CORET_CVR does not This will cause the interrupt status bit of the system timer to change. Reading CORET_CVR returns the value of the counter at the time the register was accessed.

5.2.2.4 Calibration Register (CORET_CALIB)

The CORET_CALIB register describes the calibration function of the system timer. Its reset value is implementation dependent: it needs to be obtained from the device provider.

The provided documentation provides information about the meaning of the CORET_CALIB bits and the calibration value TENMS in the CORET_CALIB register.

With the calibration value of TENMS, the software can multiply this value by a certain ratio to obtain other different counting cycles.

Each counting cycle must be within the value range of the counter; the CORET_CALIB register and its bit description are shown in [Figure 5.6](#) and [Table 5.7](#).

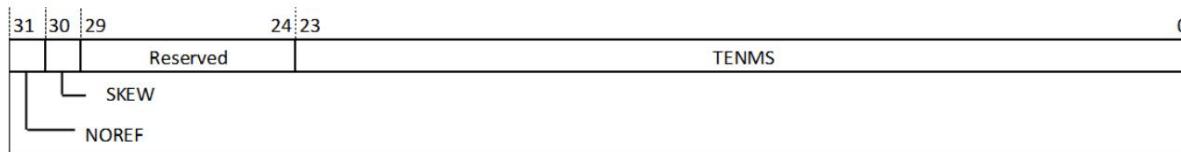


Figure 5.6: System Timer Calibration Registers

Table 5.7: System Timer Calibration Register Field Descriptions

Bit	The	describe
31	noref	whether the device implements an external reference clock: 0 The device has an external reference clock; 1 Device has no external reference clock. When this bit is 1, the CLKSOURCE bit in the CORET_CSR register is fixed to 1. Can be rewritten.
30	SKEW	Indicates whether the 10ms calibration value is accurate: 0 10ms The calibration value is accurate; 1 The 10ms calibration value has errors due to clock frequency issues.
29:24	-	reserve.
23:0	TENMS	TENMS is used to indicate the backfill value corresponding to 10ms. Depending on the specific value of SKEW, it may be Indicates the exact 10ms value or the value closest to 10ms. If the value of this field is 0, it means that the calibration value is unknown. This may be because the reference clock is an unknown The input is unknown or dynamically changing.

Note: If CORET_RVR is set to 0, the CoreTim counter will stop working in the next round, regardless of the counter usage.

Energy status.

The value of the SYST_CVR register is unknown at reset. Before enabling the CoreTime counter, software must first set the required count value to Write a value to the CORET_RVR register, and then write any value to the CORET_CVR register. The latter operation will cause the value of CORET_CVR to After the counter is enabled, the counter can read the value in the CORET_RVR backfill register and count down from this value.

This avoids starting the count from an arbitrary value.

5.2.3 Operation steps

Since the two registers CORET_RVR and CORET_CVR in the system timer have no reset values, before the system timer works, The following steps must be followed:

1. Write the required backfill value to the CORET_RVR register;
2. Write any value to the CORET_CVR register to clear it to zero;

3. Operate the CORET_CSR register to enable the system timer.

5.3 Vectored Interrupt Controller

5.3.1 Introduction

The vector interrupt controller (VIC) is an IP unit tightly coupled with the E804 for efficient interrupt processing. The vector interrupt controller can support up to 128 interrupt sources (IRQ[127:0]), each of which has an independent software-programmable interrupt priority. The vector interrupt controller collects interrupt requests from different interrupt sources and arbitrates the interrupt requests based on the interrupt priority. The highest priority interrupt will obtain interrupt control and send an interrupt request to the processor. When the processor responds to the interrupt request, the processor returns an interrupt request response signal to the VIC; when the processor exits the interrupt service routine (ISR), the processor returns an interrupt exit signal to the VIC.

The vector interrupt controller supports interrupt nesting. When the processor is processing an interrupt request and a higher priority interrupt request comes, the processor will interrupt the current interrupt service routine and respond to the higher priority interrupt request. When the higher priority interrupt request is processed, the CPU returns to the interrupted interrupt service routine to continue execution. The vector interrupt controller allows high-priority interrupt requests to preempt low-priority interrupt requests, but does not allow interrupts of the same level or lower priority to preempt, ensuring the real-time nature of interrupt response.

The system structure diagram of the vector interrupt controller is shown in Figure 5.7.

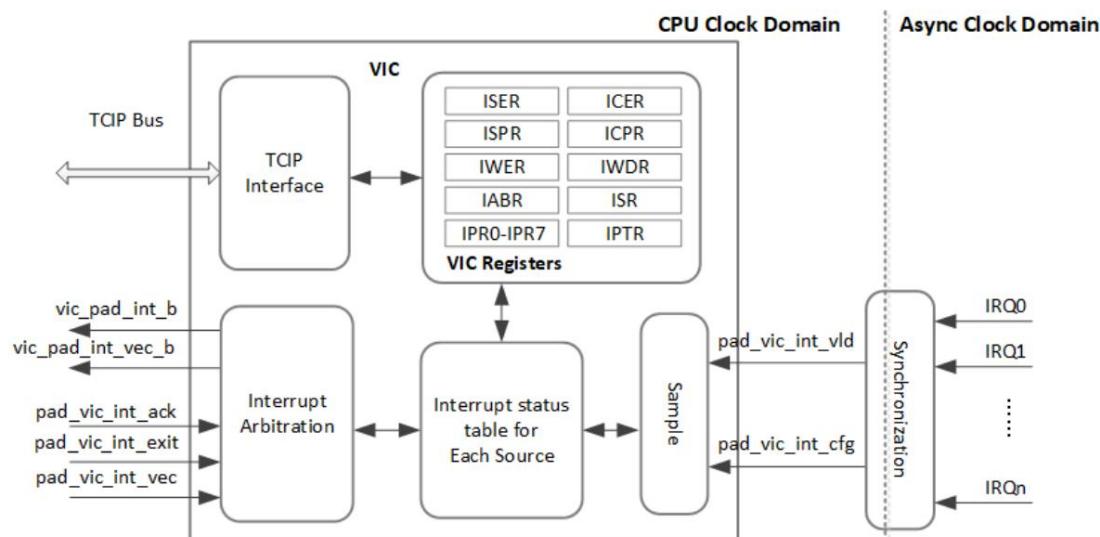


Figure 5.7: Vectored interrupt controller system block diagram

The vectored interrupt controller supports the following features:

- The number of interrupts is hardware configurable, supporting 4, 8, 16, 24, 32, 64, 96 and 128;
- The software configures the priority for each interrupt through an 8-bit register. In E804, if the number of interrupt sources is less than or equal to 32, only the most significant 2 bits of the 8 bits are used to represent the priority, and the remaining bits are always kept as 0; if the number of interrupt sources is 64, the most significant 3 bits of the 8 bits can be used to represent 8 priorities; if the number of interrupt sources is 96 or 128, the most significant 4 bits of the 8 bits can be used to represent 16 priorities. Priorities are arranged from low to high, with priority level 0 being the highest priority and priority level P (P=4, 8, 16) being the lowest priority;

- Supports both level and pulse interrupt source signals;
- The interruption supports dynamic adjustment of priority during processing, and interruption is realized with a small hardware cost by setting the priority threshold register Inversion of priorities;
- Supports interrupt nesting, allowing higher priority interrupts to preempt while the CPU is executing the interrupt service routine.

5.3.2 Register Definition

VIC provides a set of 32-bit registers, and the address space of each register is shown in Table 5.8 .

Table 5.8: Vectored Interrupt Controller Register Definitions

address	Name	Type	Initial Value	Description
0xE000E100	VIC_ISER0	Read/write	0x00000000	Interrupt enable setting register (<i>Interruptions 0-31</i>).
0xE000E104	VIC_ISER1	Read/write	0x00000000	Interrupt enable setting register (<i>Interruptions 32-63</i>).
0xE000E108	VIC_ISER2	Read/write	0x00000000	Interrupt enable setting register (<i>Interruption No. 64-95</i>).
0xE000E10C	VIC_ISER3	Read/write	0x00000000	Interrupt enable setting register (<i>Interruption No. 96-127</i>).
0xE000E110~0xE000E13F -		-	-	reserve.
0xE000E140	VIC_IWERO	Read/write	0x00000000	Low power wake-up setting register (<i>Interruptions 0-31</i>).
0xE000E144	VIC_IWERO1	Read/write	0x00000000	Low power wake-up setting register (<i>Interruptions 32-63</i>).
0xE000E148	VIC_IWERO2	Read/write	0x00000000	Low power wake-up setting register (<i>Interruption No. 64-95</i>).
0xE000E14C	VIC_IWERO3	Read/write	0x00000000	Low power wake-up setting register (<i>Interruption No. 96-127</i>).
0xE000E150~0xE000E17F -		-	-	reserve.

Continue on next page

Table 5.8 – Continued from previous page

0xE000E180	VIC_ICER0 Read/write 0x00000000 Interrupt enable			clear register (<i>Interrupts 0-31</i>).
0xE000E184	VIC_ICER1 Read/write 0x00000000 Interrupt enable			clear register (<i>Interruptions 32-63</i>).
0xE000E188	VIC_ICER2 Read/write 0x00000000 Interrupt enable			clear register (<i>Interrupt No. 64-95</i>).
0xE000E18C	VIC_ICER3 Read/write 0x00000000 Interrupt enable			clear register (<i>Interrupt No. 96-127</i>).
0xE000E190~0xE000E1BF -	-	-	-	reserve.
0xE000E1C0	VIC_IWDR0 Read/write 0x00000000 Low power wake-up			clear register (<i>Interrupts 0-31</i>).
0xE000E1C4	VIC_IWDR1 Read/write 0x00000000 Low power wake-up			clear register (<i>Interruptions 32-63</i>).
0xE000E1C8	VIC_IWDR2 Read/write 0x00000000 Low power wake-up			clear register (<i>Interrupt No. 64-95</i>).
0xE000E1CC	VIC_IWDR3 Read/write 0x00000000 Low power wake-up			clear register (<i>Interrupt No. 96-127</i>).
0xE000E1D0~0xE000E1FF -	-	-	-	reserve.
0xE000E200	VIC_ISPR0 Read/write 0x00000000 Interrupt wait setting			register (<i>Interrupts 0-31</i>).
0xE000E204	VIC_ISPR1 Read/write 0x00000000 Interrupt wait setting			register. (<i>Interruptions 32-63</i>).
0xE000E208	VIC_ISPR2 Read/write 0x00000000 Interrupt wait setting			register (<i>Interrupt No. 64-95</i>).
0xE000E20C	VIC_ISPR3 Read/write 0x00000000 Interrupt wait setting			register (<i>Interrupt No. 96-127</i>).
0xE000E210~0xE000E27F -	-	-	-	reserve.
0xE000E280	VIC_ICPR0 Read/write 0x00000000 Interrupt wait clear			register (<i>Interrupts 0-31</i>).
0xE000E284	VIC_ICPR1 Read/write 0x00000000 Interrupt wait clear			register (<i>Interruptions 32-63</i>).
0xE000E288	VIC_ICPR2 Read/write 0x00000000 Interrupt wait clear			register (<i>Interrupt No. 64-95</i>).
0xE000E28C	VIC_ICPR3 Read/write 0x00000000 Interrupt wait clear			register (<i>Interrupt No. 96-127</i>).
0xE000E290~0xE000E2FF -	-	-	-	reserve.
0xE000E300	VIC_IABR0 Read/write 0x00000000 Interrupt response			status register (<i>Interrupts 0-31</i>).

Continue on next page

Table 5.8 – Continued from previous page

0xE000E304	VIC_IABR1 Read/write 0x00000000	Interrupt response status register (Interruptions 32-63).
0xE000E308	VIC_IABR2 Read/write 0x00000000	Interrupt response status register (Interruption No. 64-95).
0xE000E30C	VIC_IABR3 Read/write 0x00000000	Interrupt response status register (Interruption No. 96-127).
0xE000E310~0xE000E3FF -	-	-
0xE000E400~0xE000E47C VIC_IPR0- VIC_IPR31	Read/write 0x00000000	Interrupt priority setting register.
0xE000E480~0xE000EBFF -	-	-
0xE000EC00	VIC_ISR Read Only 0x00000000	Interrupt Status Register.
0xE000EC04	VIC_IPTR Read/write 0x00000000	Interrupt priority threshold register.
0xE000EC08	VIC_TSPEND Read/write 0x00000000	Tspending enable setting register.
0xE000EC0C	VIC_TSABR Read/write 0x00000000	Tspending response status register.
0xE000EC10	VIC_TSPPR Read/write 0x00000000	Tspending Wait setup register.
0xE000EC14~0xE000ECFF -	-	-
		reserve.

5.3.2.1 Interrupt Enable Setting Register (VIC_ISER)

VIC_ISER is used to enable each interrupt and feedback the enable status of each interrupt. Figure 5.8 describes the bit distribution of VIC_ISER.

5.9 describes the bit definition of VIC_ISER.



Figure 5.8: Interrupt enable setting register

Table 5.9: Interrupt enable setting register field definition

Bit	Name	describe
31:0	SETENA	Sets the usage and reads the enable status of one or more interrupts. Each bit corresponds to the same number. Interrupt sources: Read Operation: 0 corresponds to interrupt not being enabled; 1 The corresponding interrupt is enabled. Write Operation: 0 is invalid; 1 enables the corresponding interrupt.

If a pending interrupt is enabled, the vectored interrupt controller activates the interrupt according to its priority.

Yes, the vectored interrupt controller will not activate the interrupt even if it is in the waiting state.

5.3.2.2 Interrupt Low Power Wake-up Setting Register (VIC_IWER)

VIC_IWER is used to enable the low-power wake-up function of each interrupt and feedback the enable status of each interrupt low-power wake-up.

Figure 5.9 describes the bit distribution of VIC_IWER, and Table 5.10 describes the bit definition of VIC_IWER.



Figure 5.9: Interrupt low power wake-up setting register

Table 5.10: Low power wake-up enable setting register field definition

Bit	Name	describe
31:0	SETENA	<p>Set to read the enable status of one or more interrupt low power wake-up. Each bit corresponds to interrupt sources with the same number: Read</p> <p>operation: 0: The low-power wake-up function of the corresponding interrupt is not enabled; 1: The low-power wake-up function of the corresponding interrupt is enabled. Write</p> <p>operation: 0: Invalid; 1: Enable the low-power wake-up function of the corresponding interrupt.</p>

If the low-power wake-up function of an interrupt is enabled and the interrupt is in the waiting state, the VIC generates a low-power wake-up request.

If the low-power wake-up function of the interrupt is not enabled, the VIC does not generate a low-power wake-up request even if the interrupt is in the waiting state.

Note: Interrupt enable and interrupt wake-up enable control interrupt transactions and interrupt wake-up functions respectively. When both are set, an interrupt in the waiting state generates both an interrupt request and a low-power wake-up request; when only one of them is enabled, only the corresponding function is activated; when neither is enabled, even if the interrupt is in the waiting state, no interrupt request or low-power wake-up request will be generated.

5.3.2.3 Interrupt Enable Clear Register (VIC_ICER)

VIC_ICER is used to clear the enable of each interrupt and feedback the enable status of each interrupt. Figure 5.10 describes the VIC_ICER

Bit distribution, Table 5.11 describes the bit definition of VIC_ICER.



Figure 5.10: Interrupt Enable Clear Register

Table 5.11: Interrupt Enable Clear Register Field Description

Bit	Name	describe
31:0	CLRENA	<p>Clear usage, read the enable status of one or more interrupts. Each bit corresponds to the same number</p> <p>Interrupt sources:</p> <p>Read Operation:</p> <p>0: The corresponding interrupt is not enabled; 1: The corresponding interrupt is enabled.</p> <p>Write Operation:</p> <p>0: invalid; 1: Clear the enable of the corresponding interrupt.</p>

5.3.2.4 Interrupt Low Power Wake-up Clear Register (VIC_IWDR)

VIC_IWDR is used to clear the low-power wake-up enable of each interrupt and feedback the enable status of each interrupt low-power wake-up. Figure 5.11 describes the bit distribution of VIC_IWDR, and Table 5.12 describes the bit definition of VIC_IWDR.



Figure 5.11: Interrupt low power wakeup clear register

Table 5.12: Interrupt low power wake-up clear register field description

Bit	Name	describe
31:0	CLRENA	<p>Clear usage, read the enable status of one or more interrupt low power wake-up. Each bit corresponds to the same number</p> <p>Interrupt sources with the same number:</p> <p>Read Operation:</p> <p>0: The low-power wake-up function of the corresponding interrupt is not enabled; 1: The low-power wake-up function of the corresponding interrupt is enabled.</p> <p>Write Operation:</p> <p>0: invalid; 1: Clear the low power wake-up function that enables the corresponding interrupt.</p>

5.3.2.5 Interrupt Wait Setting Register (VIC_ISPR)

VIC_ISPR represents setting each interrupt to the waiting state. The bit distribution of VIC_ISPR is described in Figure 5.12 .

Table 5.13 describes the bit definition of VIC_ISPR.



Figure 5.12: Interrupt wait setting register

Table 5.13: Interrupt wait setting register field description

Bit	Name Description
31:0	<p>SETPEND changes one or more interrupts to the waiting state. Each bit corresponds to the same number of interrupt sources:</p> <p>Read Operation:</p> <ul style="list-style-type: none"> 0: The corresponding interrupt is not in the waiting state. 1: The corresponding interrupt is in the waiting state. <p>Write Operation:</p> <ul style="list-style-type: none"> 0: Invalid. 1: Change the corresponding interrupt to the waiting state.

5.3.2.6 Interrupt Pending Clear Register (VIC_ICPR)

VIC_ICPR represents the wait state for clearing each interrupt. Figure 5.13 describes the bit distribution of VIC_ICPR, and Table 5.14 describes Bit definition of VIC_ICPR.



Figure 5.13: Interrupt wait clear register

Table 5.14: Interrupt wait clear register

Bit	Name Description
31:0	<p>CLRPEND clears the wait state of one or more interrupts. Each bit corresponds to the interrupt source with the same number:</p> <p>Read Operation:</p> <ul style="list-style-type: none"> 0: The corresponding interrupt is in the non-waiting state; 1: The corresponding interrupt is in the waiting state. <p>Write Operation:</p> <ul style="list-style-type: none"> 0: invalid; 1: Clear the wait state of the corresponding interrupt.

5.3.2.7 Interrupt Response Status Register (VIC_IABR)

VIC_IABR is used to indicate the current Active state of each interrupt. It is a register for software to query. In addition, the software can initialize When VIC is enabled, clear the Active status of all interrupts to 0. Figure 5.14 describes the bit distribution of VIC_IABR, and Table 5.15 describes the bit distribution of VIC_IABR. The bit definition of .



Figure 5.14: Interrupt Response Status Register

Table 5.15: Interrupt Response Status Register Field Description

Bit	name	describe
31:0	Active	<p>Query bit, indicating whether the interrupt source has been responded to by the CPU but has not been processed. Each bit corresponds to interrupt sources with the same number.</p> <p>Read Operation:</p> <p>0: No response from CPU; 1: Has been responded to by the CPU but has not been processed yet.</p> <p>Write Operation:</p> <p>0: Clear the Active state of the interrupt (the software cannot write 1 to this register, otherwise it will cause an inoperable interrupt. expected error).</p>

5.3.2.8 Interrupt Priority Setting Registers (VIC_IPR0 - VIC_IPR31)

Each interrupt priority setting register provides four interrupt source priority settings. Each interrupt source setting area corresponds to

For hardware support of 128 interrupt sources, registers from IPR0 to IPR31 are shown in Figure 5.15 and Figure 5.16 describes the interrupt sources.

Describes the interrupt priority setting register.

The vector interrupt controller selects the interrupt priority according to the priority number. The smaller the priority number, the higher the priority. If the priority numbers are the same, according to

The interrupt source number determines the priority order, the smaller the number, the higher the priority.

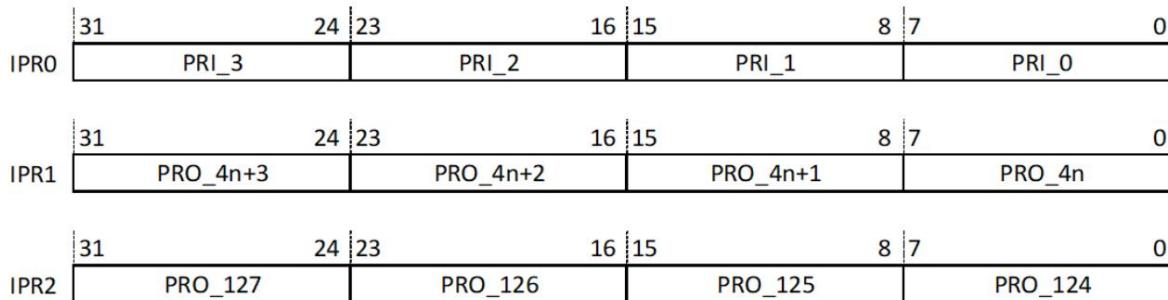


Figure 5.15: Overall layout of interrupt priority setting registers

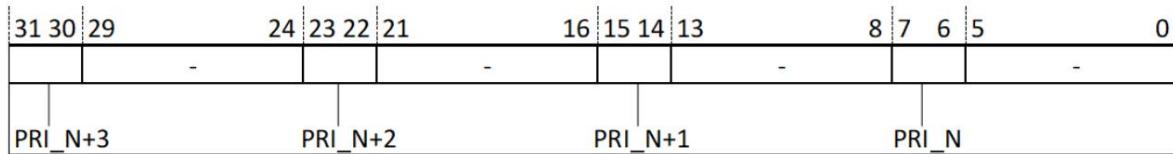


Figure 5.16: Interrupt priority setting register

Table 5.16 and Table 5.17 describe the bit definition of VIC_IPRn. In this table, N = 4n, where n is the VIC_IPRn register number.

For example, in VIC_IPR2, if n is 2, then N is 8.

Table 5.16: Interrupt priority setting register field description

Priority level 2	Bit	name	describe
(The number of interrupts is less than or equal to 32)	31:30	PRI_N+3	The priority of the interrupt number is N+3. The smaller the value, the higher the priority. The higher.
	29:24	-	reserve.
	23:22	PRI_N+2	The priority of the interrupt number is N+2. The smaller the value, the higher the priority. The higher.
	21:16	-	reserve.
	15:14	PRI_N+1	The priority of the interrupt number is N+1. The smaller the value, the higher the priority. The higher.
	13:8	-	reserve.
	7:6	PRI_N	The priority of the interrupt number is N. The smaller the value, the higher the priority. high.
	5:0	-	reserve.
(The number of interrupts is 64 individual)	31:29	PRI_N+3	The priority of the interrupt number is N+3. The smaller the value, the higher the priority. The higher.
	28:24	-	reserve.
	23:21	PRI_N+2	The priority of the interrupt number is N+2. The smaller the value, the higher the priority. The higher.
	20:16	-	reserve.
	15:13	PRI_N+1	The priority of the interrupt number is N+1. The smaller the value, the higher the priority. The higher.
	12:8	-	reserve.
	7:5	PRI_N	The priority of the interrupt number is N. The smaller the value, the higher the priority. high.
	4:0	-	reserve.

Table 5.17: Interrupt priority setting register field description

Priority level 4	Bit	name	describe
bits 16 priority levels (Number of interrupts is 96 or 128)	31:28	PRI_N+3	The priority of the interrupt number is N+3. The smaller the value, the higher the priority. The higher.
	27:24	-	reserve.
	23:20	PRI_N+2	The priority of the interrupt number is N+2. The smaller the value, the higher the priority. The higher.
	19:16	-	reserve.
	15:12	PRI_N+1	The priority of the interrupt number is N+1. The smaller the value, the higher the priority. The higher.
	11:8	-	reserve.
	7:4	PRI_N	The priority of the interrupt number is N. The smaller the value, the higher the priority. high.
	3:0	-	reserve.

5.3.2.9 Interrupt Status Register (VIC_ISR)

VIC_ISR indicates the interrupt vector number that the CPU is currently processing and the highest priority interrupt vector number that is waiting.

It is a read-only register for software query. Figure 5.17 describes the bit distribution of VIC_ISR, and Table 5.18 describes the bit definition of VIC_ISR.



Figure 5.17: Interrupt Status Register

Table 5.18: Interrupt Status Register Field Description

Bit	name	describe
31:21	Reserved	reserve.
20:12	VECT PENDING	Indicates the interrupt vector number with the highest priority currently in the waiting state.
11:9	Reserved	reserve.
8:0	VECTACTIVE	Indicates the interrupt vector number that the CPU is currently processing.

5.3.2.10 Interrupt Priority Threshold Register (VIC_IPTR)

VIC_IPTR defines the priority threshold at which the interrupt request currently in the waiting state can initiate interrupt preemption.

The priority of the interrupt request must be higher than the priority threshold defined by VIC_IPTR to initiate an interrupt preemption request. Figure 5.18 describes

The bit distribution of VIC_IPTR, Table 5.19 describes the bit definition of VIC_IPTR.

31	30	17:16	8:7	0
EN	Reserved	VECTTHRESHOLD	PRIOTHRESHOLD	

Figure 5.18: Interrupt Priority Threshold Register

Table 5.19: Interrupt Priority Threshold Register Field Description

Bit	name	describe
31	EN	Interrupt priority threshold valid bits: 0: interrupt preemption does not require a priority higher than the threshold; 1: interrupt preemption requires priority higher than the threshold.
30:17	Reserved	reserve.
16:8	VECTTHRESHOLD	Indicates the interrupt vector number corresponding to the priority threshold. CPU from VECTTHRESHOLD corresponding interrupt service routine Upon exit, the hardware automatically clears the interrupt priority threshold valid bit.
7:0	PRIOTHRESHOLD	Indicates the priority threshold for interrupt preemption. Note: The priority in E804 is determined by the number of configured interrupts The number can be set to [7:6], [7:5] or [7:4] bits to represent the priority threshold.

5.3.2.11 Tspend interrupt enable setting register (VIC_TSPEND)

VIC_TSPEND is used to enable the tspend interrupt and feedback the enable status of the tspend interrupt. Figure 5.19 describes VIC_TSPEND

Table 5.20 describes the bit definition of VIC_TSPEND.

31	1	0
	Reserved	
	SETENA	

Figure 5.19: Tspend interrupt enable setting register

Table 5.20: Tspend interrupt enable setting register field definition

Bit	name	describe
0:0	SETENA	sets the enable and reads the enable status of the tspend interrupt: Read Operation: 0: The corresponding tspend interrupt is not enabled; 1: The corresponding tspend interrupt is enabled. Write Operation: 0: invalid; 1: Enable tspend interrupt.

5.3.2.12 Tspend Interrupt Response Status Register (VIC_TSABR)

VIC_TSABR is used to indicate the current Active state of the tspend interrupt. It is a register for software to query. In addition, the software can When initializing VIC, clear the Active status of all tspend interrupts to 0. Figure 5.20 describes the bit distribution of VIC_TSABR, Table 5.21 Describes the bit definitions of VIC_TSABR.

31	Reserved	0
----	----------	---

Figure 5.20: Tspend interrupt response status register

Table 5.21: Tspend interrupt response status register field description

Bit	name	describe
0	Active	<p>Query bit, indicating whether the tspend interrupt has been responded to by the CPU but not yet processed. Only the lowest bit is valid.</p> <p>Read operation 0 is not responded by the CPU; 1 has been responded to by the CPU but has not been processed yet.</p> <p>Writing 0 clears the Active state of the tspend interrupt. (Software cannot write 1 to this register, otherwise it will cause unexpected errors)</p>

5.3.2.13 Tspend Interrupt Priority Setting Register (VIC_TSPR)

VIC_TSPR provides the priority setting for the tspend interrupt. The priority of the tspend interrupt needs to be set to the lowest. Figure 5.21 describes Bit distribution of VIC_TSPR, Table 5.22 describes the bit definition of VIC_TSPR.

31	Reserved	8:7	PRI	0
----	----------	-----	-----	---

Figure 5.21: Tspend interrupt priority setting register

Table 5.22: Tspend interrupt priority setting register field description

Bit	Name Description	
2 Bit 4 Priority Levels (The number of interrupts is less than or equal to 32)	31:8	- reserve.
	7:6	PRI is recommended to be set to the lowest priority 2'b11.
	5:0	- reserve.
3 bits 8 priority levels (Number of interrupts is 64)	31:8	- reserve.
	7:5	PRI is recommended to be set to the lowest priority 3'b111.
	5:0	- reserve.
4 bits 16 priority levels (Number of interrupts is 96 or 128)	31:8	- reserve.
	7:4	PRI is recommended to be set to the lowest priority 4'b1111.
	5:0	- reserve.

5.3.3 Interrupt handling mechanism

The vectored interrupt controller supports both level interrupts and pulse interrupts.

For level interrupts, the vector interrupt controller samples the high level of the interrupt valid signal and sets the corresponding interrupt to the waiting state, and then requests CPU responds. Level interrupts require that the interrupt service routine clear the valid signal of the peripheral interrupt source, otherwise the interrupt controller will re-initiate an interrupt request to the CPU when the interrupt exits. Based on this feature, the peripheral can always set the interrupt signal until the interrupt handler is no longer needed.

For pulse interrupts, also known as edge interrupts, the vector interrupt controller samples the rising edge of the interrupt valid signal, then sets the corresponding interrupt to the waiting state, and then initiates an interrupt request to the CPU. In order to ensure that the vector interrupt controller detects the pulse interrupt, the peripheral needs to keep the interrupt signal for at least one CPU clock cycle. Before the CPU responds to the pulse interrupt request, if the pulse interrupt source initiates multiple interrupt requests to the vector interrupt controller, the vector interrupt controller will only record the interrupt request once; after the CPU responds to the pulse interrupt request, if the pulse interrupt source initiates a request to the interrupt controller again, the vector interrupt controller will trigger the corresponding interrupt to enter the waiting state again. The interrupt request in the waiting state can only be responded to by the CPU again after the last interrupt exits.

In addition, the vector interrupt controller supports software interrupts. Software can set the interrupt setup wait register (VIC_ISPR) to high corresponding

The interrupt wait state bit triggers the interrupt to enter the wait state and send an interrupt request to the CPU.

When the processor responds to an interrupt request, the vector interrupt controller automatically clears the wait state bit of the corresponding interrupt. You can also clear the wait state bit of the corresponding interrupt by setting the interrupt clear wait register (VIC_ICPR). For level interrupts, if the interrupt valid signal continues to be high, the wait state bit cannot be cleared by setting the VIC_ICPR register.

5.3.3.1 Interrupt Status Bit

VIC provides 2 bits of status for each interrupt source, namely:

- Pending: Indicates that the interrupt is in a waiting state, that is, the interrupt request is waiting for the CPU to respond.

– 0: indicates that the interrupt is not in the waiting state;

– 1: Indicates that the interrupt is already in the waiting state.

- Active: Indicates that the interrupt request has been responded to by the CPU but has not yet been processed.

– 0: indicates that the interrupt request has not been responded to by the CPU;

– 1: Indicates that the interrupt request has been responded to by the CPU but has not yet been processed.

The setting conditions of the pending bit:

1. Level interrupt source, the interrupt source valid signal is set high, and Active is low or the CPU is exiting the interrupt service routine;
2. Pulse interrupt source, rising edge is valid;
3. Software sets ISPR.

The clearing condition of the pending bit:

1. The CPU responds to the interrupt request;
2. Clear ICPR by software.

Conditions for setting the Active bit: The CPU responds to the interrupt request.

Active bit clear condition: CPU exits the interrupt service routine.

Note: For level interrupt sources, since the level interrupt source request needs to be pulled low in the interrupt service program, it is only necessary to sample the interrupt source valid signal and set the Pending bit when Active is low or exit the interrupt service program; for pulse interrupt sources, since the request signal will be automatically pulled low, it is necessary to sample the rising edge of the pulse interrupt signal in real time and set the Pending bit.

5.3.3.2 Interrupt Priority

VIC provides priority settings for each interrupt source through the interrupt priority setting register (IPR0-IPR31). When the number of interrupts is less than or equal to 32, the hardware can provide 4 priorities; when the number of interrupts is less than or equal to 64, the hardware can provide 8 priorities; when the number of interrupts is 96 or 128, the hardware can provide 16 priorities. The lower the priority number, the higher the priority. For details, refer to the interrupt priority setting register ([VIC_IPR0 - VIC_IPR31](#)).

When the software does not set the priority register, the priority of all interrupt sources defaults to the highest priority -

0.

When multiple interrupts are in the pending state, the VIC arbitrates the interrupt request with the highest priority according to the priority of each interrupt and submits it to the CPU for processing. For example, if two interrupt requests IRQ0 and IRQ1 are in the pending state at the same time, if the priority number of IRQ1 is smaller than IRQ0, that is, the priority of IRQ1 is higher than IRQ0, IRQ0 will be submitted to the CPU for processing first.

When multiple pending interrupts have the same priority number, the order of interrupt submission is determined by the interrupt number, and the interrupt with a smaller number is submitted to the CPU for processing first. For example, two interrupt requests IRQ0 and IRQ1 have the same interrupt priority, and the interrupt source number of IRQ0 is smaller than that of IRQ1, so IRQ0 is submitted to the CPU for processing first.

5.3.3.3 Interrupt vector number

The interrupt vector number is the position number of the interrupt request in the exception vector table. Table 5.23 shows the exception vector table of E804. The vectors 0-30 at the beginning are used for internal identification of the processor; vector 31 is reserved; vector numbers starting from 32 are reserved for external interrupt requests, and each interrupt source corresponds to an interrupt vector number.

Table 5.23: Interrupt vector number description

Vector Number	Vector Offset (Hexadecimal)	Vector Allocation
0	000	Restart abnormality.
1	004	Unaligned access exception.
2	008	Access error exception.
3	00C	Division by 0 is an exception.
4	010	Illegal instruction exception.
5	014	Privilege violation exception.
6	018	Trapping exceptions.
7	01C	Breakpoint exception.
8	020	Unrecoverable error exception.
9	024	IDLY abnormal.
10	028	Normal interrupt. (auto-vectored).
11-15	02C - 03C	reserve.
16 ~ 19	040 ~ 04C	Trap instruction exceptions (TRAP #0-3).
20-21	050-054	reserve.
Twenty two	058	Tspend interrupt.
23-29	05C-074	reserve.
30	078	Floating point exception.
31	07C	reserve.
32	080	IRQ0.
33	084	IRQ1.
.....
32+n	0x80+4n	IRQn.

5.3.3.4 Interrupt handling process

The interrupt handling process can be divided into the following steps:

- Interrupt source request synchronization: When an external device generates an interrupt source request, the system completes the synchronization of the asynchronous interrupt source request to the CPUCLK clock domain
Operation, set pad_vic_int_vld high;
- Sampling of interrupt source requests: VIC samples interrupt source requests according to the type of interrupt source; when a valid interrupt request is sampled,
Set the pending status bit to trigger the corresponding interrupt to enter the waiting state;
- Interrupt request initiation: among all the interrupts in waiting state, an interrupt request is initiated to the CPU after priority arbitration;
- Interrupt response: The CPU responds to the interrupt when the instruction is retired, returns the interrupt response signal to the VIC, and updates the PSR and PC to the EPSR
and EPC, and update PSR.VEC with the interrupt vector number of the responded interrupt, clear PSR.EE, and finally get the exception entry address; VIC root
According to the interrupt response signal, clear the Pending status bit of the corresponding interrupt and set its Active status bit;
- Interrupt context saving: First save the interrupt control register context {EPSR, EPC}, open PSR.EE and PSR.IE to enable interrupts
Nesting; then save the general register context;

- Interrupt transaction: The CPU starts to process the interrupt transaction. For level interrupts, the interrupt source signal needs to be cleared, otherwise the interrupt will be reset when the interrupt exits.

Enter the interrupt;

- Interrupt context recovery and interrupt exit: first restore the general register context; then restore the interrupt control register context (EPSR, EPC), restore EPC and EPSR to PC and PSR, and exit the interrupt service routine; VIC receives the interrupt exit signal and clears the Active status bit.

The interrupt context can be saved by executing NIE and IPUSH instructions at the beginning of the interrupt service routine, and the interrupt context can be restored and exited by executing IPOP and NIR instructions at the end of the interrupt service routine.

The synchronization of interrupt source request is completed by the system and is not implemented inside VIC. Figure 5.22 shows an example of pad_vic_int_vld signal synchronization. The interrupt source request signal irq_n is synchronized to the CPU clock domain through the two-level CPUCLK register. In addition, the interrupt source type configuration signal pad_vic_int_cfg is a fixed value for a given interrupt source. 0 indicates a level interrupt source and 1 indicates a pulse interrupt source, so synchronization is not required.

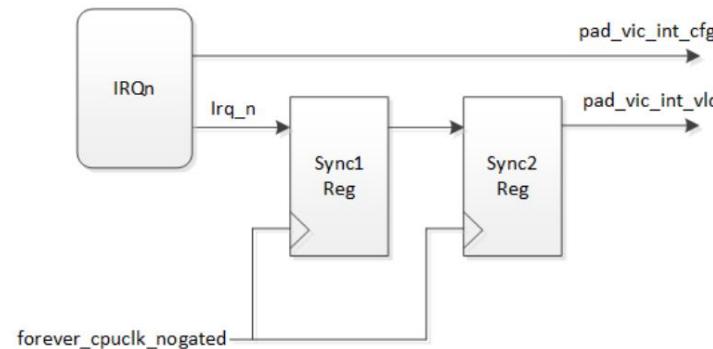


Figure 5.22: Example of interrupt source request synchronization circuit

5.3.3.5 Interrupt Nesting

VIC supports interrupt nesting, which allows interrupts with higher priority to be preempted during interrupt processing, thereby improving the real-time nature of interrupt response. When there are multiple levels of interrupt nesting, the priority of the preempted interrupt may need to be changed in certain scenarios, such as when the response time of a low-priority interrupt reaches the maximum limit. At this point, the software can set the interrupt priority threshold register to increase the priority condition for interrupt preemption, so that the preempted low-priority interrupt can be responded to in a timely manner.

Interrupt nesting priority conditions

The priority conditions for interrupt preemption can be divided into two types, as shown below:

- When the interrupt priority threshold is not enabled, the interrupt preemption priority must be higher than the priority of the interrupt currently being processed by the CPU;
 - Priority cannot be preempted;
- When the interrupt priority threshold is enabled, the interrupt preemption priority must not only be higher than the interrupt priority currently being processed by the CPU, but also higher than the interrupt priority currently being processed by the CPU.
 - The interrupt priority threshold register sets the threshold.

VIC supports dynamic adjustment of interrupt priority. When the priority of the preempted interrupt needs to be adjusted up or down, the interrupt priority threshold register is set while setting the interrupt priority setting register.

Figure 5.23 gives an example of interrupt preemption. The interrupt priority is set as: IRQ0<IRQ1<IRQ2<IRQ3; the order of interrupt source requests is: IRQ0>IRQ1>IRQ2>IRQ3. The CPU first responds to IRQ0. During the execution of the IRQ0 interrupt service program, IRQ1 with a higher priority comes, so IRQ0 is preempted and the CPU starts to execute the IRQ1 interrupt service program. Similarly, IRQ2 preempts IRQ1 and sets the interrupt priority threshold register (IPTR.VECTTHRESHOLD=IRQ0, IPTR.PRIOTHRESHOLD=0, IPTR.EN=1). When IRQ3 arrives, although its priority is higher than IRQ2, the priority of IRQ3 is not higher than the interrupt represented by the priority threshold set in IPTR, so IRQ3 cannot preempt IRQ2. IRQ3 waits until the execution of the IRQ0 interrupt service program ends and the hardware automatically clears IPTR.EN before it gets a CPU response.

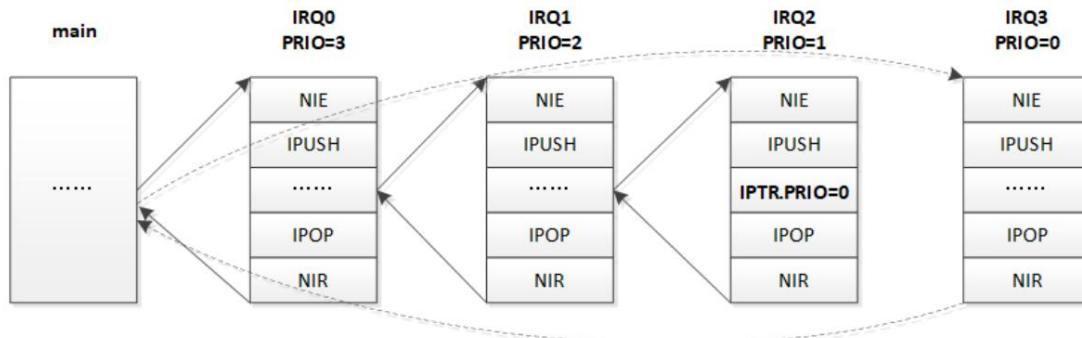


Figure 5.23: Interrupt nesting priority example

Interrupt nesting timing conditions

Interrupt preemption, while meeting the priority level, also needs to determine the current interrupt response stage. The interrupt response process refers to the interrupt handling process , and the nesting-related stages are mainly divided into the following stages:

- (1) Update of EPSR, EPC, PSR and reading of abnormal entry address
- (2) NIE instruction
- (3) IPUSH instruction
- (4) Interrupt transaction
- (5) IPOP instruction
- (6) NIR instructions.

In order to ensure the preservation and restoration of the interrupt nesting context, the CPU cannot be interrupted during the following stages:

- In the process of updating EPSR, EPC, PSR and obtaining the exception entry address after interrupt response;
- During the execution of a NIE directive, and including the retirement of a directive;
- The PSR.IC bit is off, and IPUSH and IPOP instructions are executing, excluding instruction retirement;
- During the execution of NIR instructions, instruction retirement is not included.

The CPU can safely and reliably respond to new interrupts in the following stages:

- During normal program execution, before an interrupt response;
- When IPUSH, IPOP directives retire;

- PSR.IC bit is on, during execution of IPUSH and IPOP instructions;
- When the NIR directs retirement;
- Interrupt the process of transaction processing.

When the PSR.IC bit is on, the IPUSH and IPOP instructions can respond to interrupts during execution, so the instruction needs to be re-executed when the interrupt returns. In the case where the IPUSH or IPOP instruction responds to an interrupt when it retires, the next instruction of the IPUSH/IPOP is directly executed after the interrupt exits, and the IPUSH/IPOP instruction does not need to be re-executed. The NIR instruction cannot be interrupted during execution, but it can respond to interrupts when it retires. If the interrupt is hit on the NIR instruction, the CPU directly pushes the return address of the NIR into the stack when the NIR instruction retires, and returns to the target address of the NIR when the interrupt exits.

Figure 5.24 shows the process of IRQ0/IRQ1/IRQ2/IRQ3 interrupt nesting. After IRQ0 is responded to by the CPU, IRQ1 with a higher priority is generated. When PSR.IC is turned on, IRQ1 is responded to when the IPUSH instruction is executed. IRQ2 is generated when IRQ1 handles the interrupt transaction, so it can be responded to by the CPU immediately. IRQ3 is generated when IRQ2 executes the NIR instruction, and IRQ3 is responded to when the NIR instruction retires. When IRQ3 is processed and exits the interrupt service routine, it directly returns to the point where IRQ1 is interrupted by IRQ2. When IRQ1 returns to IRQ0, the IPUSH instruction needs to be re-executed.

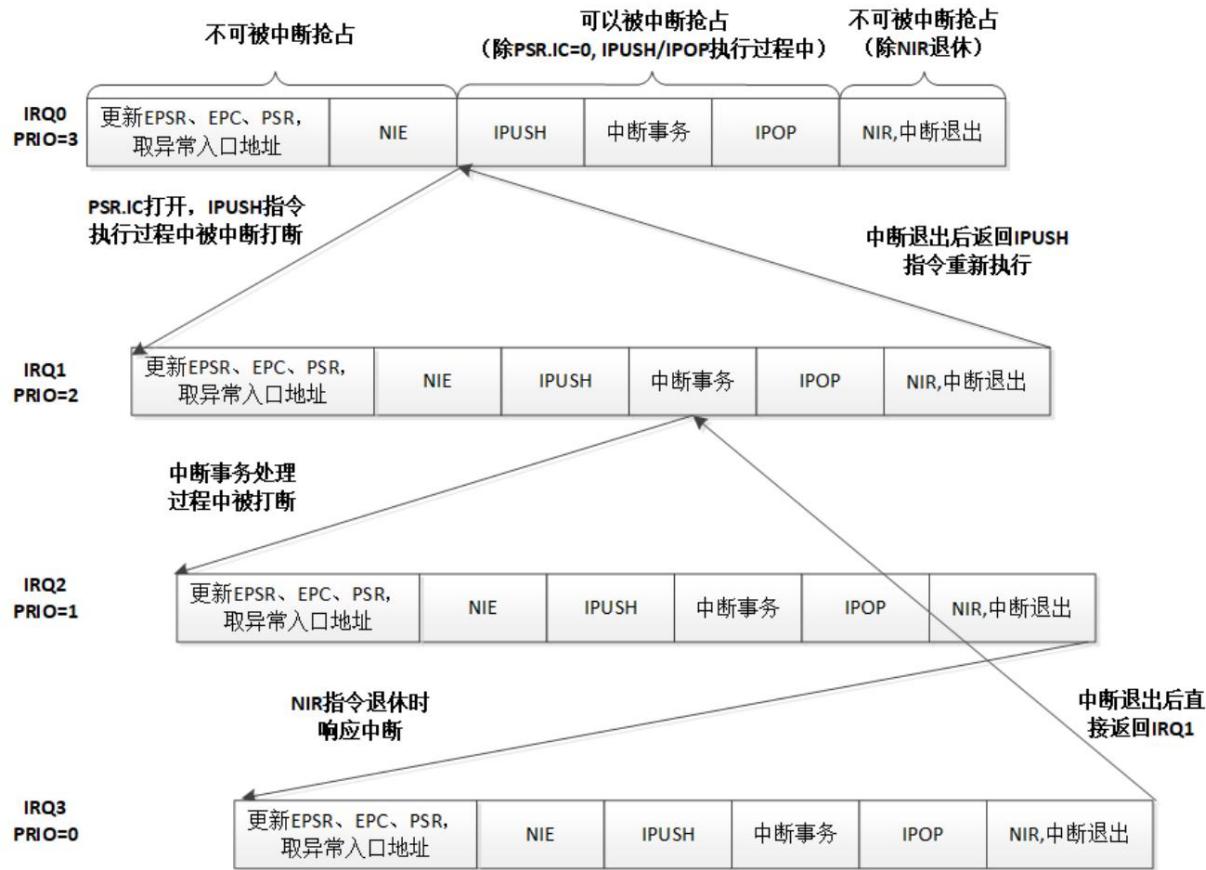


Figure 5.24: Example of interrupt nesting timing conditions

5.3.4 Tspend Interrupt

The Tspend interrupt controller provides an interrupt that does not occupy the external interrupt vector number (32-159). The Tspend interrupt vector number is 22. The function and setting of the tspend interrupt are basically the same as other interrupts. The difference between the tspend interrupt and other interrupts is:

1. Tspend interrupt, no external device is required to generate interrupt source request, no pending status bit, software sets VIC_TSPEND to enable

Enters the waiting state.

2. When the Tspend interrupt has the same priority as other interrupts, the interrupt priority is not determined by the interrupt number. When the interrupt priority is the same, Tspend

The interrupt priority is the lowest; when there are no other interrupts of the same level in the pending state, the tspend interrupt is processed.

5.3.5 Operation steps

To ensure that VIC can generate interrupt requests of the expected priority, the user needs to set the interrupt priority in advance and enable the response interrupt. The steps are as follows:

1. First, set VIC_IPR0-31 to configure the appropriate priority for each interrupt;

2. Then set VIC_ISER to enable the corresponding interrupt.

In addition, VIC supports software setting VIC_ISPR to generate interrupt requests. Before setting VIC_ISPR, you must also follow the above steps to configure

Interrupt priority and interrupt enable bits.

Note: Before the CPU responds to the interrupt request triggered by VIC, PSR.IE and PSR.EE need to be enabled to turn on the CPU interrupt response enable. Otherwise, the CPU cannot respond to the interrupt.

Before VIC samples external interrupt sources to generate interrupt low-power wake-up, VIC_IWER must be set first to set the corresponding low-power wake-up enable high bit, otherwise the low power wake-up request cannot be generated.

Note: For a request from an external interrupt source, a low-power wake-up request can be generated as long as the low-power wake-up function of the corresponding interrupt is enabled (corresponding to VIC_IWER), without relying on the interrupt enable of the interrupt itself (corresponding to VIC_ISER).

5.3.6 Interface Signals

The interface signals of the vector interrupt controller can be divided into three groups:

- Vectored interrupt controller interface with interrupt sources: VIC receives and samples interrupt request signals from interrupt sources;
- Interface between vector interrupt controller and TCIP: TCIP provides a set of interfaces for reading and writing VIC registers to complete VIC related control registers.
- Read and write interrupts, such as enabling interrupts, setting interrupt priorities, and querying the interrupts currently being processed;
- Interface between the vector interrupt controller and the processor core E804: VIC sends an interrupt request and the corresponding interrupt vector number to E804, and E804 returns Returns the VIC interrupt response and interrupt return indication signals as well as the interrupt vector number currently being processed.

Table 5.24: Vectored Interrupt Controller Interface Signals

Signal	I/O Reset Definition		
name Tightly coupled IP bus interface signal:			
tcipif_vic_sel	I	0	Selected signal: Indicates that the vectored interrupt controller is selected and data transfer is performed. 1: Select VIC; 0: VIC is not selected.
tcipif_vic_addr[15:0]	I	-	Address bus: 16-bit address bus (cut the lower 16 bits of the 32-bit address bus), Indicates the access address.
tcipif_vic_write	I	0	Read and write indication signal: Indicates whether the current TCIP access is to read or write data: 1: write access; 0: Read access.
tcipif_vic_wdata[31:0]	I	-	Write data bus: 32-bit write data bus.
vic_tcipif_rdata[31:0]	O	-	Read data bus: 32-bit read data bus.
vic_tcipif_cmplt	O	0	Transmission completion indication signal: When valid, indicates that the current transfer is complete.
Processor interrupt handshake signals:			
vic_pad_int_b	O	1	Interrupt request signal: A low level indicates a normal interrupt request.
vic_pad_intraw_b	O	1	Interrupt wake-up request signal: A low level indicates a low-power wake-up request.
vic_pad_int_vec_b[7:0]	O	-	Interrupt vector number signal: Provides the core with the vector number for interrupt processing.
pad_vic_int_ack	I	0	Interrupt response signal: Indicates that the CPU has responded to the current interrupt request.
pad_vic_int_exit	I	0	Interrupt service program exit signal: Instructs the CPU to exit the interrupt service routine.
pad_vic_int_vec[7:0]	I	-	Indicates the interrupt vector number that the CPU is processing.
pad_vic_ack_vec[7:0]	I	-	Indicates the interrupt vector number that the CPU responds to.
interrupt source signal:			
pad_vic_int_cfg[127:0]	I	-	Interrupt source type configuration signal: 0: level interrupt source; 1: Pulse interrupt source.
pad_vic_int_vld[127:0]	I	0	Interrupt valid signal: High level is valid.

Table 5.25: Vectored Interrupt Controller Interface Signals

Signal	I/O Reset Definition		
Name Clock Signal:			
forever_cpuclk	I	-	VIC's working clock: All registers in the vector interrupt controller that are not related to low power wake-up are working For this clock, the clock signal can be turned off in low power consumption mode.
forever_cpuclk_nogated	I	-	VIC interrupts the clock sampling circuit: The registers related to low power wake-up in the vector interrupt controller are all working The clock signal cannot be turned off in low power mode.
Reset signal:			
cpurst_b	I	-	Vectored Interrupt Controller Reset Signal: When the level is low, the vector interrupt controller is initialized. The controller uses asynchronous reset mode.
Other signals:			
pad_yy_gate_clk_en_b	I	-	Gated clock enable signal: Only when this signal is valid, the gated clock of the internal module of VIC To be effective. When this signal is not used, you need to connect 1.
pad_yy_test_mode	I	-	Enter test mode: Make VIC enter test mode, then VIC clock is test clock (pad_had_jtg_tclk). Only VIC enters test mode, and The processor input signal pad_yy_scan_enable is valid. It can be tested through the scan chain. When this signal is not used, it needs to be connected to 0.

When the CPU does not have an integrated vector interrupt controller, the external interrupt controller arbitrates the interrupt source and generates an interrupt request to send to the CPU. The CPU synchronizes the interrupt request signal from the external input and passes it to the core for processing. In addition, the CPU also needs to generate low-level interrupts to the external IP. The power consumption wake-up signal is synchronized.

Table 5.26: Interface signals without vectored interrupt controller configuration

Signal	I/O Reset Definition		
name Tightly coupled IP bus interface signal:			
pad_cpu_int_b	I	1	Interrupt request signal: A low level indicates a normal interrupt request.
pad_cpu_intraw_b	I	1 Low power wake-up request signal: A low level indicates a low-power wake-up request.	
pad_cpu_int_vec_b[7:0]	I	-	Interrupt vector number: Indicates the interrupt vector number corresponding to the current interrupt request.
Clock signal:			
forever_cpubclk_nogated	I	-	VIC Synchronous Circuit Clock: Used for synchronization of interrupt requests and low-power wake-up signals.
clk_en	I	-	System clock synchronization enable signal.

5.3.7 Interrupt Setting Example

```
//Set the ee and ie bits in psr so that the cpu responds to the interrupt

psrset ee,ie

//Set the interrupt enable bit for interrupt number 32-35

lw r1,0x0
bseti r1, 0x0 //Set the enable bit of interrupt 32
bseti r1, 0x1 //Set the enable bit of interrupt 33
bseti r1, 0x2 //Set the enable bit of interrupt 34
bseti r1, 0x3 //Set the enable bit of interrupt 35
lw r2, 0xe000e100 //The address corresponding to the interrupt enable register ISER
st.w r1, (r2, 0x0) // Enable 4 interrupts with interrupt numbers 32-35

// Other interrupts are enabled in the same way as above. The corresponding interrupts are enabled by setting different bits of the ISER register.

//Set the interrupt priority register IPR0 and set the priority of interrupts 32-35

lw r1,0x0

//Set IPR0[7:6] to set the priority of interrupt 32 to 2'b11, the lowest priority

//Set IPR0[15:14] to set the priority of interrupt 33 to 2'b00, the highest priority
```

(Continued on next page)

(Continued from previous page)

```

//Set IPR0[23:22] to set the priority of interrupt 34 to 2'b10

//Set IPR0[31:30] to set the priority of interrupt 35 to 2'b10

        bseti r1, 0x6 //Set the priority of interrupt No. 32 through the low two bits [7:6]bseti
        r1, 0x7 //Set the priority of interrupt No. 32 to the lowest, IPR0[7:6]=2'b11 bclri r1, 0x14 //Set the
        priority of interrupt No. 33 through the two bits [15:14]bclri r1, 0x15 //Set the priority
        of interrupt No. 33 to the highest, IPR0[15:14]=2'b00 bclri r1, 0x22 //Set the priority of interrupt No. 34
        through the two bits [23:22]bseti r1, 0x23 //Set the priority of interrupt No. 34 to 2,
        IPR0[23:22]=2'b10 bclri r1, 0x30 //Set the priority of interrupt No. 35 through the two bits [31:30]bseti
        r1, 0x31 //Set the priority of interrupt No. 35 The priority of the interrupt is 2,
        IPR0[31:30]=2'b10

//Write the set priority into IPR0

        Irw r2, 0xe000e400 //The address corresponding to the interrupt enable register
        IPR0 st.w r1, (r2, 0x0) //Complete the interrupt priority setting for No. 32-35

//The priority settings for other interrupts are the same as above, just set the corresponding interrupt priority registers IPR1-IPR7.

//Set the interrupt priority threshold register VIC_IPTR

        Irw r1, 0x2200 //Set the interrupt priority threshold register interrupt number, IPTR[16:8]=34 bseti
        r1, 0x0 //Enable interrupt priority threshold register

//The two bits of IPTR[7:6] can be used to preempt the interrupt priority of interrupt 34.

//Set the priority of interrupt 34 to be higher than 2'b01,
//That is, only the interrupt with priority 2'b00 (highest) can preempt interrupt 34

        bseti r1, 0x6
        bclri r1, 0x7

//Write the priority threshold, enable and interrupt number corresponding to interrupt number 34 into IPTR

        Irw r2, 0xe000ec04 //Interrupt priority threshold register address st.w
        r1, (r2, 0x0) //Complete the priority threshold setting for interrupt No. 34

```

Chapter 6 Instruction Set

6.1 Overview

The instruction set of E804 has high-level language features and is optimized for some frequently executed instructions. The instruction set includes standard arithmetic and logic instructions, bit operation instructions, byte extraction instructions, data transfer instructions, control flow change instructions and conditional execution instructions, which help reduce conditional transfers of short jumps.

E804 instructions have two widths: 16-bit instructions and 32-bit instructions. The instruction code is a mixture of the two types of instructions, and there is no additional overhead in switching between the two types of instructions.

6.2 32-bit instructions

This section mainly introduces the 32-bit instruction set of CSKY V2 implemented by E804.

6.2.1 32-bit instruction function classification

The 32-bit instruction set of E804 can be divided into the following types according to the functions implemented by the instructions:

- Data operation instructions;
- Branch instructions;
- Memory access instructions;
- Privileged instructions;
- Special function instructions;
- DSP extended instructions.

6.2.1.1 Data operation instructions

Data operation instructions can be further divided into:

Addition and subtraction instructions

Table 6.1: 32-bit addition and subtraction instruction list

	ADDU32 Unsigned addition instruction.
	ADDC32 Unsigned add with carry instruction.
	ADDI32 Unsigned immediate addition instruction.
	SUBU32 Unsigned subtraction instruction.
	SUBC32 Unsigned subtract with borrow instruction.
	SUBI32 Unsigned immediate subtraction instruction.
	RSUB32 Reverse subtraction instruction.
	IXH32 Index halfword instruction.
	IXW32 Index word instruction.
	IXD32 Indexed double word instruction.
INCF32	C is 0 immediate addition instruction.
INCT32	C is 1 immediate addition instruction.
DECFS32	C is 0. Immediate subtraction instruction.
DECT32	C is 1. Immediate subtraction instruction.
DEC GT32	Set C bit if subtraction is greater than zero.
DECLT32	Set C bit if subtraction is less than zero.
DECNE32	Set C bit if subtraction is not equal to zero.

Logical operation instructions

Table 6.2: 32-bit logical operation instruction list

	AND32 Bitwise AND instruction.
	ANDI32 Bitwise AND instruction for immediate values.
	ANDN32 Bitwise AND instruction.
	ANDNI32 Bitwise AND of immediate data.
	OR32 Bitwise OR instruction.
	ORI32 Immediate bitwise OR instruction.
	XOR32 Bitwise exclusive OR instruction.
	XORI32 Immediate bitwise exclusive OR instruction.
	NOR32 Bitwise OR instruction.
	NOT32 Bitwise NOT instruction.

Shift Instructions

Table 6.3: List of 32-bit shift instructions

LSL32 Logical Shift Left instruction.
LSLI32 Immediate logical left shift instruction.
LSLC32 Immediate Logical Shift Left to C instruction.
LSR32 Logical right shift instruction.
LSRI32 Immediate Logical Shift Right Instruction.
LSRC32 Immediate logical shift right to C instruction.
ASR32 Arithmetic Shift Right instruction.
ASRI32 Immediate Arithmetic Shift Right Instruction.
ASRC32 Immediate Arithmetic Shift Right to C instruction.
ROTL32 Rotate left instruction.
ROTLI32 Immediate value rotate left instruction.
XSR32 Extended right shift instruction.

Comparison Instructions

Table 6.4: List of 32-bit comparison instructions

CMPNE32 Not equal comparison instruction.
CMPNEI32 Immediate inequality comparison instruction.
CMPHS32 Unsigned greater than or equal to comparison instruction.
CMPHSI32 Immediate unsigned greater than or equal to comparison instruction.
CMPLT32 Signed less-than comparison instruction.
CMPLTI32 Immediate signed less than comparison instruction.
TST32 Zero test instruction.
TSTNBZ32 No byte equal to zero register test instruction.

Data transfer instructions

Table 6.5: 32-bit data transfer instruction list

MOV32 data transfer instruction.
MOVF32 C is 0 data transfer instruction.
MOVT32 C is 1 data transfer instruction.
MOVI32 immediate data transfer instruction.
MOVIH32 immediate high-order data transfer instruction.
MVCV32 C bit invert transfer instruction.
MVC32 C bit transfer instruction.
CLRF32 C is 0 clear instruction.
CLRT32 C is 1 to clear the instruction.
LRW32 memory read instruction.
GRS32 symbol generation instructions.

Bit operation instructions

Table 6.6: List of 32-bit operation instructions

BSETI32 Set immediate bit instruction.
BTSTI32 Immediate digital test instruction.

Extract Insert Instructions

Table 6.7: List of 32-bit extract and insert instructions

ZEXT32-bit extract and no sign-extend instruction.
SEXT32-bit extract and sign-extend instruction.
INS32 bit insert instruction.
ZEXTB 32-byte extract and no sign-extend instruction.
ZEXTH32 Halfword extract and no sign-extend instruction.
SEXTB 32-byte extract and sign-extend instruction.
SEXTH32 Extract halfword and sign-extend instruction.
XTRB0.32 extracts byte 0 without sign extension.
XTRB1.32 Extract byte 1 without sign extension instruction.
XTRB2.32 Extract byte 2 without sign extension instruction.
XTRB3.32 Extract byte 3 without sign extension instruction.
BREV32 Bit Reverse Instruction.
REVB32 Byte reverse instruction.
REVH32 Reverse bytes within halfword.

Multiplication and division instructions

Table 6.8: List of 32-bit multiplication and division instructions

MULT32	Multiplication Instructions
MULLL.S16 (MULSH32)*	16-bit signed multiply instruction.
MUL.(U/S)32	32-bit (unsigned/signed) multiplication instruction.
MULA.32.L	32-bit signed multiply-accumulate instruction with the lower 32 bits taken.
MULA.(U/S)32	32-bit (un/yes) signed multiply-accumulate instruction.
MULALL.S16.S	16-bit signed low halfword multiply-accumulate instruction with saturation.
DIVU32	Unsigned divide instruction.
DIVS32	Signed divide instruction.

Note: * The instruction MULSH32 in the historical version will be replaced by MULLL.S16, and the function remains unchanged.

Miscellaneous operation instructions

Table 6.9: List of 32-bit miscellaneous operation instructions

ABS32	Absolute value instruction.
FF0.32	Quick find 0 instruction.
FF1.32	Quick find 1 instruction.
BMASKI32	Immediate bit mask generation instruction.
BGENR32	register bit generation instruction.
BGENI32	Generate immediate digital instruction.

6.2.1.2 Branch instructions

Branch instructions can be further divided into:

Branch Instructions

Table 6.10: List of 32-bit branch instructions

BT32	C is 1 branch instruction.
BF32	C is 0 branch instruction.
BEZ32	register equals zero branch instruction.
BNEZ32	Register not equal to zero branch instruction.
BNEZAD32	Register decrement greater than zero branch instruction.
BHZ32	Branch if register greater than zero instruction.
BHSZ32	Branch if register is greater than or equal to zero.
BLZ32	Branch if register is less than zero.
BLSZ32	Branch if register is less than or equal to zero.

Jump Instructions

Table 6.11: List of 32-bit jump instructions

BR32	Unconditional jump instruction.
BSR32	Jump to subroutine instruction.
JMPI32	Indirect jump instruction.
JSRI32	Indirect jump to subroutine instruction.
JMP32	Register jump instruction.
JSR32	Register Jump to Subroutine instruction.
RTS32	Link register jump instruction.

6.2.1.3 Memory access instructions

Memory access instructions can be further divided into:

Immediate offset access instructions

Table 6.12: 32-bit immediate offset access instruction list

LD32.B	Unsigned byte-extension load instruction.
LD32.BS	Sign-extended byte load instruction.
LD32.H	Unsign-extend halfword load instruction.
LD32.HS	Sign-extend load halfword instruction.
LD32.W	word load instruction.
ST32.B	Byte store instruction.
ST32.H	Half-word store instruction.
ST32.W	Word store instruction.

Vector register offset access instructions

Table 6.13: 32-bit vector register offset access instruction list

LDR32.B	Register shift addressing unsigned extended byte load instruction.
LDR32.BS	Register shift addressed sign-extended byte load instruction.
LDR32.H	Register shift addressing unsigned-extended halfword load instruction.
LDR32.HS	Register shift addressing sign-extended load halfword instruction.
LDR32.W	Register shift addressed word load instruction.
STR32.B	Register shift addressed byte store instruction.
STR32.H	Register shift addressing half-word storage instruction.
STR32.W	Register shift addressed word store instruction.

Multiple register access instructions

Table 6.14: 32-bit multiple register access instruction list

LDQ32	Continuous four-word load instruction.
LDM32	Continuous multi-word load instruction.
STQ32	Continuous four-word storage instruction.
STM32	continuous multi-word storage instruction.
PUSH32	push instruction.
POP32	pop instruction.

Symbolic access instructions

Table 6.15: List of 32-bit symbol access instructions

LRS32.B	Byte symbol load instruction.
LRS32.H	Half-word load instruction.
LRS32.W	character load instruction.
SRS32.B	Byte signed store instruction.
SRS32.H	Half-word storage instruction.
SRS32.W	word storage instruction.

6.2.1.4 Privileged Instructions

Privileged instructions can be further divided into:

Control register operation instructions

Table 6.16: 32-bit control register operation instruction list

MFCR32	control register read transfer instruction.
MTCR32	control register write transfer instruction.
PSRSET32	PSR bit set instruction.
PSRCLR32	PSR bit clear instruction.

Low power instructions

Table 6.17: 32-bit low power instruction list

WAIT32	Enter low power wait mode instruction.
DOZE32	Enter low power sleep mode instruction.
STOP32	Enter low power stop mode instruction.

Exception return instruction

Table 6.18: 32-bit exception return instruction list

RTE32 exception and normal interrupt return instructions.

6.2.1.5 Special function instructions

Special function instructions are shown in [Table 6.19](#).

Table 6.19: 32-bit special function instruction list

SYNC32 CPU synchronization instruction.
BKPT32 breakpoint instruction.
SCE32 Conditional execution setup instruction.
IDLY32 Interrupt identification disable instruction.
TRAP32 Unconditional operating system trap instruction.

6.2.1.6 DSP extension instructions

When the E804 is equipped with a DSP expansion unit, it will support DSP expansion instructions.

Set to describe.

The mnemonic for each DSP instruction consists of three parts, each separated by a ". ". The first part identifies the basic operation of the instruction, and the second part

The first part identifies the data type of the operand, and the third part identifies the processing of the operation result. Taking "16-bit parallel signed addition instruction with saturation operation-PADD.S16.S" as an example, PADD indicates that this is a parallel addition instruction, S16 indicates that the operand is a signed half-word, and S indicates the processing of the operation result. Saturation operation.

Table 6.20: Common letters in instruction mnemonics and their meanings

Mnemonics	English expression meaning	
Part 1 Part 1	P(Parallel)	Parallel computing means single instruction multiple data operation.
Part 1 Part 1	R(fraction) Fractional operation.	
Part 2	X(Unalign) takes	The operand width is not aligned, or the operand position is not aligned.
	H(Half) the average.	
	(S/U)/(8/16/32) (signed/unsigned)/(byte/halfword/word)	If there is no sign bit indication, it means that the operation of this instruction is not related to the sign bit. Heart.
Part 3 Part 3	S (Saturate)	Saturation operation.
Part 3	R (Round)	Rounding operation.
	E (Extension) extension operation	that is, the operands are extended and then participate in the calculation to produce the result.

The specific instructions are shown in [Table 6.21](#).

Table 6.21: DSP extended instruction list

Instruction name	Instruction Description
Addition and subtraction instructions	
PADD.8	8-bit parallel addition instruction.
PADD.16	16-bit parallel addition instruction.
PADD.(U/S)8.S PADD.(U/	8-bit parallel (without/with) signed add instruction with saturation.
S)16.S ADD.(U/S)32.S	16-bit parallel (without/with) signed add instruction with saturation.
	32-bit (without/with) signed add instruction with saturation.
PSUB.8	8-bit parallel subtraction instruction.
PSUB.16	16-bit parallel subtraction instruction.
PSUB.(U/S)8.S	8-bit parallel (un/with) signed subtract instruction with saturation.
PSUB.(U/S)16.S	16-bit parallel (un/with) signed subtract instruction with saturation.
SUB(U/S)32.S	32-bit (un/with) signed subtract instruction with saturation.
PADDH.(U/S)8	8-bit parallel (un/signed) add averaging instruction.
PADDH.(U/S)16	16-bit parallel (un/signed) add averaging instruction.
ADDH.(U/S)32	32-bit (unsigned/signed) addition and averaging instruction.
PSUBH.(U/S)8	8-bit parallel (un/signed) subtraction averaging instruction.
PSUBH.(U/S)16	16-bit parallel (un/signed) subtraction averaging instruction.
SUBH.(U/S)32	32-bit (unsigned/signed) subtraction and averaging instruction.
PASX.16	16-bit parallel cross addition and subtraction instructions.
PSAX.16	16-bit parallel cross add and subtract instructions.
PASX.(U/S)16.S PSAX.(U/	16-bit parallel (unsigned/signed) interleaved add/subtract instructions with saturation.
S)16.S PASXH.(U/S)16	16-bit parallel (un/signed) interleaved subtract-add instructions with saturation.
PSAXH.(U/S)16	16-bit parallel (with/without sign) interleaved add/subtract averaging instructions.
	16-bit parallel (unsigned/signed) interleaved subtract-add average instruction.
ADD.64	64-bit addition instruction.
ADD.(U/S)64.S	64-bit (un/signed) add instruction with saturation.
SUB.64	64-bit subtraction instruction.
SUB.(U/S)64.S shift	64-bit (un/signed) subtraction instruction with saturation.
instruction	
ASRI.S32.R	Immediate arithmetic right shift with rounding.
ASR.S32.R	Arithmetic right shift instruction with rounding.
LSRI.U32.R	Immediate logical right shift instruction with rounding.
LSR.U32.R	Logical right shift instruction with rounding.
LSLI.(U/S)32.S LSL.	Immediate (unsigned/signed) logical left shift instruction with saturation.
(U/S)32.S	Signed logical left shift instruction with saturation.
PASRI.S16	16-bit parallel immediate arithmetic shift right instruction.
PASR.S16	16-bit parallel arithmetic right shift instruction.
PASRI.S16.R	16-bit parallel immediate arithmetic right shift instruction with rounding.

Continue on next page

Table 6.21 – Continued from previous page

Command name	Instruction Description
PASR.S16.R	16-bit parallel arithmetic right shift instruction with rounding.
PLSRI.U16	16-bit parallel immediate logical right shift instruction.
PLSR.U16	16-bit parallel logical right shift instruction.
PLSRI.U16.R	6-bit parallel immediate logical right shift instruction with rounding.
PLSR.U16.R	16-bit parallel logical right shift instruction with rounding.
PLSLI.16	16-bit parallel immediate logical left shift instruction.
PLSL.16	16-bit parallel logical shift left instruction.
PLSLI.(U/S)16.S PSL.(U/	16-bit parallel immediate (unsigned/signed) logical left shift instruction with saturation.
S)16.S Control flow data	16-bit parallel (un/with) signed logical left shift instruction with saturation.
quantity instruction (also called compare/select instruction)	
SEL	Bit select instruction.
PCMPNE.8	8-bit parallel compare-not-equal instruction.
PCMPNE.16	16-bit parallel compare-not-equal instruction.
PCMPSHS.(U/S)8 PCMPSHS.	8-bit parallel (no/yes) signed greater than or equal to comparison instructions.
(U/S)16 PCMPLT.(U/S)8	16-bit parallel (no/yes) signed greater than or equal to comparison instructions.
PCMPLT.(U/S)16 PMAX.(U/	8-bit parallel (without/with) signed less than compare select instruction.
S)8 PMAX.(U/S)16 MAX.(U/	16-bit parallel (no/yes) signed less than compare select instruction.
S)32 PMIN.(U/S)8 PMIN.	8-bit parallel (without/with) signed large value instruction.
(U/S)16 MIN.(U/S)32	16-bit parallel (without/with) signed large value instruction.
Data extraction/	32-bit (without/with) signed large value instruction.
unpacking/saturation	8-bit parallel (without/with) signed small instruction.
instructions	16-bit parallel (without/with) signed small instruction.
	32-bit (without/with) signed small-value instruction.
DEXTI	Immediate digital intercept instruction.
DEXT	Word interception instruction.
PKG	Immediate shift packed instructions.
PKGLL	Low halfword packing instruction.
PKG	Upper halfword packing instruction.
PEXT.U8.E	8-bit parallel unsigned-extend instruction.
PEXT.S8.E	8-bit parallel sign-extend instruction.
PEXTX.U8.E	8-bit parallel unsigned cross-extend instruction.
PEXTX.S8.E	8-bit parallel signed interleave extend instruction.
NARL	Intercept assembly instructions at low level.
NARH	High-order interception and splicing instructions.
NARLX	Low-level cross-interception assembly instructions.
NARHX	High-order cross-interception and splicing instructions.

Continue on next page

Table 6.21 – Continued from previous page

Instruction	Instruction Description
name CLIP.(U/S)32	Immediate value (no/yes) Sign-clip saturation instruction.
CLIP.(U/S)32 PCLIP.	(No/Yes) Sign clipping saturation instruction.
(U/S)16 PCLIP.(U/S)16	16-bit parallel immediate (without/with) sign clipping and saturation instructions.
	16-bit parallel (without/with) sign clipping and saturation instructions.
PABS.S8.S	8-bit parallel absolute value instruction with saturation.
PABS.S16.S	16-bit parallel absolute value instruction with saturation.
ABS.S32.S	32-bit absolute value instruction with saturation.
PNEG.S8.S	8-bit parallel negate instruction with saturation.
PNEGS16.S	16-bit parallel negate instruction with saturation.
NEG.S32.S	32-bit negate instruction with saturation.
DUP.8	8-bit operand copy instruction.
DUP.16	16-bit operand copy instruction.
Multiply/Multiply-Accumulate/Multiply-Accumulate-Subtract Instructions	
32x32 takes 64 bits	
MULS.(U/S)32 MULA.	32-bit (no/yes) signed multiply-accumulate-subtract instructions.
(U/S)32.S MULS.(U/S)32.S	32-bit (without/with) signed multiply-accumulate instruction with saturation.
32x32 Take the high word	32-bit (without/with) signed multiply-accumulate-subtract instruction with saturation.
MUL.S32.H	32-bit signed multiply extract high 32-bit instruction.
MUL.S32.RH	32-bit signed multiply with rounding.
RMUL.S32.H	32-bit signed fractional multiply extract high 32-bit instruction.
RMUL.S32.RH	32-bit signed fractional multiply with rounding.
MULA.S32.HS	32-bit signed multiply-accumulate fetch high 32-bit instruction with saturation.
MULS.S32.HS	32-bit signed multiply-accumulate-subtract-upper-32-bit instruction with saturation.
MULA.S32.RHS	32-bit signed multiply-accumulate fetch high 32 instruction with rounding and saturation.
MULS.S32.RHS	32-bit signed multiply-accumulate-subtract-upper-32 instruction with rounding and saturation.
32x16 High Word	
MULXL.S32	32-bit signed lower halfword unaligned multiply instruction.
MULXL.S32.R	32-bit signed low halfword unaligned multiply instruction with rounding.
MULXH.S32	32-bit signed upper halfword unaligned multiply instruction.
MULXH.S32.R	32-bit signed upper halfword unaligned multiply instruction with rounding.
RMULXL.S32	32-bit signed lower halfword unaligned fractional multiply instruction.
RMULXL.S32.R	32-bit signed lower halfword unaligned fractional multiply instruction with rounding.
RMULXH.S32	32-bit signed upper halfword unaligned fractional multiply instruction.
RMULXH.S32.R	32-bit signed upper halfword unaligned fractional multiply instruction with rounding.
MULAXL.S32.S	32-bit signed lower halfword unaligned multiply-accumulate instruction with saturation.
MULAXL.S32.RS	32-bit signed low halfword unaligned multiply-accumulate instruction with rounding and saturation.

Continue on next page

Table 6.21 – Continued from previous page

Command name	Instruction Description
MULAXH.S32.S	32-bit signed upper halfword unaligned multiply-accumulate instruction with saturation.
MULAXH.S32.RS	32-bit signed upper halfword unaligned multiply-accumulate instruction with rounding and saturation.
16x16, non-SIMD	
MULLL.S16	16-bit signed low halfword multiply instruction.
MULHH.S16	16-bit signed upper halfword multiply instruction.
MULHL.S16	16-bit signed high and low halfword multiply instruction.
RMULLL.S16	16-bit signed low halfword fractional multiply instruction.
RMULHH.S16	16-bit signed upper halfword fractional multiply instruction.
RMULHL.S16	16-bit signed high and low halfword fractional multiply instruction.
MULAHH.S16.S	16-bit signed upper halfword multiply-accumulate instruction with saturation.
MULAHL.S16.S	16-bit signed high and low halfword multiply-accumulate instruction with saturation.
MULALL.S16.E	16-bit signed low halfword multiply-accumulate instruction with extend operation.
MULAHH.S16.E	16-bit signed upper halfword multiply-accumulate instruction with extend operation.
MULAHL.S16.E	16-bit signed high and low halfword multiply-accumulate instruction with extend operation.
16x16, SIMD	
PMUL.(U/S)16 PMULX.	16-bit parallel (no/yes) signed multiply instruction.
(U/S)16	16-bit parallel (without/with) sign-crossed multiply instruction.
PRMUL.S16	16-bit parallel signed fractional multiply instructions.
PRMULX.(U/S)16	16-bit parallel (without/with) sign-interleaved fractional multiply instructions.
PRMUL.S16.H	16-bit parallel signed fractional multiply extract high 16-bit instruction.
PRMUL.S16.RH	16-bit parallel signed fractional multiply with rounding.
PRMULX.S16.H	16-bit parallel signed interleaved fractional multiply extract high 16-bit instruction.
PRMULX.S16.RH	16-bit parallel signed interleaved fractional multiply with rounding.
16x16, chain addition and subtraction	
MULCA.S16.S	16-bit signed multiply-chain-add instruction with saturation.
MULCAX.S16.S	16-bit signed cross-multiply chain-add instruction with saturation.
MULCS.S16	16-bit signed multiply-chain-subtract instruction.
MULCSR.S16	16-bit signed reverse multiply chain subtract instruction.
MULCSX.S16	16-bit signed cross-multiply chain-subtract instruction.
16x16, chain accumulation and subtraction	
MULACA.S16.S	16-bit signed multiply-chain-add-accumulate instruction with saturation.
MULACAX.S16.S	16-bit signed cross-multiply chain-add-accumulate instruction with saturation.
MULACS.S16.S	16-bit signed multiply-chain-subtract-accumulate instruction with saturation.
MULACSR.S16.S	16-bit signed inverse multiply-chain subtract-accumulate instruction with saturation.
MULACSX.S16.S	16-bit signed cross-multiply chain subtract-accumulate instruction with saturation.
MULSCA.S16.S	16-bit signed multiply-chain-add-accumulate instruction with saturation.
MULSCAX.S16.S	16-bit signed cross-multiply chain add-accumulate instruction with saturation.

Continue on next page

Table 6.21 – Continued from previous page

Command name	Instruction Description
MULACA.S16.E	16-bit signed multiply-chain-add-accumulate instruction with extend operation.
MULACAX.S16.E	16-bit signed cross-multiply chain-add-accumulate instruction with extend operation.
MULACS.S16.E	16-bit signed multiply-chain-subtract-accumulate instruction with extend operation.
MULACSR.S16.E	16-bit signed reverse multiply chain subtract accumulate instruction with extend operation.
MULACSRX.S16.E	16-bit signed cross-multiply chain subtract-accumulate instruction with extend operation.
MULSCA.S16.E	16-bit signed multiply-chain-add-accumulate instruction with extend operation.
MULSCAX.S16.E	16-bit signed cross-multiply chain add-accumulate instruction with extend operation.
Miscellaneous	
PSABSA.U8	8-bit parallel unsigned subtract absolute value chained add instruction.
PSABSAA.U8	8-bit parallel unsigned subtract absolute value chained add and accumulate instruction.
DIVSL	Signed long division instruction.
DIVUL	Unsigned long division instruction.
MULACA.S8	8-bit parallel signed multiply-chain-add-accumulate instruction.
Featured Technology 1: Loop Acceleration	
BLOOP	Loop body acceleration instructions.
Featured Technology 2: LD/ST	
LDBI.W	Word load instruction with incremented address.
LDBI.H	Unsigned halfword load instruction with incrementing address.
LDBI.HS	Signed halfword load instruction with incrementing address.
LDBI.B	Unsigned byte load instruction with incrementing address.
LDBI.BS	Signed byte load instruction with incrementing address.
PLDBI.D	Double word load instruction with incrementing address.
STBI.W	Word storage instruction with auto-increment address.
STBI.H	Half-word store instruction with incremented address.
STBI.B	Byte store instruction with incremented address.
LDBIR.W	Word load instruction with incremented register address.
LDBIR.H	Unsigned halfword load instruction with incrementing register address.
LDBIR.HS	Signed halfword load instruction with incrementing register address.
LDBIR.B	Unsigned byte load instruction with incrementing register address.
LDBIR.BS	Signed byte load instruction with incrementing register address.
PLDBIR.D	Double word load instruction with incremented register address.
STBIR.W	Word storage instruction that increments the register address.
STBIR.H	A half-word store instruction that increments the register address.
STBIR.B	Byte store instruction that increments the register address.

6.3 16-bit instructions

This section mainly introduces the 16-bit instruction set of E804.

6.3.1 16-bit instruction function classification

The 16-bit instruction set of E804 can be divided into the following types according to the functions implemented by the instructions:

- Data operation instructions;
- Branch instructions;
- Memory access instructions;
- Multiple register access instructions.

6.3.1.1 Data operation instructions

Data operation instructions can be further divided into:

Addition and subtraction instructions

Table 6.22: 16-bit addition and subtraction instruction list

ADDU16 Unsigned	addition instruction.
ADDC16 Unsigned	addition with carry instruction.
ADDI16 Unsigned	immediate addition instruction.
SUBU16 Unsigned	subtraction instruction.
SUBC16 Unsigned	subtract with borrow instruction.
SUBI16 Unsigned	immediate subtraction instruction.

Logical operation instructions

Table 6.23: 16-bit logical operation instruction list

AND16 Bitwise AND	instruction.
ANDN16 Bitwise NOT	instruction.
OR16 Bitwise OR	instruction.
XOR16 Bitwise exclusive OR	instruction.
NOR16 Bitwise OR	instruction.
NOT16 Bitwise NOT	instruction.

Shift Instructions

Table 6.24: List of 16-bit shift instructions

LSL16 Logical left shift instruction.
LSLI16 Immediate logical shift left instruction.
LSR16 Logical right shift instruction.
LSRI16 Immediate logical shift right instruction.
ASR16 Arithmetic shift right instruction.
ASRI16 Immediate Arithmetic Shift Right Instruction.
ROTL16 Rotate left instruction.

Comparison Instructions

Table 6.25: 16-bit comparison instruction list

CMPNE16 Not equal to comparison instruction.
CMPNEI16 Immediate inequality comparison instruction.
CMPHS16 Unsigned greater than or equal to comparison instruction.
CMPHSI16 Immediate unsigned greater than or equal to comparison instruction.
CMPLT16 Signed less than comparison instruction.
CMPLTI16 Immediate signed less than comparison instruction.
TST16 Zero test instruction.
TSTNBZ16 No Byte Equals Zero Register Test Instruction.

Data transfer instructions

Table 6.26: 16-bit data transfer instruction list

MOV16 data transfer instruction.
MOVI16 immediate data transfer instruction.
MVCV16 C-bit transfer instruction.
LRW16 Memory read instruction.

Bit operation instructions

Table 6.27: List of 16-bit bit operation instructions

BCLRI16 Immediate digital clear instruction.
BSETI16 Immediate bit set instruction.
BTSTI16 Immediate digital test instruction.

Extract Insert Instructions

Table 6.28: 16-bit extract and insert instruction list

ZEXTB	16-byte extract and no sign-extend instruction.
ZEXTH16	Halfword extract without sign-extend instruction.
SEXTB	16-byte extract and sign-extend instruction.
SEXTH16	Extract halfword and sign-extend instruction.
REVB16	Byte reverse instruction.
REVH16	Reverse bytes within halfword.

Multiplication and division instructions

Table 6.29: List of 16-bit multiplication instructions

MULT16	Multiplication instruction.
--------	-----------------------------

6.3.1.2 Branch instructions

Branch instructions can be further divided into:

Branch Instructions

Table 6.30: List of 16-bit branch instructions

BT16 C is 1	branch instruction.
BF16 C is 0	branch instruction.

Jump Instructions

Table 6.31: List of 16-bit jump instructions

BR16	Unconditional jump instruction.
JMP16	Register jump instruction.
JSR16	register jump to subroutine instruction.
RTS16	Link register jump instruction.

6.3.1.3 Memory access instructions

Memory access instructions can be further divided into:

Immediate offset access instructions

Table 6.32: 16-bit immediate offset access instruction list

LD16.B Unsigned	Extend byte load instruction.
LD16.H Unsigned	Extend halfword load instruction.
LD16.W Word	Load instruction.
ST16.B Byte store instruction.	
ST16.H Half-word store instruction.	
ST16.W Word	Storage instruction.

6.3.1.4 Multiple register access instructions

Table 6.33: 16-bit multiple register access instruction list

POP16	Pop instruction.
PUSH16	Push instruction.
IPOP16	Interrupt pop instruction.
IPUSH16	Interrupt push instruction.
NIE16	Interrupt nesting enable instruction.
NIR16	Interrupt nested return instruction.

Note: NIE16 and NIR16 need to be executed in superuser mode.

6.4 Instruction Set List

The E804 instruction set has high-level language features and is optimized for some frequently executed instructions. The instruction set includes standard arithmetic logic editing instructions, bit operation instructions, byte extraction instructions, data transfer instructions, control flow change instructions and conditional execution instructions. Helps reduce conditional branches that are short.

Table 6.34 lists all 16-bit and 32-bit instructions in the E804 instruction set.

Table 6.34: E804 instruction set

Assembly instructions	32-bit	16-bit assembly	format	Instruction Description
ABS	O	x	ABS32 RZ, RX	Absolute value instruction.
ADDC	OO	ADDC32 RZ,RX,RY ADDC16 RZ, RX		Unsigned add with carry instruction.
ADDI	OO	ADDI32 RZ, RX, OIMM12 ADDI16 RZ, OIMM8		Unsigned immediate addition instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format			Instruction Description
ADDU	OO ADDU32 RZ, RX, RY ADDU16 RZ, RX			Unsigned addition instruction.
AND	OO AND32 RZ, RX, RY AND16 RZ, RX			Bitwise AND instruction.
ANDI	O \times ANDI32 RZ, RX, IMM12 Bitwise AND instruction for immediate data.			
ANDN	OO ANDN32 RZ, RZ, RX ANDN16 RZ, RX			Bitwise NOT instruction.
ANDNI	O \times ANDNI32 RZ, RX, IMM12 Bitwise NOT of immediate data.			
ASR	OO ASR32 RZ, RX, RY ASR16 RZ, RX			Arithmetic right shift instruction.
ASRC	O \times ASRC32 RZ, RX, OIMM5 Immediate Arithmetic Shift Right to C instruction.			
ASRI	OO ASRI32 RZ, RX, IMM5 ASRI16 RZ, RX, IMM5			Immediate arithmetic shift right instruction.
BCLRI	OO BCLRI32 RZ, RX, IMM5 BCLRI16 RZ, IMM5			Immediate digital clear instruction.
BEZ	O \times BEZ32 RX, LABEL			Register equals zero branch instruction.
BF	OO BF32 LABEL BF16 LABEL			C is 0 branch instruction.
BGENI	O \times BGENI32 RZ, IMM5			Immediate digital generation instruction.
BGENR	O \times BGENR32 RZ, RX			Register bit generation instruction.
BZ	O \times BHSZ32 RX, LABEL			Branch if register is greater than or equal to zero.
BZ	O \times BHZ32 RX, LABEL			Register greater than zero branch instruction.
BKPT	\times O BKPT16			Breakpoint instruction.
BLS	O \times BLSZ32 RX, LABEL			Branch if register is less than or equal to zero.
BLZ	O \times BLZ32 RX, LABEL Branch if register is less than zero.			
BMASKI	O \times BMASKI32 RZ, OIMM5 Immediate digital mask generation instruction.			
BNEZ	O \times BNEZ32 RX, LABEL			Register is not equal to zero branch instruction.
BR	OO BR32 LABEL BR16 LABEL		Unconditional jump instruction.	
BREV	O \times BREV32 RZ, RX			Bit reversal instruction.
BSETI	OO BSETI32 RZ, RX, IMM5 BSETI16 RZ, MM5			Immediate bit setting instruction.
BSR	O \times BSR32 LABEL			Jump to subroutine instruction.
BT	OO BT32 LABEL BT16 LABEL			C is 1 branch instruction.
BTSTI	OO BTSTI32 RX, IMM5 BTSTI16 RX, IMM5			Immediate digital test instruction.
CLRF	O \times CLRF32 RZ			C is 0 to clear the instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format			Instruction Description
CLRT	O	x	CLRT32 RZ	C is 1 to clear the instruction.
CMPHS	OO	CMPHS32 RX, RY CMPHS16 RX, RY		Unsigned greater than or equal to comparison instruction.
CMPHSI	OO	CMPHSI32 RX, OIMM16 CMPHSI16 RX, IMM5		Immediate unsigned greater than or equal to comparison pointer make.
CMPLT	OO	CMPLT32 RX, RY CMPLT16 RX, RY		Signed less-than comparison instruction.
CMPLTI	OO	CMPLT132 RX, OIMM16 CMPLTI16 RX, OIMM5		Immediate signed less than comparison instruction.
CMPNE	OO	CMPNE32 RX, RY CMPNE16 RX, RY		Not equal to comparison instruction.
CMPNEI	OO	CMPNEI32 RX, IMM16 CMPNEI16 RX, IMM5		Immediate comparison instruction.
DECDF	O	x	DECDF32 RZ, RX, IMM5	C is 0 immediate subtraction instruction.
DECGBT	O	x	DECGBT32 RZ, RX, IMM5 Set C bit if subtraction is greater than zero.	
DECLT	O	x	DECLT32 RZ, RX, IMM5 Set C bit if subtraction is less than zero.	
DECNE	O	x	DECNE32 RZ, RX, IMM5 Set C bit if subtraction is not equal to zero.	
DECT	O	x	DECT32 RZ, RX, IMM5 Signed	C is 1 immediate subtraction instruction.
DIVS	O	x	DIVS32 RZ, RX, RY	division instruction.
DIVU	O	x	DIVU32 RZ, RX, RY	Unsigned division instruction.
DOZE	O	x	DOZE32	low power sleep mode instruction.
FF0	O	x	FF0.32 RZ, RX	Quickly find 0 command.
FF1	O	x	FF1.32 RZ, RX	Quickly find 1 command.
GRS	O	x	GRS32 RZ, LABEL GRS32 RZ, IMM32	Symbol generation instructions.
IDLY	O	x	IDLY32 N	Interrupt recognition disable instruction.
INCF	O	x	INCF32 RZ, RX, IMM5 C is 1.	C is 0 immediate addition instruction.
INCT	O	x	INCT32 RZ, RX, IMM5 INS32 Immediate	addition instruction.
INS	O	x	RZ, RX, MSB, LSB Bit insert instruction.	
IPOP	x	O IPOP16		Interrupt pop instruction.
IPUSH	x	O IPUSH16		Interrupt push instruction.
IxD	O	x	IXD32 RZ, RX, RY	Indexes double-word instructions.
IX	O	x	IXH32 RZ, RX, RY	Index halfword instructions.
IX	O	x	IXW32 RZ, RX, RY	Index word directive.
JMP	OO	JMP32 RX JMP16 RX		Register jump instruction.
JMPI	O	x	JMPI32 LABEL	Indirect jump instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format			Instruction Description
JSR	OO JSR32 RX	JSR16 RX		Register jump to subroutine instruction.
JSRI	O	x	JSRI32 LABEL	Indirect jump to subroutine instruction.
LD.B	OO LD32.B RZ, (RX, DISP)	LD16.B RZ, (RX, DISP)		Unsign-extend byte load instruction.
LD.BS	O	x	LD32.BS RZ, (RX, DISP)	Sign-extended byte load instruction.
LD.H	OO LD32.H RZ, (RX, DISP)	LD16.H RZ, (RX, DISP)		Unsign-extend halfword load instruction.
LD.HS	O	x	LD32.HS RZ, (RX, DISP)	Sign-extend load halfword instruction.
LD.W	OO LD32.W RZ, (RX, DISP)	LD16.W RZ, (RX, DISP)		Word load instruction.
LDM	O	x	LDM32 RY-RZ, (RX)	Continuous multi-word load instructions.
LDQ	O	x	LDQ32 R4-R7, (RX)	Continuous four-word load instruction.
LDR.B	O	x	LDR32.B RZ, (RX, RY << 0) LDR32.B RZ, (RX, RY << 1) LDR32.B RZ, (RX, RY << 2) LDR32.B RZ, (RX, RY << 3)	Register shift addressing unsigned extended byte Load instructions.
LDR.BS	O	x	LDR32.BS RZ, (RX, RY << 0) LDR32.BS RZ, (RX, RY << 1) LDR32.BS RZ, (RX, RY << 2) LDR32.BS RZ, (RX, RY << 3)	Register shift addressing sign-extended byte Load instructions.
LDR.H	O	x	LDR32.H RZ, (RX, RY << 0) LDR32.H RZ, (RX, RY << 1) LDR32.H RZ, (RX, RY << 2) LDR32.H RZ, (RX, RY << 3)	Register shift addressing unsigned halfword Load instructions.
LDR.HS	O	x	LDR32.HS RZ, (RX, RY << 0) LDR32.HS RZ, (RX, RY << 1) LDR32.HS RZ, (RX, RY << 2) LDR32.HS RZ, (RX, RY << 3)	Register shift addressing sign-extended halfword Load instructions.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format		Instruction Description
LDR.W	O	×	LDR32.W RZ, (RX, RY << 0) LDR32.W RZ, (RX, RY << 1) LDR32.W RZ, (RX, RY << 2) LDR32.W RZ, (RX, RY << 3)
LRS.B	O	×	LRS32.B RZ, [LABEL]
LRS.H	O	×	LRS32.H RZ, [LABEL]
LRS.W	O	×	LRS32.W RZ, [LABEL]
LRW	OO LRW16 LABEL		Memory read instruction. LRW16 IMM32 LRW32 LABEL LRW32 IMM32
LSL	OO LSL32 RZ, RX, RY		Logical left shift instruction. LSL16 RZ, RY
LSLC	O	×	LSLC32 RZ, RX, OIMM5 Immediate logical left shift to C position instruction.
LSLI	OO LSLI32 RZ, RX, IMM5		Immediate logical left shift instruction. LSLI16 RZ, RX, IMM5
LSR	OO LSR32 RZ, RX, RY		Logical right shift instruction. LSR16 RZ, RY
LSRC	O	×	LSRC32 RZ, RX, OIMM5 Immediate logical right shift to C position instruction.
LSRI	OO LSRI32 RZ, RX, IMM5		Immediate logical right shift instruction. LSRI16 RZ, RX, IMM5
MFCR	O	×	MFCR32 RZ, CR <X, SEL> Control register read transfer instruction.
MOV	OO MOV16 RZ, RX	C is 0	Data transfer instructions.
MOVF	O	×	MOVF32 RZ, RX instruction.
MOVI	OO MOVI32 RZ, IMM16		immediate data transfer instruction. MOVI16 RZ, IMM8
MOVIH	O	×	MOVIH32 RZ, IMM16
MOVT	O	×	MOVT32 RZ, RX
MTCR	O	×	MTCR32 RX, CR<Z, SEL> Control register write transfer instruction.
MULA.32.L	O	×	MULA.32.L RZ, RX, RY 32-bit signed multiply-accumulate, take the lower 32 bits make.
MULALL.S16.S	O	×	MULALL.S16.S RZ, RX, RY 16-bit signed lower half with saturation operation
MULLL.S16(MULSH)	OO MULLL.S16	RZ, RX, RY	Word multiply-accumulate instructions. 16-bit signed multiply instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format			Instruction Description
MULT	OO MULT32 RZ, RX, RY MULT16 RZ, RX			Multiplication instruction.
MUL.(U/S)32	O × MUL.S32 RZ, RX, RY			32-bit (un/yes) signed multiply instruction.
MVC	O × MVC32 RZ			C bit transfer instruction.
MVCV	OO MVCV32 RZ MVCV16 RZ			C bit inverts the transfer instruction.
NIE	× O NIE			Interrupt nesting enable instruction.
NIR	× O NIR			Interrupt nested return instruction.
NOR	OO NOR32 RZ, RX, RY NOR16 RZ, RX			Bitwise OR instruction.
NOT	OO NOT32 RZ, RX NOT16 R			Bitwise NOT instruction.
OR	OO OR32 RZ, RX, RY OR16 RZ, RX			Bitwise OR instruction.
ORI	O × ORI32 RZ, RX, IMM16			Bitwise OR instruction for immediate values.
POP	OO POP32 REGLIST POP16 REGLIST			Pop instruction.
PSRCLR	O × PSRCLR32 EE, IE, FE, AF PSR bit clear instruction.			
PSRSET	O × PSRSET32 EE, IE, FE, AF PSR set bit instruction.			
PUSH	OO PUSH32 REGLIST PUSH16 REGLIST			Push instruction.
REVB	OO REVB32 RZ, RX REVB16 RZ, RX			Byte reversal instruction.
REV	OO REVH32 RZ, RX REVH16 RZ, RX			Instruction to reverse bytes within a halfword.
ROTL	OO ROTL32 RZ, RX, RY ROTL16 RZ, RY			Circular shift left instruction.
ROTLI	O × ROTLI32 RZ, RX, IMM5 Immediate value rotate left instruction.			
RSUB	O × RSUB32 RZ, RX, RY Reverse subtraction instruction.			
RTE	O × RTE32			Exception and normal interrupt return instructions.
RTS	OO RTS16			Link register jump instruction.
SCE	O × SCE32 COND			Conditional execution setup instructions.
SEXT	O × SEXT32 RZ, RX, MSB, LSB bit extract and sign extend instruction.			
SEXTB	OO SEXTB32 RZ, RX SEXTB16 RZ, RX			Byte extract and sign-extend instructions.
SEXTH	OO SEXTH32 RZ, RX SEXTH16 RZ, RX			Halfword extract and sign-extend instructions.
SRS.B	O × SRS32.B RZ, [LABEL]			Byte symbol storage instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit	16-bit assembly format	Instruction Description	
SRS.H	O	×	SRS32.H RZ, [LABEL]	Half-word store instruction.
SRS.W	O	×	SRS32.W RZ,[LABEL]	Word number storage instruction.
ST.B	OO ST32.B RZ, (RX, DISP)		ST16.B RZ,(RX, DISP)	Byte storage instruction.
ST.H	OO ST32.H RZ, (RX, DISP)		ST16.H RZ,(RX, DISP)	Halfword store instruction.
ST.W	OO ST32.W RZ, (RX, DISP)		ST16.W RZ,(RX, DISP)	Word storage instruction.
STM	O	×	STM32 RY-RZ, (RX)	Continuous multi-word storage instructions.
STOP	O	×	STOP32	Enter low power suspend mode instruction.
STQ	O	×	STQ32 R4-R7, (RX)	Continuous four-word storage instruction.
STR.B	O	×	STR32.B RZ, (RX, RY << 0) STR32.B RZ,(RX, RY << 1) STR32.B RZ, (RX, RY << 2) STR32.B RZ, (RX, RY << 3)	Register shift addressed byte store instruction.
STR.H	O	×	STR32.H RZ, (RX, RY << 0) STR32.H RZ, (RX, RY << 1) STR32.H RZ, (RX, RY << 2) STR32.H RZ, (RX, RY << 3)	Register shift addressed halfword store instruction.
STR.W	O	×	STR32.W RZ, (RX, RY << 0) STR32.W RZ, (RX, RY << 1) STR32.W RZ, (RX, RY << 2) STR32.W RZ, (RX, RY << 3)	Register shift addressed word store instruction.
SUBC	OO SUBC32 RZ,	RX, RY SUBC16 RZ, RY		Unsigned subtract with borrow instruction.
SUBI	OO SUBI32 RZ,	RX, OIMM12 SUBI16 RZ, OIMM8		Unsigned immediate subtraction instruction.
SUBU	OO SUBU32 RZ,	RX, RY SUBU16 RZ, RY		Unsigned subtraction instruction.
SYNC	O	×	SYNC32 IMM5	CPU synchronization instruction.
TRAP	O	×	TRAP32 0 TRAP32 1 TRAP32 2 TRAP32 3	Unconditional Operating System Trap instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format OO			Instruction Description
TST	TST32 RX, RY		TST16 RX, RY	Zero test instructions.
TZ	OO TSTNBZ32 RX		TSTNBZ16 RX	No byte equals zero register Test instructions.
WAIT	O	x	WAIT32	Enter low power wait mode instruction.
XOR	OO XOR32 RZ, RX, RY		XOR16 RZ, RX	Bitwise XOR instruction.
XORI	O	x	XORI32 RZ, RX, IMM12 Bitwise exclusive OR instruction for immediate data.	
XSR	O	x	XSR32 RZ, RX, OIMM5 Extended right shift instruction.	
XTRB0	O	x	XTRB0.32 RZ, RX	There is no sign-extending instruction to extract byte 0.
XTRB1	O	x	XTRB1.32 RZ, RX	Extract byte 1 without sign-extending instruction.
XTRB2	O	x	XTRB2.32 RZ, RX	Extract byte 2 without sign-extending instruction.
XTRB3	O	x	XTRB3.32 RZ, RX	Extract byte 3 without sign-extending the instruction.
ZEXT	O	x	ZEXT32 RZ, RX, MSB, LSB Bit extract and no sign extension instruction.	
ZEXTB	OO ZEXTB32 RZ, RX		ZEXTB16 RZ, RX	There is no sign-extend instruction for byte extraction.
ZEXTH	OO ZEXTH32 RZ, RX		ZEXTH16 RZ, RX	There is no sign-extend instruction for halfword fetches.
DSP instructions				
PADD.8	O	x	PADD.8 RZ, RX, RY 16-bit	8-bit parallel addition instruction.
PADD.16	O	x	PADD.16 RZ, RX, RY PADD Parallel addition instruction.	
PADD.(U/S)8.S	O	x	(U/S)8.S RZ, RX, RY 8-bit parallel with saturation operation (no/yes)	Signed addition instruction.
PADD.(U/S)16.SO		x	PADD.(U/S)16.S RZ, RX, RY	16-bit parallelism with saturation (no/yes) Signed addition instruction.
ADD.(U/S)32.S	O	x	ADD.(U/S)32.S RZ, RX, RY 32-bit (without/with) saturation operation	Number addition instruction.
PSUB.8	O	x	PSUB.8 RZ, RX, RY 8-bit parallel subtraction instruction.	
PSUB.16	O	x	PSUB.16 RZ, RX, RY PSUB.	16-bit parallel subtraction instruction.
PSUB.(U/S)8.S	O	x	(U/S)8.S RZ, RX, RY 8-bit parallel operation with saturation (no/yes)	Signed subtraction instruction.
PSUB.(U/S)16.S	O	x	PSUB.(U/S)16.S RZ, RX, RY 16-bit parallel with saturation operation (no/yes)	Signed subtraction instruction.
SUB(U/S)32.S	O	x	SUB(U/S)32.S RZ, RX, RY 32-bit (without/with) saturation operation	Subtraction instruction.
PADDH.(U/S)8	O	x	PADDH.(U/S)8 RZ, RX, RY 8-bit parallel	(with/without sign) addition Mean instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly	32-bit 16-bit assembly format		Instruction Description
instruction PADDH.(U/S)16 O		×	PADDH.(U/S)16 RZ, RX, RY 16-bit parallel (unsigned/signed) addition and averaging Mean instruction.
ADDH.(U/S)32	O	×	ADDH.(U/S)32 RZ, RX, RY 32-bit (unsigned/signed) addition to find the average value instruction.
PSUBH.(U/S)8	O	×	PSUBH.(U/S)8 RZ, RX, RY 8-bit parallel (unsigned/signed) subtraction Mean instruction.
PSUBH.(U/S)16	O	×	PSUBH.(U/S)16 RZ, RX, RY 16-bit parallel (unsigned/signed) subtraction Mean instruction.
SUBH.(U/S)32	O	×	SUBH.(U/S)32 RZ, RX, RY 32-bit (unsigned/signed) subtraction average instruction.
PASX.16	O	×	PASX.16 RZ, RX, RY 16-bit parallel cross addition and subtraction instructions.
PSAX.16	O	×	PSAX.16 RZ, RX, RY 16-bit parallel cross add/subtract instruction.
PASX.(U/S)16.S	O	×	PASX.(U/S)16.S RZ, RX, RY 16-bit parallel with saturation operation (no/yes) Sign cross addition and subtraction instructions.
PSAX.(U/S)16.S	O	×	PSAX.(U/S)16.S RZ, RX, RY 16-bit parallel with saturation operation (no/yes) Sign cross subtraction and addition instructions.
PASXH.(U/S)16	O	×	PASXH.(U/S)16 RZ, RX, RY 16-bit parallel (without/with) sign cross add/subtract Average command.
PSAXH.(U/S)16	O	×	PSAXH.(U/S)16 RZ, RX, RY 16-bit parallel (with/without sign) cross add/subtract Average command.
ADD.64	O	×	ADD.64 RZ, RX, RY 64-bit addition instruction.
ADD.(U/S)64.S	O	×	ADD.(U/S)64.S RZ, RX, RY 64-bit (without/with) register with saturation operation Number addition instruction.
SUB.64	O	×	SUB.64 RZ, RX, RY SUB.(U/ S)64.S RZ, RX, RY 64-bit (without/with) saturation operation Subtraction instruction.
ASRI.S32.R	O	×	ASRI.S32.R RZ,RX,OIMM5 Immediate arithmetic shift right with rounding make.
ASR.S32.R	O	×	ASR.S32.R RZ, RX, RY Arithmetic right shift with rounding.
LSRI.U32.R	O	×	LSRI.U32.R RZ,RX,OIMM5 Immediate logical right shift with rounding make.
LSR.U32.R	O	×	LSR.U32.R RZ, RX, RY Logical right shift with rounding.
LSLI.(U/S)32.S	O	×	LSLI.(U/S)32.S RZ,RX,OIMM5 Immediate value with saturation operation (no/yes) Logical left shift instruction.
LSL.(U/S)32.S	O	×	LSL.(U/S)32.S RZ, RX, RY (Without/With) Signed Logic with Saturation Operation Shift left instruction.
PASRI.S16	O	×	PASRI.S16 RZ,RX,OIMM4 16-bit parallel immediate arithmetic shift right instruction.
PASR.S16	O	×	PASR.S16 RZ, RX, RY 16-bit parallel arithmetic right shift instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit	16-bit assembly	format	Instruction Description
PASRI.S16.R	O	x	PASRI.S16.R RZ,RX,OIMM4	16-bit parallel immediate with rounding Arithmetic right shift instruction.
PASR.S16.R	O	x	PASR.S16.R RZ, RX, RY 16-bit parallel	arithmetic right with rounding Move instruction
PLSRI.U16	O	x	PLSRI.U16 RZ,RX,OIMM4 16-bit Parallel	Immediate Logical Shift Right instruction.
PLSR.U16	O	x	PLSR.U16 RZ, RX, RY 16-bit	16-bit parallel logical right shift instruction.
PLSRI.U16.R	O	x	PLSRI.U16.R parallel immediate value with rounding RZ,RX,OIMM4	Logical right shift instruction.
PLSR.U16.R	O	x	PLSR.U16.R RZ, RX, RY 16-bit parallel	logic right with rounding Move instruction.
PLSLI.16	O	x	PLSLI.16 RZ,RX,OIMM4 16-bit	16-bit parallel immediate logical left shift instruction.
PLSL.16	O	x	PLSL.16 RZ, RX, RY PLSLI.	parallel logical shift left instruction.
PLSL.(U/S)16.S	O	x	(U/S)16.S RZ,RX,OIMM4	16-bit parallel immediate with saturation (Unsigned/Yes) Signed logical left shift instruction.
PLSL.(U/S)16.S	O	x	PLSL.(U/S)16.S RZ, RX, RY 16-bit parallel	with saturation operation (no/yes) Signed logical left shift instruction.
SEL	O	x	SEL RZ, RX, RY,RS	Bit select instruction.
PCMPNE.8	O	x	PCMPNE.8 RZ, RX, RY	8-bit parallel compare-not-equal instruction.
PCMPNE.16	O	x	PCMPNE.16 RZ, RX, RY PCMPHS.	16-bit parallel compare-not-equal instruction.
PCMPHS.(U/S)8 O		x	(U/S)8 RZ, RX, RY	8-bit parallel (no/yes) signed greater than or equal to Comparison instructions.
PCMPHS.(U/S)16 O		x	PCMPHS.(U/S)16 RZ, RX, RY	16-bit parallel (unsigned/signed) greater than or equal to Comparison instructions.
PCMPLT.(U/S)8 O		x	PCMPLT.(U/S)8 RZ, RX, RY	8-bit parallel (no/yes) signed less than comparison Select a command.
PCMPLT.(U/S)16 O		x	PCMPLT.(U/S)16 RZ, RX, RY	16-bit parallel (no/yes) signed less than comparison Select a command.
PMAX.(U/S)8	O	x	PMAX.(U/S)8 RZ, RX, RY 8-bit parallel	(without/with) Signed large value make.
PMAX.(U/S)16	O	x	PMAX.(U/S)16 RZ, RX, RY 16-bit parallel	(without/with) Signed larger value make.
MAX.(U/S)32 PMIN.	O	x	MAX.(U/S)32 RZ, RX, RY 32-bit (without/with)	signed larger value instruction.
(U/S)8	O	x	PMIN.(U/S)8 RZ, RX, RY 8-bit parallel (without/with)	Small value of sign make.
PMIN.(U/S)16	O	x	PMIN.(U/S)16 RZ, RX, RY 16-bit parallel	(no/yes) signed small value make.
MIN.(U/S)32	O	x	MIN.(U/S)32 RZ, RX, RY 32-bit (without/with)	signed minimum value instruction.
DEXTI	O	x	DEXTI RZ, RX, RY,IMM5	Immediate digital intercept instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit	16-bit assembly	format	Instruction Description
DEXT	O	x DEXT	RZ, RX, RY,RS	Word interception instruction.
PKG	O	x PKG	RZ, IMM4A,RY,OIMM4B	Immediate shift packed instructions.
PKGLL	O	x	PKGLL RZ, RX, RY x	Low halfword packing instruction.
PKG	O	PKGHH	RZ, RX, RY	Upper halfword packing instruction.
PEXT.U8.E	O	x	PEXT.U8.E RZ,RX	8-bit parallel unsigned-extend instruction.
PEXT.S8.E	O	x	PEXT.S8.E RZ,RX	8-bit parallel sign-extend instruction.
PEXTX.U8.E	O	x	PEXTX.U8.E RZ,RX	8-bit parallel unsigned cross-extend instruction.
PEXTX.S8.E	O	x	PEXTX.S8.E RZ,RX NARL	8-bit parallel signed interleave extend instruction.
NARL	O	x	RZ, RX, RY x NARH RZ,	Intercept assembly instructions at low level.
NARH	O	RX, RY		High-order interception and splicing instructions.
NARLX	O	x	NARLX RZ, RX, RY	Low-level cross-interception assembly instructions.
NARHX	O	x NARHX	RZ, RX, RY	High-order cross-interception and splicing instructions.
CLIP.(U/S)32	O	x	CLIP.(U/S)32 RX,IMM5	Immediate value (no/yes) Sign clipping saturation pointer make.
CLIP.(U/S)32	O	x	CLIP.(U/S)32 RZ, RX, RY (No/Yes)	Signed clipping and saturation instruction.
PCLIP.(U/S)16	O	x	PCLIP.(U/S)16 RZ,RX,IMM4	16-bit parallel immediate value (no/yes) sign cut instruction.
PCLIP.(U/S)16	O	x	PCLIP.(U/S)16 RZ, RX, RY 16-bit parallel	(without/with) sign clipping saturation instruction.
PABS.S8.S	O	x	PABS.S8.S RZ,RX	8-bit parallel absolute value pointer with saturation make.
PABS.S16.S	O	x	PABS.S16.SRZ,RX	16-bit parallel absolute value with saturation instruction.
ABS.S32.S	O	x	ABS.S32.S RZ,RX	32-bit absolute value instruction with saturation.
PNEG.S8.S	O	x	PNEG.S8.S RZ,RX	8-bit parallel inverted instruction with saturation make.
PNEGS16.S	O	x	PNEGS16.S RZ,RX	16-bit parallel inverted instruction with saturation make.
NEG.S32.S	O	x	NEG.S32.S RZ,RX 8-bit	32-bit negate instruction with saturation.
DUP.8	O	x	DUP.8 RZ, RX, INDEX 16-bit	and copy instruction.
DUP.16	O	x	DUP.16 RZ, RX, INDEX MULS.16	and copy instruction.
MULS.(U/S)32 MULA.	O	x	U)32 RZ, RX, RY 32-bit (without/with) signed multiply-accumulate-subtract instruction.	
(U/S)32.SO		x	MULA.(S/U)32.S RZ, RX, RY	32-bit (without/with) symbol with saturation Multiply and accumulate instructions.
MULS.(U/S)32.SO		x	MULS.(S/U)32.S RZ, RX, RY	32-bit (without/with) symbol with saturation Multiply, accumulate and subtract instructions.
MUL.S32.H	O	x	MUL.S32.H RZ, RX, RY	32-bit signed multiply extract high 32-bit instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly	format	Instruction Description
MUL.S32.RH	O	×	MUL.S32.RH RZ, RX, RY 32-bit signed multiplication with rounding Fetch the high 32 bits of the instruction.
RMUL.S32.H	O	×	RMUL.S32.H RZ, RX, RY 32-bit signed fractional multiplication, take the high 32 bits instruction.
RMUL.S32.RH	O	×	RMUL.S32.RH RZ, RX, RY 32-bit signed fractional number with rounding Multiplication takes the upper 32 bits of the instruction.
MULA.S32.HS	O	×	MULA.S32.HS RZ, RX, RY 32-bit signed multiply-accumulate with saturation Add the high 32 bits of the instruction.
MULS.S32.HS	O	×	MULS.S32.HS RZ, RX, RY 32-bit signed multiply-accumulate with saturation Subtract upper 32 bits instruction.
MULA.S32.RHS	O	×	MULA.S32.RHS RZ, RX, RY 32-bit signed integer with rounding and saturation The multiplication and accumulation instruction takes the high 32 bits.
MULS.S32.RHS	O	×	MULS.S32.RHS RZ, RX, RY 32-bit signed integer with rounding and saturation The multiplication and accumulation and subtraction instructions are used to extract the upper 32 bits of the numbers.
MULXL.S32	O	×	MULXL.S32 RZ, RX, RY 32-bit signed lower halfword unaligned multiplication make.
MULXL.S32.R	O	×	MULXL.S32.R RZ, RX, RY 32-bit signed low half with rounding Word-unaligned multiply instruction.
MULXH.S32	O	×	MULXH.S32 RZ, RX, RY 32-bit signed upper halfword unaligned multiplication make. Word-unaligned multiply instruction.
MULXH.S32.R	O	×	MULXH.S32.R RZ, RX, RY 32-bit signed high half with rounding Word-unaligned multiply instruction.
RMULXL.S32	O	×	RMULXL.S32 RZ, RX, RY 32-bit signed lower halfword unaligned fractional multiplication instruction. Word-unaligned fractional multiply instruction.
RMULXL.S32.R	O	×	RMULXL.S32.R RZ, RX, RY 32-bit signed low half with rounding Word-unaligned fractional multiply instruction.
RMULXH.S32	O	×	RMULXH.S32 RZ, RX, RY 32-bit signed upper halfword unaligned fractional multiplication instruction. Word-unaligned fractional multiply instruction.
RMULXH.S32.R	O	×	RMULXH.S32.R RZ, RX, RY 32-bit signed high half with rounding Word-unaligned fractional multiply instruction.
MULAXL.S32.S	O	×	MULAXL.S32.S RZ, RX, RY 32-bit signed low half with saturation Word-unaligned multiply-accumulate instructions.
MULAXL.S32.RS O		×	MULAXL.S32.RS RZ, RX, RY 32-bit signed integer with rounding and saturation The lower halfword is not aligned to the multiply-accumulate instruction.
MULAXH.S32.S	O	×	MULAXH.S32.S RZ, RX, RY 32-bit signed high half with saturation Word-unaligned multiply-accumulate instructions.
MULAXH.S32.RS O		×	MULAXH.S32.RS RZ, RX, RY 32-bit signed integer with rounding and saturation The upper halfword of the multiply-accumulate instruction is not aligned.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly	format	Instruction Description
MULLL.S16	O	×	MULLL.S16 RZ, RX, RY MULHH.S16 16-bit signed low halfword multiply instruction.
MULHH.S16	O	×	RZ, RX, RY 16-bit signed high halfword multiplication instruction.
MULHL.S16	O	×	MULHL.S16 RZ, RX, RY RMULLL.S16 16-bit signed high and low halfword multiply instruction.
RMULLL.S16	O	×	RZ, RX, RY 16-bit signed low halfword fractional multiplication instruction make.
RMULHH.S16	O	×	RMULHH.S16 RZ, RX, RY 16-bit signed upper halfword fractional multiplication instruction make.
RMULHL.S16	O	×	RMULHL.S16 RZ, RX, RY 16-bit signed high and low half-word fractional multiplication instruction make.
MULAHH.S16.S	O	×	MULAHH.S16.S RZ, RX, RY 16-bit signed high half with saturation operation Word multiply-accumulate instructions.
MULAHL.S16.S	O	×	MULAHL.S16.S RZ, RX, RY 16-bit signed high and low with saturation operation Halfword multiply-accumulate instruction.
MULALL.S16.E	O	×	MULALL.S16.E RZ, RX, RY 16-bit signed lower half with extended operation Word multiply-accumulate instructions.
MULAHH.S16.E	O	×	MULAHH.S16.E RZ, RX, RY 16-bit signed high half with extended operation Word multiply-accumulate instructions.
MULAHL.S16.E	O	×	MULAHL.S16.E RZ, RX, RY 16-bit signed high and low with extended operation Halfword multiply-accumulate instruction.
PMUL.(U/S)16	O	×	PMUL.(S/U)16 RZ, RX, RY 16-bit parallel (without/with) signed multiplication make.
PMULX.(U/S)16 O		×	PMULX.(S/U)16 RZ, RX, RY 16-bit parallel (no/yes) signed cross-multiplication make.
PRMUL.S16	O	×	PRMUL.S16 RZ, RX, RY 16-bit parallel signed fractional multiply instructions.
PRMULX.(U/S)16 O		×	PRMULX.S16 RZ, RX, RY 16-bit parallel (without/with) sign interleaved decimal Multiplication instruction.
PRMUL.S16.H	O	×	PRMUL.S16.H RZ, RX, RY 16-bit parallel signed fractional multiplication high 16-bit instructions
PRMUL.S16.RH	O	×	PRMUL.S16.RH RZ, RX, RY 16-bit parallel signed with rounding Fractional multiplication takes the high 16 bits instruction.
PRMULX.S16.H	O	×	PRMULX.S16.H RZ, RX, RY 16-bit parallel signed interleaved fractional multiplication High 16 bits of instruction.
PRMULX.S16.RH O		×	PRMULX.S16.RH RZ, RX, RY 16-bit parallel signed with rounding Cross-fractional multiplication takes the upper 16 bits instruction.
MULCA.S16.S	O	×	MULCA.S16.S RZ, RX, RY 16-bit signed multiply chain with saturation Add instructions.
MULCAX.S16.S	O	×	MULCAX.S16.S RZ, RX, RY 16-bit signed interleaved with saturation Multiply-chain-add instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit 16-bit assembly format		Instruction Description
MULCS.S16	O	×	MULCS.S16 RZ, RX, RY 16-bit signed multiply-chain-subtract instruction.
MULCSR.S16	O	×	MULCSR.S16 RZ, RX, RY 16-bit signed reverse multiply chain subtract instruction.
MULCSX.S16	O	×	MULCSX.S16 RZ, RX, RY 16-bit signed cross-multiply chain-subtract instruction.
MULACA.S16.S	O	×	MULACA.S16.S RZ, RX, RY 16-bit signed multiply chain with saturation Add and accumulate instructions.
MULACAX.S16.SO		×	MULACAX.S16.S RZ, RX, RY 16-bit signed interleaved with saturation Multiply-chain-add-accumulate instruction.
MULACS.S16.S	O	×	MULACS.S16.S RZ, RX, RY 16-bit signed multiply chain with saturation Subtract and accumulate instructions.
MULACSR.S16.SO		×	MULACSR.S16.S RZ, RX, RY 16-bit signed reverse with saturation Multiply-chain subtract-accumulate instructions.
MULACSX.S16.SO		×	MULACSX.S16.S RZ, RX, RY 16-bit signed interleaved with saturation Multiply-chain subtract-accumulate instructions.
MULSCA.S16.S	O	×	MULSCA.S16.S RZ, RX, RY 16-bit signed multiply chain with saturation Add and subtract instructions.
MULSCAX.S16.SO		×	MULSCAX.S16.S RZ, RX, RY 16-bit signed interleaved with saturation Multiply-chain add-accumulate instruction.
MULACA.S16.E	O	×	MULACA.S16.E RZ, RX, RY 16-bit signed multiply chain with extend operation Add and accumulate instructions.
MULACAX.S16.EO		×	MULACAX.S16.E RZ, RX, RY 16-bit signed interleave with extend operation Multiply-chain-add-accumulate instruction.
MULACS.S16.E	O	×	MULACS.S16.E RZ, RX, RY 16-bit signed multiply chain with extend operation Subtract and accumulate instructions.
MULACSR.S16.EO		×	MULACSR.S16.E RZ, RX, RY 16-bit signed reverse with extend operation Multiply-chain subtract-accumulate instructions.
MULACSRX.S16.EO		×	MULACSRX.S16.E RZ, RX, RY 16-bit signed interleave with extend operation Multiply-chain subtract-accumulate instructions.
MULSCA.S16.E	O	×	MULSCA.S16.E RZ, RX, RY 16-bit signed multiply chain with extend operation Addition and subtraction instructions
MULSCAX.S16.EO		×	MULSCAX.S16.E RZ, RX, RY 16-bit signed interleave with extend operation Multiply-chain add-accumulate instruction.
PSABSA.U8	O	×	PSABSA.U8 RZ, RX, RY 8-bit parallel unsigned subtraction absolute value chain addition instruction.
PSABSAA.U8	O	×	PSABSAA.U8 RZ, RX, RY 8-bit parallel unsigned subtraction absolute value chain addition Accumulate instruction.
DIVSL	O	×	DIVUL RZ, RX, RY Unsigned long division instruction.
DIVUL	O	×	DIVSL RZ, RX, RY long division instruction.
MULACA.S8	O	×	MULACA.S8 RZ, RX, RY 8-bit parallel signed multiply chain add accumulate instruction.

Continue on next page

Table 6.34 – Continued from previous page

Assembly instructions	32-bit	16-bit assembly format	Instruction Description
BLOOP	O	x	BLOOP RX, LABEL1, LA-BEL2
LDBI.W	O	x	LDBI.W RZ, (RX)
LDBI.H	O	x	LDBI.HRZ, (RX)
LDBI.HS	O	x	LDBI.HS RZ, (RX)
LDBI.B	O	x	LDBI.B RZ, (RX)
LDBI.BS	O	x	LDBI.BS RZ, (RX)
PLDBI.D	O	x	PLDBI.D RZ, (RX)
STBI.W	O	x	STBI.W RZ, (RX)
STBI.H	O	x	STBI.HRZ, (RX)
STBI.B	O	x	STBI.B RZ, (RX)
LDBIR.W	O	x	LDBIR.W RZ, (RX), RY Register address increment word load instruction
LDBIR.H	O	x	LDBIR.H RZ, (RX), RY Register address increment unsigned half-word addition Load instructions.
LDBIR.HS	O	x	LDBIR.HS RZ, (RX), RY Register address auto-increment signed half-word addition Load instructions.
LDBIR.B	O	x	LDBIR.B RZ, (RX), RY Unsigned byte increment of register address Load instructions.
LDBIR.BS	O	x	LDBIR.BS RZ, (RX), RY Register address auto-increment signed byte addition Load instructions.
PLDBIR.D	O	x	PLDBIR.D RZ, (RX) , RY register address auto-increment double word load pointer command.
STBIR.W	O	x	STBIR.W RZ, (RX) , RY Word store instruction with register address incremented.
STBIR.H	O	x	STBIR.H RZ, (RX) , RY Register address increments half word storage instruction make.
STBIR.B	O	x	STBIR.B RZ, (RX), RY register address increments by byte command.

Note: O indicates that the instruction exists in the corresponding instruction set, and x indicates that the instruction does not exist in the corresponding instruction set. * indicates DSP extension instructions.

6.5 Instruction Execution Delay

Table 6.35: Instruction execution delay table

Instruction	Execution cycle notes			
type Basic instruction				
Addition and subtraction instructions				
ADDU32/16	1	-		
ADDC32/16	1	-		
ADDI32/16	1	-		
SUBU32/16	1	-		
SUBC32/16	1	-		
SUBI32/16	1	-		
RSUB32	1	-		
IXH32	1	-		
IXW32	1	-		
IXD32	1	-		
INCF32	1	-		
INCT32	1	-		
DECFF32	1	-		
DECT32	1	-		
DEC GT32	1	-		
DECLT32	1	-		
DECNE32	1	-		
Logical operation instructions				
AND32/16	1	-		
ANDI32/16	1	-		
ANDN32	1	-		
ANDNI32	1	-		
OR32/16	1	-		
ORI32	1	-		
XOR32	1	-		
XORI32	1	-		
NOR32/16	1	-		
NOT32/16	1	-		
shift instructions				
LSL32/16	1	-		
LSLI32/16	1	-		
LSLC32	1	-		
LSR32/16	1	-		

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
LSRI32/16	1	-
LSRC32	1	-
ASR32/16	1	-
ASRI32/16	1	-
ASRC32	1	-
ROTL32/16	1	-
ROTLI32	1	-
XSR32	1	-
Comparison Instructions		
CMPNE32/16	1	-
CMPNEI32/16	1	-
CMPHS32/16	1	-
CMPHSI32/16	1	-
CMPLT32/16	1	-
CMPLTI32/16	1	-
TST32/16	1	-
TSTNBZ32/16 data	1	-
transfer instructions		
MOV32/16	1	-
MOVF32	1	-
MOVT32	1	-
MOVI32/16	1	-
MOVIH32	1	-
MVCV32/16	1	-
MVC32	1	-
CLRF32	1	-
CLRT32	1	-
LRW32/16*		-
GRS32	1	-
Bit operation instructions		
BCLRI32/16	1	-
BSETI32/16	1	-
BTSTI32/16	1	-
Extract Insert Instruction		
ZEXT32/16	1	-
SEXT32/16	1	-
INS32	1	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
ZEXTB32	1	-
ZEXTH32	1	-
SEXTB32/16	1	-
SEXTH32/16	1	-
XTRB0.32	1	-
XTRB1.32	1	-
XTRB2.32	1	-
XTRB3.32	1	-
BREV32	1	-
REV32/16	1	-
REVH32/16	1	-
multiplication and division instructions		
MULT32/16 1/3		1 cycle when configured for DSP or floating point; 3 cycles in basic configuration
MULLLL.S16(MULSH) 1 MUL.		-
(U/S)32 1/2/5		1 cycle when DSP is configured; 2 cycles when floating point is configured; basic configuration The following is 5 cycles
MULA.32.L	1/2/4	1 cycle when configuring DSP; 2 cycles when configuring floating point; basic configuration The following is 4 cycles
MULA.(U/S)32	1/2/5	1 cycle when DSP is configured; 2 cycles when floating point is configured; basic configuration The following is 4 cycles
MULALL.S16.S	1-4	1 cycle when DSP is configured; 2 cycles when floating point is configured; basic configuration The following is 2 cycles
DIVU32	5-36	-
DIVS32	5-36	-
Miscellaneous operation instructions		
ABS32	1	-
FF0.32	1	-
FF1.32	1	-
BMASKI32	1	-
BGENR32	1	-
BGENI32	1	-
Branch Instructions		
BT32/16	1	-
BF32/16	1	-
BEZ32	1	-
BNEZ32	1	-
BHZ32	1	-
BLSZ32	1	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
BLZ32	1	-
BHSZ32	1	-
Jump Instructions		
BR32/16	1	-
BSR32	1	-
JMPI32*	3	Split into LRW and JMP instructions
JSRI32*	3	Split into LRW and JSR instructions
JMP32/16	1	-
JSR32/16	1	-
RTS32/16	1	-
Immediate offset access instructions		
LD32/16.B*	1	-
LD32.BS*	1	-
LD32/16.H*	1	-
LD32.HS*	1	-
LD32/16.W*	1	-
ST32/16.B*	1	-
ST32/16.H*	1	-
ST32/16.W*	1	-
Register offset access instructions		
LDR32.B*	3	-
LDR32.BS*	3	-
LDR32.H*	3	-
LDR32.HS*	3	-
LDR32.W*	3	-
STR32.B*	3	-
STR32.H*	3	-
STR32.W*	3	-
Multiple register access instructions		
LDQ32*	4	Split into 4 LD.W instructions.
LDM32*	N	Split into N LD.W instructions.
STQ32*	4	Split into 4 ST.W instructions.
STM32*	N	Split into N ST.W instructions.
PUSH32/16*	1+N	Split into SUB and N ST.W instructions.
POP32/16*	2+N	Split into N LD.W, ADD and RTS instructions.
IPOP16*	7	Split into 7 atomic instructions
IPUSH16*	7	Split into 7 atomic instructions

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
NIE16*	5	Split into 5 atomic instructions
NIR16*	5	Split into 5 atomic instructions
Symbolic access instructions		
LRS32.B*	1	-
LRS32.H*	1	-
LRS32.W*	1	-
SRS32.B*	1	-
SRS32.H*	1	-
SRS32.W*	1	-
Control register operation instructions		
MFCR32	1	-
MTCR32	1	-
PSRSET32	1	-
PSRCLR32	1	-
Low power instructions		
WAIT32	1	-
DOZE32	1	-
STOP32	1	-
Exception return instruction		
RTE32	1	-
Special function instructions		
SYNC32	1	-
BKPT32	1	-
SCE32	1	-
IDLY32	1	-
TRAP32	1	-
DSP instructions		
Addition and subtraction instructions		
PADD.8	1	-
PADD.16	1	-
PADD.(U/S)8.S	1	-
PADD.(U/S)16.S	1	-
ADD.(U/S)32.S	1	-
PSUB.8	1	-
PSUB.16	1	-
PSUB.(U/S)8.S	1	-
PSUB.(U/S)16.S	1	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction	Execution cycle	notes
type SUB(U/	1	-
S)32.S PADDH.(U/	1	-
S)8 PADDH.(U/S)16	1	-
ADDH.(U/S)32	1	-
PSUBH.(U/S)8	1	-
PSUBH.(U/S)16	1	-
SUBH.(U/S)32	1	-
PASX.16	1	-
PSAX.16	1	-
PASX.(U/S)16.S	1	-
PSAX.(U/S)16.S	1	-
PASXH.(U/S)16	1	-
PSAXH.(U/S)16	1	-
ADD.64	1	-
ADD.(U/S)64.S	1	-
SUB.64	1	-
SUB.(U/S)64.S	1	-
PABS.S8.S	1	-
PABS.S16.S	1	-
ABS.S32.S	1	-
PNEG.S8.S	1	-
PNEGS16.S	1	-
NEG.S32.S	1	-
PSABSA.U8	1	-
PSABSAA.U8	1	-
Shift Instructions		
ASRI.S32.R	1	-
ASR.S32.R	1	-
LSRI.U32.R	1	-
LSR.U32.R	1	-
LSLI.(U/S)32.S LSL.	1	-
(U/S)32.S	1	-
PASRI.S16	1	-
PASR.S16	1	-
PASRI.S16.R	1	-
PASR.S16.R	1	-
PLSRI.U16	1	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
PLSR.U16	1	-
PLSRI.U16.R	1	-
PLSR.U16.R	1	-
PLSLI.16	1	-
PLSL.16	1	-
PLSLI.(U/S)16.S PSL.	1	-
(U/S)16.S comparison	1	-
instruction		
PCMPNE.8	1	-
PCMPNE.16	1	-
PCMMPHS.(U/S)8	1	-
PCMMPHS.(U/S)16	1	-
PCMPLT.(U/S)8	1	-
PCMPLT.(U/S)16	1	-
PMAX.(U/S)8	1	-
PMAX.(U/S)16	1	-
MAX.(U/S)32	1	-
PMIN.(U/S)8	1	-
PMIN.(U/S)16	1	-
MIN.(U/S)32	1	-
Special function instructions		
DEXTI	1	-
DEXT	1	-
PKG	1	-
PKGLL	1	-
PKG	1	-
PEXT.U8.E	1	-
PEXT.S8.E	1	-
PEXTX.U8.E	1	-
PEXTX.S8.E	1	-
NARL	1	-
NARH	1	-
NARLX	1	-
NARHX	1	-
CLIP.(U/S)32	1	-
CLIP.(U/S)32	1	-
PCLIP.(U/S)16	1	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction	Execution cycle	notes
type PCLIP.(U/S)16	1	-
DUP.8	1	-
DUP.16	1	-
BLOOP*	1	-
SEL	1	-
Multiplication Instructions		
MULS.(U/S)32	2	-
MULA.(U/S)32.S	2	-
MULS.(U/S)32.S	2	-
MUL.S32.H	2	-
MUL.S32.RH	2	-
RMUL.S32.H	2	-
RMUL.S32.RH	2	-
MULA.S32.HS	2	-
MULS.S32.HS	2	-
MULA.S32.RHS	2	-
MULS.S32.RHS	2	-
MULXL.S32	2	-
MULXL.S32.R	2	-
MULXH.S32	2	-
MULXH.S32.R	2	-
RMULXL.S32	2	-
RMULXL.S32.R	2	-
RMULXH.S32	2	-
RMULXH.S32.R	2	-
MULAXL.S32.S	2	-
MULAXL.S32.RS	2	-
MULAXH.S32.S	2	-
MULAXH.S32.RS	2	-
MULLL.S16	2	-
MULHH.S16	2	-
MULHL.S16	2	-
RMULLL.S16	2	-
RMULHH.S16	2	-
RMULHL.S16	2	-
MULAHH.S16.S	2	-
MULAHL.S16.S	2	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
MULALL.S16.E	2	-
MULAHH.S16.E	2	-
MULAHL.S16.E	2	-
PMUL.(U/S)16	2	-
PMULX.(U/S)16	2	-
PRMUL.S16	2	-
PRMULX.(U/S)16	2	-
PRMUL.S16.H	2	-
PRMUL.S16.RH	2	-
PRMULX.S16.H	2	-
PRMULX.S16.RH	2	-
MULCA.S16.S	2	-
MULCAX.S16.S	2	-
MULCS.S16	2	-
MULCSR.S16	2	-
MULCSX.S16	2	-
MULACA.S16.S	2	-
MULACAX.S16.S	2	-
MULACS.S16.S	2	-
MULACSR.S16.S	2	-
MULACSX.S16.S	2	-
MULSCA.S16.S	2	-
MULSCAX.S16.S	2	-
MULACA.S16.E	2	-
MULACAX.S16.E	2	-
MULACS.S16.E	2	-
MULACSR.S16.E	2	-
MULACSRX.S16.E	2	-
MULSCA.S16.E	2	-
MULSCAX.S16.E	2	-
DIVUL	5-68	-
DIVSL	5-68	-
MULACA.S8	2	-
Memory access instructions		
LDBI.W*	1	-
LDBI.H*	1	-
LDBI.HS*	1	-

Continue on next page

Table 6.35 – Continued from previous page

Instruction Type	Execution cycle notes	
LDBI.B*	1	-
LDBI.BS*	1	-
PLDBI.D*	1	-
STBI.W*	1	-
STBI.H*	1	-
STBI.B*	1	-
LDBIR.W*	1	-
LDBIR.H*	1	-
LDBIR.HS*	1	-
LDBIR.B*	1	-
LDBIR.BS*	1	-
PLDBIR.D*	2	-
STBIR.W*	1	-
STBIR.H*	1	-
STBIR.B*	1	-

Note: * indicates memory access related instructions. The instruction completion cycle depends on the bus delay or cache hit situation. The values listed in the table are the maximum Quick situation.

Chapter 7 Memory Protection

7.1 Introduction to Memory Protection Unit

In a protected system, there are two main types of resources that need to be monitored: memory system and peripheral devices. The memory protection unit is responsible for the access legitimacy of the memory system (including peripheral devices) is checked. Its main functions include:

1. Determine whether the CPU has read/write access rights to the memory address in the current operating mode.
2. Obtain additional attributes of the access address, including security attributes, etc.

The memory protection unit supports N entries (N is 1-8 hardware configurable) and can set the access rights and attributes of N areas.

Table entries are identified and indexed by numbers from 0 to 7. The contents of the table entries of the memory protection unit are shown in Figure 7.1.

31	30	11	10	6	5	4	3	2	1	0
EN	Base Address		SIZE	S	NX	AP	-			

Figure 7.1: Memory Protection Unit Table Entry

in:

EN: Indicates whether the area is effective;

Base Address: indicates the starting address of the area;

Size: indicates the size of the area;

S: indicates the security attribute of the area;

NX: indicates the executability of the region fetch;

AP: Indicates the access permission of the area.

For the specific properties of each field, refer to the relevant system control register .

7.2 Related System Control Registers

7.2.1 Cache Configuration Register (CCR, CR<18,0>)

The cache configuration register is used to configure memory protection enable, big-endian mode, and the clock ratio of the system and processor.

Reset	31	17	16	15	11	10	8	7	6	5	4	3	2	1	0
		0		0		SCK		BE	0	0	0	0	0	0	MP
		0					0	0	0	0	0	0	0	0	0

Figure 7.2: Cache Configuration Registers

SCK - System and processor clock ratio:

This bit is used to indicate the clock ratio between the system and the processor. The calculation formula is: Clock Ratio = SCK + 1. There is a corresponding pin on the CPU. SCK is configured at power on reset and cannot be changed afterwards.

This domain currently has no functionality and is only used for software queries.

BE-Endian mode:

- When BE is 0, little endian;
- When BE is 1, big endian.

BE is configured during power on reset and cannot be changed afterwards. There are corresponding pins on the CPU.

MP-memory protection setting bits:

MP is used to set whether MPU is valid, as shown in [Table 7.1](#).

Table 7.1: E804 memory protection settings

MP Function
00 MPU is invalid.
01 MPU is valid.
10 MPU invalid.
11 MPU is valid.

7.2.2 Highly Bufferable Access Permission Configuration Register (CAPR, CR<19,0>)

The various bits of [CAPR](#) are shown in Figure 7.3.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
S7	S6	S5	S4	S3	S2	S1	S0	AP7	AP6	AP5	AP4	AP3	AP2	AP1	AP0	NX7	NX6	NX5	NX4	NX3	NX2	NX1	NX0									
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Figure 7.3: Highly configurable access rights configuration register

NX0~NX7-non-executable attribute setting bit:

When NX is 0, the area is executable;

When NX is 1, the area is non-executable.

Note: When the processor accesses a non-executable area, an access error exception occurs.

S0~S7-security attribute setting bits:

- When S is 0, the area is a non-safe area;
- When S is 1, the area is a safe area.

AP0~AP7-Access permission setting bits:

Table 7.2: Access permission settings

AP	Super User Permissions	Normal User Permissions
00	Not accessible	Not accessible
01	Read and write	Inaccessible
10	Read-write	Read-only
11	Read and Write	Read and Write

7.2.3 Protection Area Control Register (PACR, CR<20,0>)

Each bit of [PACR](#) is shown in Figure 7.4.

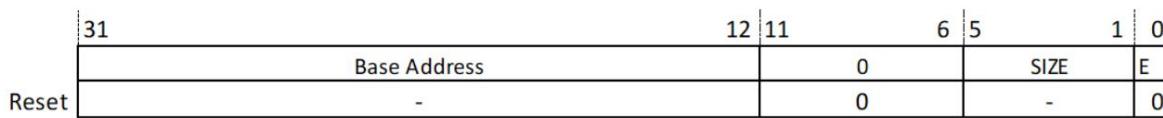


Figure 7.4: Protection zone control register

Base Address - The base address of the protected area:

- This register indicates the base address of the protected area. However, the written base address must be aligned with the set page size.

The minimum is 8M, CR<20,0>[22:12] must be 0, and the specific requirements of each page [are shown in Table 7.3](#).

- The size of the protected area ranges from 4KB to 4GB, which can be calculated using the formula: protected area size = $2^{(\text{Size}+1)}$.

The range is 01011 to 11111; other values may produce unpredictable results.

Table 7.3: Protection zone size configuration and its base address requirements

Size	The size of the protected area requires the base address
00000—01010 Reserved	-
01011	4KB No requirements
01100	8KB CR<20,0>.bit[12]=0
01101	16KB CR<20,0>.bit[13:12] = 0
01110	32KB CR<20,0>.bit[14:12] = 0
01111	64KB CR<20,0>.bit[15:12] = 0
10000	128KB CR<20,0>.bit[16:12] = 0
10001	256KB CR<20,0>.bit[17:12] = 0
10010	512KB CR<20,0>.bit[18:12] = 0
10011	1MB CR<20,0>.bit[19:12] = 0
10100	2MB CR<20,0>.bit[20:12] = 0
10101	4MB CR<20,0>.bit[21:12] = 0
10110	8MB CR<20,0>.bit[22:12] = 0
10111	16MB CR<20,0>.bit[23:12] = 0
11000	32MB CR<20,0>.bit[24:12] = 0
11001	64MB CR<20,0>.bit[25:12] = 0
11010	128MB CR<20,0>.bit[26:12] = 0
11011	256MB CR<20,0>.bit[27:12] = 0
11100	512MB CR<20,0>.bit[28:12] = 0
11101	1GB CR<20,0>.bit[29:12] = 0
11110	2GB CR<20,0>.bit[30:12] = 0
11111	4GB CR<20,0>.bit[31:12] = 0

E-protection zone effective settings:

- When E is 0, the protection zone is invalid;
- When E is 1, the protection zone is effective.

7.2.4 Protection Zone Select Register (PRSR, CR<21,0>)

PRSR is used to select the protection zone for the current operation. Its bits are shown in Figure 7.5:

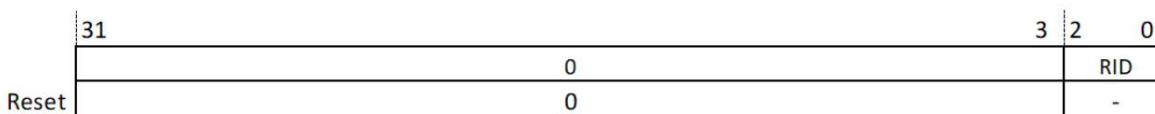


Figure 7.5: Protection zone selection register

RID - protection zone index value:

- RID indicates the corresponding protection zone selected, such as 000 indicates the 0th protection zone.

7.3 Memory Access Processing

The processing flow and results of the memory protection unit's CPU memory access are shown in Table 7.4 .

Table 7.4: Memory access processing flow and results

Memory Protection Unit Enable	Memory Protection Unit Hit	Processing results
Disable	Don't care	All accesses to memory are treated as: Both super user mode and normal user mode are readable and writable; Access attribute is not bufferable.
Enable	Hit Miss	The processor will generate an Access Error exception.
	Hit	If the access hits a protected area, the access rights and attributes of the protected area will prevail; If the access address hits multiple protection zones, the access rights of the protection zone with the highest index number are used. Attributes prevail. The access rights of the address are combined with the current operating mode of the processor to determine whether the access is legal. If the access complies with the access rights of the area, the memory protection unit allows the memory operation. And provide the access attributes of the access address; if the access request is not allowed, the memory protection unit will generate an access error exception and interrupt the execution of the processor.

Note: The access address may hit multiple protection zones, in which case the protection zone with the highest index number is considered to have hit.

Chapter 8 On-Chip Cache

8.1 Introduction to Cache

The E804 provides a hardware-configurable cache with the following key features:

- Cache size is hardware configurable, supporting 2KB/4KB/8KB/16KB;
- 4-way set associative, cache line size is 16 bytes;
- The maximum width of each access is 4 bytes, supporting byte/halfword/word access;
- Physical address index, physical address tag;
- The write strategy supports both write-through mode and write-back mode;
- The cache uses a first-in, first-out (FIFO) replacement policy;
- Supports invalidation and clearing operations for the entire cache, and supports invalidation and clearing operations for individual cache lines.

When the memory accesses the non-cacheable area, it directly bypasses the on-chip cache and accesses the off-chip memory through the bus, speeding up the access speed of the non-cacheable area. When the memory accesses the cacheable area, it first accesses the on-chip cache; in the case of a cache miss, it accesses the off-chip memory (the write access in write-direct mode also needs to be written back to the off-chip memory when the cache hits).

8.2 Related System Control Registers

Cache provides a set of 32-bit registers, including cache enable, cache line invalidation and cache area configuration.

The memory address space is shown in Table 8.1 .

Table 8.1: Cache Control Register Definition

address	Name	Type	Initial Value	Description
0xE000F000	CER	Read/write	0 Cache Enable Register.	
0xE000F004	CIR	Read/write	0 Cache Invalidate Register.	
0xE000F008	CRCR0	Read/write	0 No. 0 High Buffer Configuration Register.	
0xE000F00C	CRCR1	Read/write	0 1 High Buffer Configuration Register.	
0xE000F010	CRCR2	Read/write	0 2nd high buffer configuration register.	
0xE000F014	CRCR3	Read/write	0 No. 3 high buffer configuration register.	
0xE000F018~0xE000FFF3 -	-	-	-	reserve.
0xE000FFF4	CPFCR	Read/Write	0	Cache profiling control register.
0xE000FFF8	CPFATR	Read/write	0	Cache access count register.
0xE000FFFC	CPFMTR	Read/Write	0	Cache miss count register.

8.2.1 Cache Enable Register (CER)

The cache enable register is used to configure the cache enable and cache operating mode.

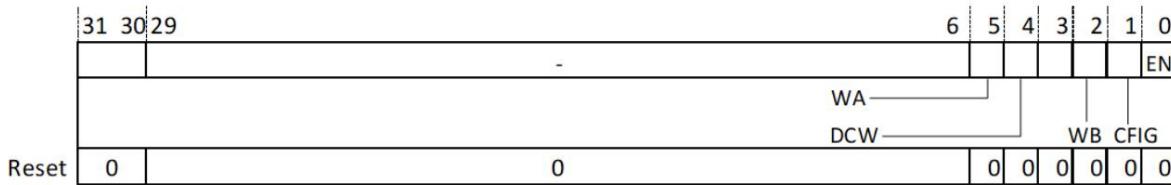


Figure 8.1: Cache Enable Register

WA Cache Write Allocate Valid Set Bit:

- 0: the cache is in write non-allocate mode;
- 1: The cache is in write allocate mode;
- The write allocate mode can only be used in the writeback mode; otherwise the results are unpredictable.

DCW-Cache writable attribute configuration bit:

- 0: cache is not writable;
- 1: Cache is writable.

WB-Cache write-back mode configuration bits:

- 0: cache is in write-through mode;
- 1: Cache is in write-back mode.

CFIG-Cache attribute configuration bits:

- 0: instruction and data cache;

- 1: Instruction cache.

EN-Cache enable bit:

- When EN is 0, the cache is off;
- When EN is 1, the cache is in working state.

8.2.2 Cache Invalidiation Register (CIR)

The cache invalidation register is used to control the clearing and invalidation of the cache, which can support operations on specific cache lines and the entire cache.

Two modes of operation.

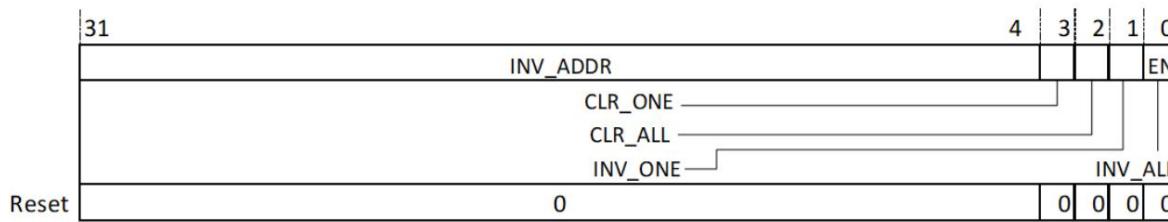


Figure 8.2: Cache Invalidation Register

INV_ADDR - cache line address:

- Identifies the cache line address that needs to be invalidated.

CLR_ONE - Single cache line clear set bit:

- When CLR_ONE is 1, the selected cache line, if marked as dirty, will be written back to the off-chip memory.

CLR_ALL - entire cache clear set bit:

- When CLR_ALL is 1, all cache lines marked as dirty will be written back to the off-chip memory.

INV_ONE - single cache line invalidation set bit:

- When INV_ONE is 1, invalidate the selected cache line.

INV_ALL - Entire cache invalidation bit set:

- When INV_ALL is 1, invalidate all cache lines in the cache.

Note: Any row operation cannot be performed simultaneously with any full cache operation. For example, if 32'b1111 is written to CIR, INV_ONE and CLR_ONE operations will be blocked.

8.2.3 High Cache Configuration Registers 0~3 (CRCR)

The cacheable area configuration register is used to configure the cacheable address space range. Four cacheable area configuration registers are provided to configure four different cacheable areas.

31	10:9	6:5	1:0
Base Address	Reserved	Size	EN

Figure 8.3: Cacheable Configuration Registers

Base Address-high buffer base address:

- The cache size can be configured from 4KB to 4GB. This field indicates the high bits of the cache address. For example, if you set the page size to 8M, CRCR[22:12] must be 0. For specific requirements of different sizes of each page, see Table 8.2 . Size - Buffer size:
 - The size of the buffer can be calculated by the formula: The size of the buffer can be calculated by the formula: $2^{\lceil \text{Size}+1 \rceil}$. Therefore, the range of Size that can be set is 01011
- Any other value will produce unpredictable results.

Table 8.2: Configurable cache sizes and their base address requirements

Size	Cache size requirements for base address
00000—01010 Reserved	-
01011	4KB No requirements
01100	8KB CRCR.bit[12]=0
01101	16KB CRCR.bit[13:12] = 0
01110	32KB CRCR.bit[14:12] = 0
01111	64KB CRCR.bit[15:12] = 0
10000	128KB CRCR.bit[16:12] = 0
10001	256KB CRCR.bit[17:12] = 0
10010	512KB CRCR.bit[18:12]=0
10011	1MB CRCR.bit[19:12]=0
10100	2MB CRCR.bit[20:12]=0
10101	4MB CRCR.bit[21:12]=0
10110	8MB CRCR.bit[22:12]=0
10111	16MB CRCR.bit[23:12]=0
11000	32MB CRCR.bit[24:12]=0
11001	64MB CRCR.bit[25:12]=0
11010	128MB CRCR.bit[26:12]=0
11011	256MB CRCR.bit[27:12]=0
11100	512MB CRCR.bit[28:12]=0
11101	1GB CRCR.bit[29:12]=0
11110	2GB CRCR.bit[30:12]=0
11111	4GB CRCR.bit[31:12]=0

EN-high buffer valid bit:

- When EN is 0, the high buffer area is invalid;
- When EN is 1, the high buffer area is valid.

8.2.4 Cache Performance Analysis Control Register (CPFCR)

The cache profiling control register is used to enable the cache profiling function (Cache Profiling) and reset the profiling related counters.

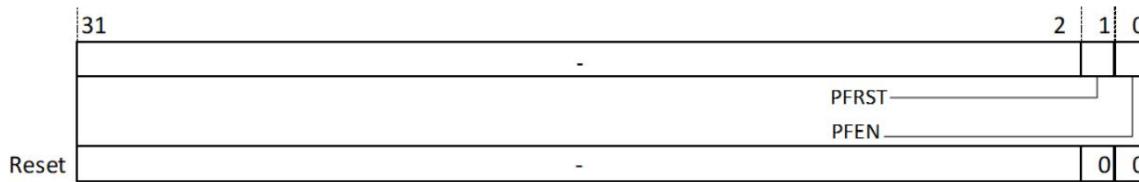


Figure 8.4: Cache Profiling Control Registers

PFRST-Cache profiling reset bit:

- When PERST is 1, CPFATR/CPFMTR is reset to 0.

PFEN-Cache profiling enable bit:

- When PFEN is 0, the cache performance analysis function is disabled;
- When PFEN is 1, the cache performance analysis function is enabled.

8.2.5 Cache Access Count Register (CPFATR)

The cache access count register is used to record the number of times the cache is accessed.



Figure 8.5: Cache access count register

Cache Access Times – Cache access times:

- Record the number of cache accesses. If cache performance analysis is enabled, each (256) cache access will cause the register number to be

The value increases by 1.

8.2.6 Cache Miss Count Register (CPFMTR)

The cache miss count register is used to record the number of misses in the access cache.

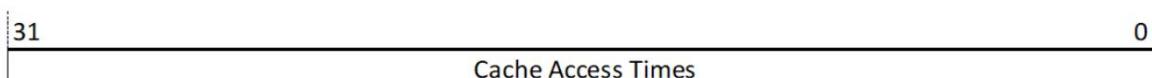


Figure 8.6: Cache Miss Count Register

Cache Miss Times – Number of cache misses:

- Records the number of cache misses. If cache profiling is enabled, each (256) cache miss will cause the register

The register value increases by 1.

Note: It is recommended that all cache configuration operations be performed with it turned off.

Chapter 9 Bus Matrix and Bus Interface

9.1 Introduction

E804 implements multiple bus interfaces, including system bus, instruction bus and data bus.

The bus matrix provides interconnection for the processor's internal requests to access the external bus interface. The connection relationship between the bus matrix and the CPU internal request and bus interface is shown in Figure 9.1. The bus matrix arbitrates the bus interface type according to the address of the memory access and distributes the processor's internal access to the system bus, instruction bus or data bus.

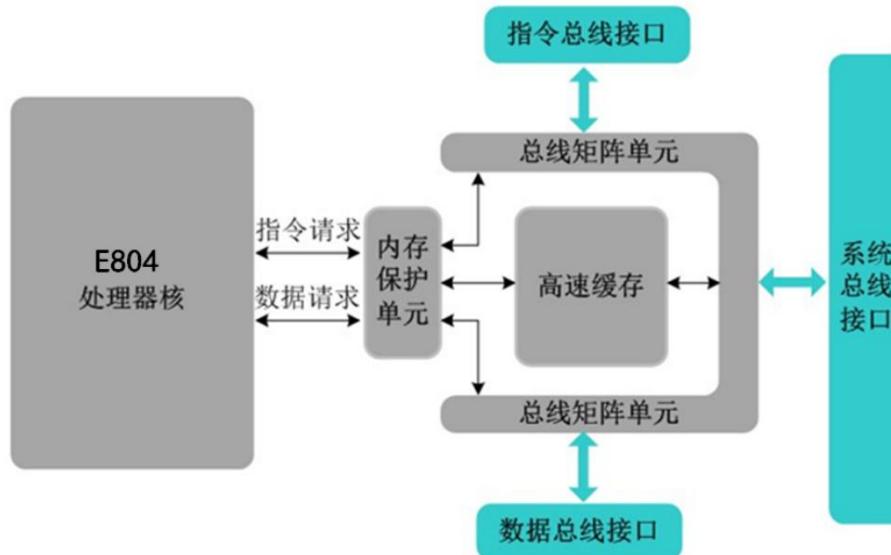


Figure 9.1: E804 bus matrix

The instruction fetch access and data access within the processor have the same bus access rights and can access all bus interfaces. In order to solve the problem of instruction fetch access and data access competing for the same bus interface in the same clock cycle, the bus matrix is also responsible for the priority judgment of the request. When the instruction fetch request and data request compete for the same bus interface, the data request has a higher priority.

The basic information and configurability of the E804 multi-bus interface are shown in Table 9.1 .

Table 9.1: Basic information and configurability of multi-bus interface

Bus interface configurability	Bus protocol timing mode
System bus is fixed and included, protocol can be configured with AHB direct output	
The system bus is fixed and the protocol can be configured with AHB-Lite for direct output	
The command bus is fixed with	AHB-Lite direct output
Data bus fixed included	AHB-Lite direct output

In addition, E804 supports pointing to the

The base address and space size of the instruction bus and data bus are hardware configurable. Among them, pad_bmu_xahbl_base specifies the base address and space size of the instruction bus and data bus.

pad_bmu_xahbl_base is a signal with a bit width of 20. The real base address it represents needs to expand the low bit 0 to pad_bmu_xahbl_mask specifies the address pairs in different address spaces.

The address space of the instruction bus or data bus can be configured from 4KB to 4GB. For example, if the address space size is set to 8M, pad_bmu_xahbl_base[10:0] must be 11'b0, pad_bmu_xahbl_mask[19:0] must be 20'b1111 1111 1000 0000

0000, see Table 9.2 for specific requirements of address spaces of different sizes .

Table 9.2: Base address and address alignment requirements for instruction bus and data bus

Address space size requirements for pad_bmu_xahbl_base	requirements for pad_bmu_xahbl_mask
4KB	No
8KB	requirement
16KB	bit[0] =1'b0 bit[1:0]
32KB	=2'b0 bit[2:0] =3'b0
64KB	bit[3:0] =4'b0
128KB	bit[4:0] =5'b0
256KB	bit[5:0] =6'b0
512KB	bit[6:0] =7'b0
1MB	bit[7:0] =8'b0
2MB	bit[8:0] =9'b0
4MB	bit[9:0] =10'b0
8MB	bit[10:0] =11'b0
16MB	bit[11:0] =12'b0
32MB	bit[12:0] =13'b0
64MB	bit[13:0] =14'b0
128MB	bit[14:0] =15'b0
256MB	bit[15:0] =16'b0
512MB	bit[16:0] =17'b0
1GB	bit[17:0] =18'b0
2GB	bit[18:0] =19'b0
4GB	bit[19:0] =20'b0

9.2 System Bus Interface

9.2.1 Features

The system bus interface of E804 can be configured to support AMBA2.0 AHB protocol (please refer to AMBA 2.0 specification - AMBA™ Specification Rev 2.0) or AMBA3.0 AHB-Lite protocol (please refer to AMBA 3.0 specification - AMBA3 AHB-Lite Protocol Specification Rev 1.0).

The basic characteristics of the system bus interface include:

- Supports AMBA2.0 AHB or AMBA3.0 AHB-Lite bus protocols, configurable by the user;
- Only supports direct output (non-Flop-out);
- The processor and system clock frequency ratio must be 1:1;

9.2.2 Contents of the Agreement

9.2.2.1 Supported transmission types

Considering the application field and cost of E804, the system bus interface only implements part of the AHB/AHB-Lite protocol.

Under the AHB-Lite protocol, as a master device, the bus interface supports the following transmission types:

- HBURST supports SINGLE and WRAP4 transfers, but not other burst types;
- HTRANS supports IDLE, NONSEQ and SEQ, but not other transfer types;
- HSIZE supports word, byte and half-word transfers, other transfer sizes are not supported;
- HWRITE supports both read and write operations.

Under the AHB protocol, as a master device, the bus interface supports the following transmission types:

- HBURST supports SINGLE and WRAP4 transfers, but not other burst types;
- HTRANS supports IDLE, NONSEQ and SEQ, but not other transfer types;
- HSIZE supports word, byte and half-word transfers, other transfer sizes are not supported;
- HWRITE supports both read and write operations.

9.2.2.2 Supported response types

Under the AHB and AHB-Lite protocols, the bus interface accepts the following response types from the slave device:

- HREADY supports Ready and Not Ready;
- HRESP supports OKAY and ERROR, but not other response types.

9.2.3 Behavior under different bus responses

Table 9.3 lists the CPU behavior when different bus responses occur on the bus.

Table 9.3: Bus exception handling

HREADY	HRESP	Results
Don't care	ERROR	Access error, the bus ends the transfer and handles the access error.
High	OKEY	Transmission ends.
Low	OKEY	Inserts a wait state.

9.2.4 AHB protocol interface signals

Table 9.4: AHB protocol interface signals

Signal	I/O	Reset	Definition
name biu_pad_haddr[31:0]	O	-	Address bus: 32-bit address bus.
biu_pad_hwdata[31:0]	O	-	Write data bus: 32-bit write data bus.
biu_pad_hburst[2:0]	O	-	Burst transmission indication signal: 000: SINGLE; 001: INCR; 010: WRAP4. E804 supports SINGLE and WRAP4 burst transmission.
biu_pad_hsize[2:0]	O	-	Transmission width indication signal: 000: byte; 001: halfword; 010: word.
biu_pad_htrans[1:0]	O	00	Transmission type indicates signal: 00: IDLE; 01: BUSY; 10: NONSEQ; 11: SEQ. The E804 supports IDLE, NONSEQ and SEQ transmission types.
biu_pad_hwrite	O	0	Read and write indication signals: 1: indicates a write bus transfer; 0: Indicates a read bus transfer.

Continue on next page

Table 9.4 – Continued from previous page

Signal	I/O	Reset	Definition
name biu_pad_hprot[3:0]	O	-	Protection control signal: ***0: fetch instruction; ***1: Data access; **0*: user access; **1*: Superuser access; *0**: Not buferable; *1**: buferable; 0***: Not cacheable; 1***: cacheable.
biu_pad_hbusreq	O	0	Bus request signal: Instructs the CPU to request the right to use the bus.
biu_pad_hlock	O	0	Lock bus signal: When lock is declared, the CPU has exclusive control of the bus. Other busmasters can occupy the bus.
pad_biu_hrdata[31:0]	I	-	Read data bus: 32-bit read data bus.

Table 9.5: AHB protocol interface signals

Signal name	I/O	Reset	Definition
pad_biu_hready	I	-	Transfer completion indication signal: When valid, it indicates that the current transfer has been completed and the CPU puts the bus in Standby status.
pad_biu_hgrant	I	-	Bus occupancy indication signal: When valid, indicates that the CPU is currently occupying the bus.
pad_biu_hresp[1:0]	I	-	Transmission response signal: 00: OKAY; 01: ERROR; 10: RETRY; 11: SPLIT. E804 supports only OKAY and ERROR response types.

9.2.5 AHB-Lite protocol interface signals

Table 9.6: AHB-Lite protocol interface signals

Signal	I/O	Reset Definition	
name biu_pad_haddr[31:0]	O	-	Address bus: 32-bit address bus.
biu_pad_hwdata[31:0]	O	-	Write data bus: 32-bit write data bus.
biu_pad_hburst[2:0]	O	-	Burst transmission indication signal: 000: SINGLE; 001: INCR; 010: WRAP4. E804 supports SINGLE and WRAP4 burst transmission.
biu_pad_hsize[2:0]	O	-	Transmission width indication signal: 000: byte; 001: halfword; 010: word.
biu_pad_htrans[1:0]	O	00	Transmission type indicates signal: 00: IDLE; 01: BUSY; 10: NONSEQ; 11: SEQ. The E804 supports IDLE, NONSEQ and SEQ transmission types.
biu_pad_hwrite	O	0	Read and write indication signals: 1: indicates a write bus transfer; 0: Indicates a read bus transfer.
biu_pad_hprot[3:0]	O	-	Protection control signal: ***0: fetch instruction; ***1: Data access; **0*: user access; **1*: Superuser access; *0**: Not buferable; *1**: buferable; 0***: Not cacheable; 1***: cacheable.
pad_biu_hrdata[31:0]	I	-	Read data bus: 32-bit read data bus.

Table 9.7: AHB-Lite protocol interface signals

Signal	I/O	Reset Definition	
name pad_biu_hready	I	-	Transfer completion indication signal: When valid, it indicates that the current transfer has been completed and the CPU puts the bus in Standby status.
pad_biu_hresp[1:0]	I	-	Transmission response signal: 00: OKAY; 01: ERROR; 10: RETRY; 11: SPLIT. E804 supports only OKAY and ERROR response types.

9.3 Instruction Bus Interface

9.3.1 Features

The E804 instruction bus only supports the AMBA3.0 AHB-Lite protocol. Please refer to the AMBA 3.0 Specification - AMBA3 AHB-Lite Protocol Specification Rev 1.0.

The basic characteristics of the instruction bus interface are:

- Compatible with AMBA3.0 AHB-Lite bus protocol;
- Only supports direct output (Non-Flop-out) mode;

9.3.2 Contents of the Agreement

9.3.2.1 Supported transmission types

Considering the application field and cost of E804, the instruction bus interface only implements part of the AHB-Lite protocol.

Under the AHB-Lite protocol, as a master device, the bus interface supports the following transmission types:

- HBURST only supports SINGLE transmission, other burst types are not supported;
- HTRANS only supports IDLE and NONSEQ, no other transfer types are supported;
- HSIZE supports word, byte and half-word transfers, other transfer sizes are not supported;
- HWRITE supports both read and write operations.

9.3.2.2 Supported response types

Under the AHB-Lite protocol, the bus interface accepts the following response types from the slave device:

- HREADY supports Ready and Not Ready;
- HRESP supports OKAY and ERROR, but not other response types.

9.3.3 Behavior under different bus responses

Table 9.8 lists the CPU behavior when different bus responses occur on the bus.

Table 9.8: Bus exception handling

HREADY HRESP Results	
Don't care ERROR Access error, the bus ends the transfer and handles the access error.	
High	OKEY Transmission ends.
Low	OKEY Inserts a wait state.

9.3.4 Command bus interface signals

Table 9.9: Command bus interface signals

Signal	I/O	Reset	definition
name iahbl_pad_haddr[31:0]	O	-	Address bus: 32-bit address bus.
iahbl_pad_hburst[2:0]	O	-	Burst transmission indication signal: 000: SINGLE; 001:INCR; 010:WRAP4. E804 supports only SINGLE burst transfers.
iahbl_pad_hprot[3:0]	O	-	Protection control signal: ***0: fetch instruction; ***1: Data access; **0*: user access; **1*: Superuser access; *0**: Not buferable; *1**: buferable; 0***: Not cacheable; 1***: cacheable.
iahbl_pad_hsize[2:0]	O	-	Transmission width indication signal: 000:byte; 001: halfword; 010: word.
iahbl_pad_htrans[1:0]	O	00	The transmission type indicates the signal: 00: IDLE; 01: BUSY; 10:NONSEQ; 11: SEQ. E804 only supports IDLE and NONSEQ transmission classes type.
iahbl_pad_hwdata[31:0]	O	-	Write data bus: 32-bit write data bus.
iahbl_pad_hwwrite	O	0	Read and write indication signals: 1: indicates a write bus transfer; 0: Indicates a read bus transfer.
pad_iahbl_hrdta[31:0]	I	-	Read data bus: 32-bit read data bus.

Table 9.10: Command bus interface signals

Signal	I/O	Reset	definition
name pad_iahbl_hready	I	-	Transfer completion indication signal: When valid, it indicates that the current transfer has been completed and the CPU will always The line is placed on standby.
pad_iahbl_hresp	I	-	Transmission response signal: 0: OK; 1: ERROR.
pad_bmu_iahbl_base[19:0]	I	-	IAHB-Lite base address control signal, after power-on reset Need to be fixed
pad_bmu_iahbl_mask[19:0]	I	-	IAHB-Lite address alignment control signal, power-on reset Need to be fixed later

9.4 Data Bus Interface

9.4.1 Features

The data bus of E804 only supports AMBA3.0 AHB-Lite protocol. Please refer to AMBA 3.0 Specification - AMBA3 AHB-Lite Protocol Specification Rev 1.0.

The basic characteristics of the data bus interface are:

- Compatible with AMBA3.0 AHB-Lite bus protocol;
- Only supports direct output (Non-Flop-out) mode.

9.4.2 Agreement Content

9.4.2.1 Supported transmission types

Considering the application field and cost of E804, the data bus interface only implements part of the AHB-Lite protocol.

Under the AHB-Lite protocol, as a master device, the bus interface supports the following transmission types:

- HBURST supports SINGLE transmission, other burst types are not supported;
- HTRANS only supports IDLE and NONSEQ, no other transfer types are supported;
- HSIZE supports word, byte and half-word transfers, other transfer sizes are not supported;
- HWRITE supports both read and write operations;

9.4.2.2 Supported response types

Under the AHB-Lite protocol, the bus interface accepts the following response types from the slave device:

- HREADY supports Ready and Not Ready;
- HRESP supports OKAY and ERROR, but not other response types.

9.4.3 Behavior under different bus responses

Table 9.11 lists the CPU behavior when different bus responses occur on the bus:

Table 9.11: Bus exception handling

HREADY	HRESP	Results
Don't care	ERROR	Access error, the bus ends the transfer and handles the access error.
High	OKEY	Transmission ends.
Low	OKEY	Inserts a wait state.

9.4.4 Data bus interface signals

Table 9.12: Data bus interface signals

Signal	I/O	Reset	Definition
name dahbl_pad_haddr[31:0]	O	-	Address bus: 32-bit address bus.
dahbl_pad_hburst[2:0]	O	-	Burst transmission indication signal: 000: SINGLE; 001: INCR; 010: WRAP4. E804 supports only SINGLE burst transfers.
dahbl_pad_hprot[3:0]	O	-	Protection control signal: ***0: fetch instruction; ***1: Data access; **0*: user access; **1*: Superuser access; *0**: Not buferable; *1**: buferable; 0***: Not cacheable; 1***: cacheable.
dahbl_pad_hsize[2:0]	O	-	Transmission width indication signal: 000: byte; 001: halfword; 010: word.
dahbl_pad_htrans[1:0]	O	00	Transmission type indicates signal: 00: IDLE; 01: BUSY; 10: NONSEQ; 11: SEQ. E804 supports only IDLE and NONSEQ transfer types.
dahbl_pad_hwdata[31:0]	O	-	Write data bus: 32-bit write data bus.
dahbl_pad_hwwrite	O	0	Read and write indication signals: 1: indicates a write bus transfer; 0: Indicates a read bus transfer.
pad_dahbl_hrdta[31:0]	I	-	Read data bus: 32-bit read data bus.

Table 9.13: Data bus interface signals

Signal	I/O	Reset	definition
name pad_dahbl_hready	I	-	Transfer completion indication signal: When valid, it indicates that the current transfer has been completed and the CPU will always The line is placed on standby.
pad_dahbl_hresp	I	-	Transmission response signal: 0: OK; 1: ERROR.
pad_bmu_dahbl_base[19:0]	I	-	DAHB-Lite base address control signal, power-on reset Need to be fixed later
pad_bmu_dahbl_mask[19:0]	I	-	DAHB-Lite address alignment control signal, power-on reset Need to be fixed after

9.5 Instruction and Data Access Order

Once the CPU is configured with cache, all non-cacheable data will bypass the cache and directly access the corresponding bus from the bus matrix.

This is shown as channel 1 in Figure 9.2 ;

The cacheable data will first access the cache. If it hits, the data will be retrieved directly from the cache. If it does not hit, the cache will pass A request is made to the bus through the bus matrix, as shown in channel 2 in Figure 9.2.

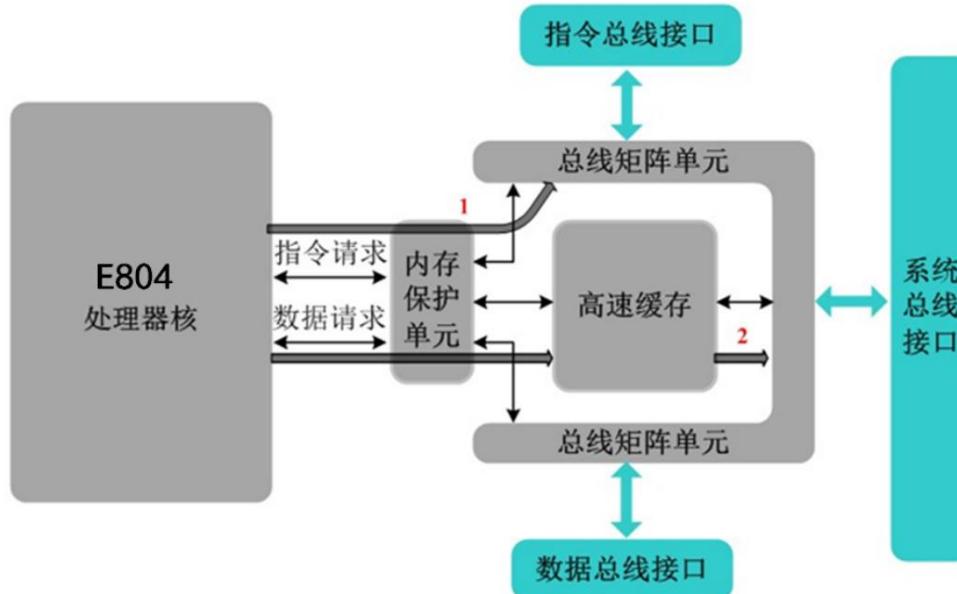


Figure 9.2: Access sequence of external bus

Chapter 10 Debug Interface

10.1 Overview

The debug interface is a channel for software to interact with the processor. Users can obtain information such as CPU registers and memory contents through the debug interface.

In addition, program download and other operations can also be completed through the debug interface.

In order to meet the low-cost application requirements and save CPU external pins, CSKY defines a set of debugging interfaces, JTAG2 interfaces. JTAG2 debugging interface includes JTAG2 communication protocol and JTAG2 interface controller. E804 supports 2-wire JTAG protocol, and the debugging interface uses JTAG2 protocol to communicate with external debuggers.

The main features of the debug interface are as follows:

- Support 2-wire JTAG interface;
- Non-intrusive access to CPU status;
- Supports synchronous debugging and asynchronous debugging to ensure that the processor enters debug mode in extremely harsh conditions;
- Support soft breakpoints;
- Multiple memory breakpoints can be set;
- Check and set the values of CPU registers;
- Inspect and change memory values;
- Instructions can be executed in single or multi-step;
- Fast downloading of programs;
- Debug mode can be entered after a CPU reset or in normal user mode;
- The debug register resources can be accessed using the TCIP interface, see the debug manual for details.

The debugging of C-SKY CPU is completed by the debugging software, debugging agent service program, debugger and debugging interface. The position of the debugging interface in the entire CPU debugging environment is shown in Figure 10.1. Among them, the debugging software and debugging agent service program are interconnected through the network, the debugging agent service program and the debugger are connected through USB, and the debugger communicates with the CPU debugging interface in JTAG mode.

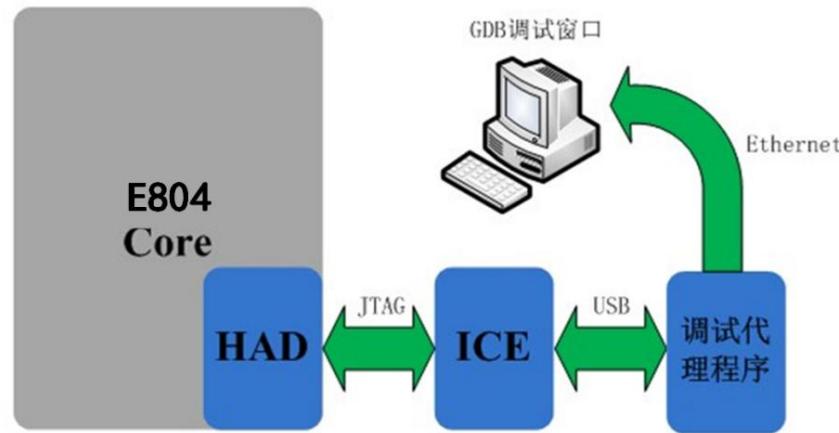


Figure 10.1: The location of the debug interface in the entire CPU debug environment

10.2 External Interface

The interface between the debug module and the outside world is mainly JTAG-related interface signals and debug-related interface signals. Table 10.1 lists the debug-related interface signals.

Table 10.1: Interface signals between debug module and external devices

Signal Name	Direction	(sysio_pad_dbg_b)
Output pad_had_jdb_req_b	Input	
had_pad_jdb_ack_b	Output	pad_had_jtg_tap_en
Input had_pad_jtg_tap_on	Output	
had_pad_jdb_pm[1:0]	Output	pad_had_jtg_tclk
Input pad_had_jtg_trst_b	Input	
pad_had_jtg_tms_i	Input	had_pad_jtg_tms_o
Output had_pad_htg_tms_oe	Output	

1. biu_pad_dbg_b (sysio_pad_dbg_b)

A low level indicates that the CPU is in debug mode.

2. pad_had_jdb_req_b and had_pad_jdb_ack_b

The pad_had_jdb_req_b signal is a request signal for the CPU to asynchronously enter the debug state. This signal must be kept low for at least two JTAG clock cycles to ensure that the CPU enters the debug state and can debug the program. If this signal is kept low for less than two JTAG clock cycles, the CPU may enter the debug state but cannot debug the program, because the

The signal is also used to enable the TAP state machine in the debug interface.

After the CPU enters the debug state, the had_pad_jdb_ack_b signal will maintain a low level for two JTAG clock cycles as an acknowledgement.

3. pad_had_jtg_tap_en and had_pad_jtg_tap_on

The pad_had_jtg_tap_en signal is the enable signal of the TAP state machine in the debug interface. Maintaining this signal at a high level for at least one JTAG clock cycle can enable the TAP state machine in the debug interface. If this signal is always invalid after the CPU is powered on, the program may not be debugged when the CPU enters the debug state using a synchronous method (such as setting the DR bit in the debug interface register HCR, breakpoints, etc.).

After the TAP state machine is started, the had_pad_jtg_tap_on signal will be pulled high.

4. had_pad_jdb_pm[1:0]

The had_pad_jdb_pm[1:0] signal indicates the current working mode of the CPU. This signal can be used to determine whether the CPU has entered the debug mode. See [Table 10.2](#) for details .

Table 10.2: had_pad_jdb_pm indicates the current CPU status

had_pad_jdb_pm[1:0] Indicates normal mode.	
00	Low power
01	mode (STOP, DOZE, WAIT). Debug mode. Reserved.
10	
11	

5. pad_had_jtg_tclk

JTAG clock signal. This signal is an external input signal, usually a clock signal generated by a debugger. The frequency of this clock signal must be lower than 1/2 of the frequency of the CPU clock signal to ensure normal operation between the debugging module and the CPU core.

6. pad_had_jtg_trst_b

The pad_had_jtg_trst_b signal is the JTAG reset signal, which can reset the TAP state machine and other related control signals.

7. JTAG_2 related signals

The pad_had_jtg_tms_i signal is a 2-wire JTAG serial data input signal. The debug interface is on the JTAG clock signal.

The rising edge of TCLK samples it, while the external debugger sets this signal on the falling edge of the JTAG clock;

This signal must be kept at a high level when idle, and the clock signal is preferably stopped when idle. Users can use this signal to synchronously reset the HAD logic: In the debug module that implements the 2-wire JTAG interface, if the clock signal is always valid, the user only needs to keep this signal at a high level and maintain it for 80 clock cycles to synchronously reset the debug module. After resetting the debug module, the TAP state machine of the debug module returns to the RESET state, and the debug module register HACR is reset to 0x82 (pointing to the ID register).

The had_pad_jtg_tms_o signal is a 2-wire JTAG serial data output signal. The debug interface is on the JTAG clock signal.

The falling edge of TCLK sets it, while the external debugger samples it on the rising edge of the JTAG clock;

The had_pad_jtg_tms_oe signal is a valid indication signal of the had_pad_jtg_tms_o signal. The CPU should use this signal to combine the pad_had_jtg_tms_i and had_pad_jtg_tms_o signals into a bidirectional port signal.

Chapter 11 Working Mode and Conversion

E804 has three types of working modes: normal operation mode, low power mode and debug mode. The low power mode is divided into three types: STOP mode, DOZE mode and WAIT mode. The differences between these three low power modes are defined by the SOC designer. This chapter will introduce the CPU working mode and the conversion between modes in detail.

As shown in Figure 11.1 , there are three working modes of E804, namely normal working mode, low power working mode and debugging mode. The working mode of the CPU can be obtained by querying the sysio_pad_jdb_pm[1:0] signal.

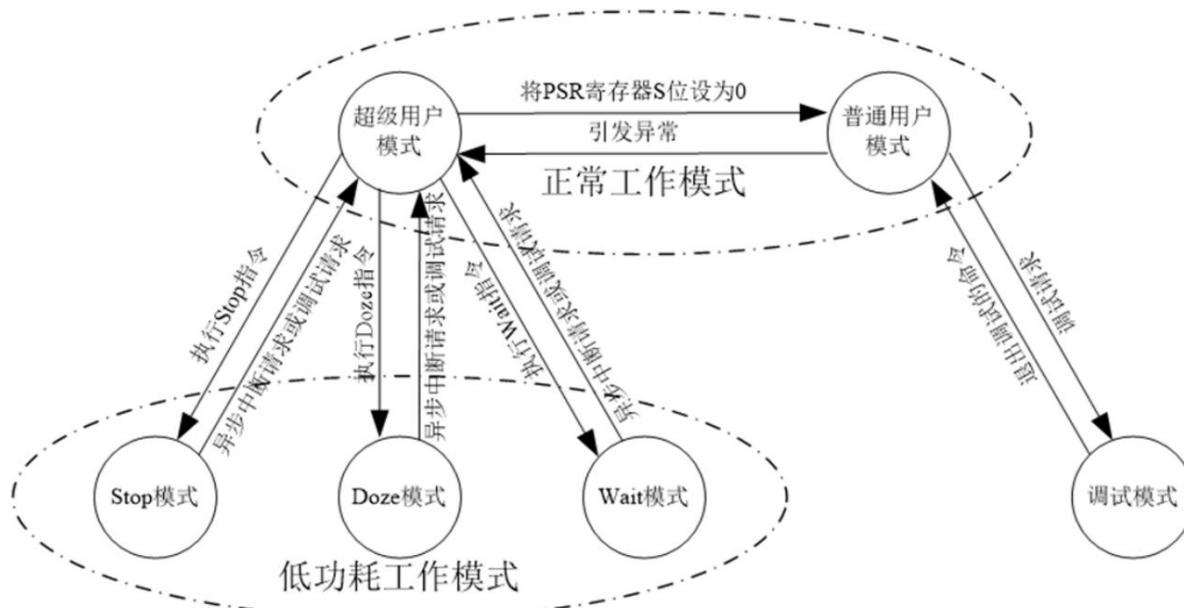


Figure 11.1: Schematic diagram of various working states of the CPU

11.1 Normal working mode

The CPU can be operated in two normal modes: supervisor mode and normal user mode.

Query the S bit in the PSR register to obtain it.

When the S bit is 1, the CPU operates in supervisor mode;

When the S bit is 0, the CPU operates in normal user mode.

When the CPU is working in supervisor mode, you can enter normal user mode by setting the S bit to 0; when the CPU is working in normal user mode, In user mode, enter super user mode by raising an exception.

11.2 Low Power Mode

After the CPU executes the low-power instruction (STOP, DOZE, WAIT), the CPU will enter the low-power mode. The process of executing the three instructions is that after the CPU executes the low-power instruction, it will wait for all previous instructions to be executed, and then complete the low-power instruction. At the same time, according to the type of low-power instruction, the sysio_pad_lpmd_b[1:0] signal is pulled up to stop executing instructions and freeze the pipeline, turning off the internal clock.

Only asynchronous interrupt requests (pad_sysio_intraw_b and pad_sysio_fntraw_b signals) or debug requests can cause the CPU to exit low power mode, and then the CPU continues to execute subsequent instructions from the low power instruction that entered the low power mode. The low power mode that the current CPU is working in can be obtained by querying the signal line sysio_pad_lpmd_b[1:0]. The specific low power scenario corresponding to each mode is determined by the SOC designer.

11.3 Debug Mode

11.3.1 Debug Mode

After entering the debug mode, the CPU stops fetching and executing instructions and enters a waiting state. In this state, the CPU only executes the instructions input by the HAD. The CPU state can be queried and modified through the HAD input instructions, and then debugging can be performed.

11.3.2 Entering debug mode

When the CPU receives a debug request, it enters debug mode. The debug request sources can be the following seven:

- The external input signal of HAD (pad_had_jdb_req_b) enables the processor to enter debug mode asynchronously;
- When the IDRE bit of E804 HCR is valid, the external input signal biu_had_sdb_req_b (pad_biu_dbgreq_b) of HAD is synchronized by the CPU to request the processor to enter the debug mode synchronously;
- When the ADR bit in the E804 HCR is valid, the processor enters debug mode directly;
- When the DR bit of the E804 HCR is valid, the processor enters debug mode after completing the current instruction;
- When the FDB bit in the E804 CSR is valid, the processor enters debug mode when executing the bkpt instruction;
- When the TME bit in the E804 HCR is valid, the processor enters debug mode after the trace counter value is decremented to 0;
- Enter debug mode by storing hardware breakpoints at E804.

The processor executes the current instruction, saves the pipeline information, and then enters the debug mode. When the processor is in low power mode, the processor can exit low power mode and enter debug mode by setting the ADR in HCR and the debug request of DR. After entering debug mode, the CPU stops executing the current instruction and waits for the user to enter a valid instruction through the debug interface for execution or exiting debug mode.

11.3.3 Exit debug mode

If the GO and EX bits of the E804 HACR are set to 1 and R/W is 0 (write operation), RS selects WBBR, PSR,

If the PC, IR, CSR or Bypass register is set, the CPU will exit the debug mode and enter the normal working mode when executing the instruction.

Note: Since PC, CSR, and PSR are variable in debug mode, the values in the above registers must be set to the same value when exiting debug mode.

The value saved when entering debug mode.

Chapter 12 Floating Point Processing Unit

12.1 Overview

12.1.1 Introduction

As a configurable hardware unit of E804, the floating point unit is designed to improve the processing capability of E804 for floating point applications. The E804 floating point unit provides a low-cost, high-performance hardware floating point implementation.

The floating point unit supports single-precision floating point operations in the IEEE-754 floating point standard and implements 16 single-precision floating point registers.

With support, E804 can support double-precision floating-point operations.

12.1.2 Features

The main features of the floating-point unit architecture and programming model are as follows:

- Fully compatible with ANSI/IEEE Std 754 floating point standard (with system software support);
- Only supports single-precision floating-point operations;
- Supports four rounding modes: rounding towards zero, rounding towards positive infinity, rounding towards negative infinity and rounding to the nearest;
- Supports both trapping and non-trapping processing modes for floating-point exceptions;
- Support accurate handling of floating point exceptions;
- Support for floating-point hardware division and square root.

The main features of the floating-point unit's microarchitecture are as follows:

- 16 independent single-precision floating-point registers;
- Single-issue architecture, processing one floating-point arithmetic instruction per cycle;
- Support in-order issuance, in-order execution, and in-order write-back of floating-point arithmetic instructions;
- Contains three independent execution pipelines, namely floating-point ALU, floating-point multiplication and floating-point division square root;
- Optimized execution delay technology, except for the floating-point division square root instruction, all instructions can be executed within 1-2 clock cycles;
- Cost optimization technology based on reuse of computing components;

- Power optimization technology based on clock gating and data path isolation.

12.2 Microarchitecture

12.2.1 Introduction to Floating Point Unit

The floating-point unit mainly includes the following three execution sub-units:

- Arithmetic Logic Unit (FALU)
- Multiplication (FMULU) unit
- Division and Square Root (FDSU) Unit

The arithmetic logic unit (FALU) is responsible for the execution of instructions such as addition, subtraction, comparison, conversion, negation, absolute value, register transfer, etc.

Results were achieved in 1-2 cycles.

The multiplication unit (FMULU) is responsible for multiplication, multiplication and negation operations, and the multiplication result is completed in 2 cycles.

The Division Square Root Unit (FDSU) implements division, square root, and reciprocal operations, producing 1 bit of result per cycle and taking up to 28 cycles to complete.

12.3 Programming Model

12.3.1 Introduction

The E804 floating-point unit implements the single-precision portion of the ANSI/IEEE Std 754 floating-point standard.

- Supports the loading of floating-point immediate numbers and improves the efficiency of generating floating-point immediate numbers;

12.3.2 Data Format

The floating point unit only supports single precision data format. The value of the floating point register can be assigned by three methods: floating point load instruction, floating point constant load instruction, and floating point register transfer instruction. The data in the floating point register has the same format as the data in the memory. Any transfer or loading will not change this format.

The program itself is responsible for recording the data type stored in each register. Therefore, the hardware will not make any judgments about the data type of the register. For example, even if the data stored in the floating-point register is an integer, a floating-point instruction will only process this data as single-precision data. In other words, the hardware's interpretation of the data depends only on the executed instruction.

12.3.2.1 Integer Data Format

The floating point unit supports signed or unsigned 32-bit integers. The integer format in the floating point registers and the E804 main pipeline registers are Or consistent in memory.

12.3.2.2 Single-precision data format

Figure 12.1 shows the region segmentation of single-precision floating-point numbers:

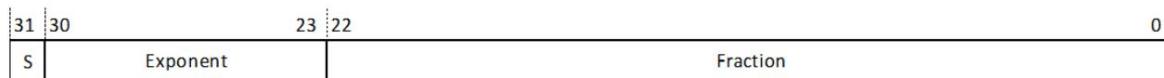


Figure 12.1: Single-precision data format

Single-precision data formats include:

- Sign bit (S): bit[31];
- Biased exponent domain: The biased exponent $e=E+\text{bias}$, here bias=127, bits[30:23];
- The fraction field: The fraction, bits[22:0].

12.3.3 Floating-point register description

The E804 floating-point unit has independent floating-point registers, which consist of 16 32-bit single-precision registers.

Figure 12.2 shows the E804 register resource diagram including the floating point unit. Compared with the shaping pipeline, the floating point unit has more floating point registers, which can be used to Accessed in both normal and privileged modes. Three system registers have been added, which are only accessed in privileged mode.

12.3.3.1 Data Transfer with General Registers

Data transfer between general registers and floating-point registers can be achieved through floating-point register transfer instructions.

include:

- FMTVRL Floating-point register write transfer instruction.
- FMFVRL Floating Point Register Read Transfer instruction.

12.3.3.2 Maintaining Register Precision Consistency

Floating point registers can store single precision floating point numbers and integer data. For example, the data type stored in floating point register FR1 may be either of the two data types, depending on the last write operation.

The floating-point unit does not perform any data format detection on the data type in hardware. The hardware's interpretation of the data format in the floating-point register depends only on the executed floating-point instruction itself, and does not care about the data format used in the last write operation of this register. It is entirely up to the compiler or the program itself to ensure the consistency of the data precision in the register.

12.3.4 System Registers

The floating-point unit has three related system registers: floating-point version register FID (CR<0,2>), floating-point control register FCR (CR<1,2>), and floating-point exception status register FESR (CR<2,2>). The floating-point version register records the implementation and version information of the floating-point unit.



Figure 12.2: Register resources in programming mode

The floating point control register is used to configure floating point exception handling and execution, while the floating point exception status register records the floating point exceptions during execution.

Click abnormal information.

Table 12.1 lists the three system registers of the floating-point unit.

Table 12.1: Overview of floating point system registers

Register access mode	CR		
Floating point version register, FID (FPU ID Register)	CR<0,2> Read-only privileged mode		
Floating point control register, FCR (FPU Control Register)	CR<1,2> Read/write privileged mode		
Floating point exception status register FESR (Exception Status Register) CR<2,2>	Read/write privileged mode		

12.3.4.1 Reading and Writing Methods of Floating-Point System Registers

Use MTCR and MFCR instructions to read and write floating-point system registers.

Therefore, MTCR and MFCR need to read and write coprocessor No. 2. The specific register index method is: floating-point version register

CR<0,2>, floating point control register CR<1,2>, floating point exception status register CR<2,2>.

The instruction to write floating-point system registers is MTCR Rx, CR<y,2>, which uses the value of general register Rx to rewrite the value of floating-point system register y. value.

The instruction to read the floating-point system register is MFCR Rx, CR<y,2>, which reads the value of the floating-point system register y and rewrites the general register Rx.

12.3.4.2 Floating Point Version Register (FPU ID Register, FID)

The floating-point version register is a read-only register used to identify the version information of the floating-point unit.

Figure 12.3 shows the bit fields of the floating-point version register:

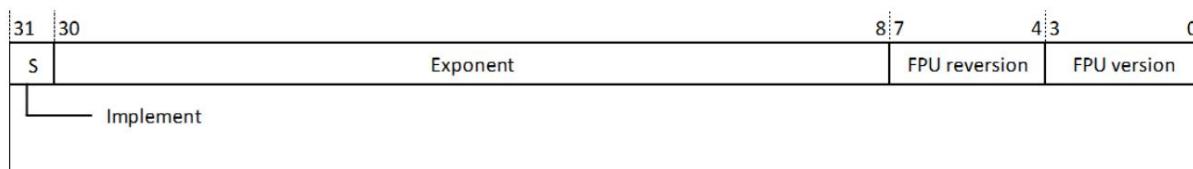


Figure 12.3: Floating point version registers

Software users can determine whether the current processor is configured with a floating-point unit and version information by querying the floating-point version register.

Table 12.2 describes the definition of each bit of the FID register:

Table 12.2: Floating-point version register description

	Function	Read/Write	describe
Position [31]	Implement	Read Only	Floating point unit implementation: 1: Implemented hardware floating point unit; 0: Hardware floating point unit is not implemented, the processor will simulate it through software Floating point function.
[30:8]	Reserved	Read-only	All 0s.
[7:4]	FPU reversion Read-only	Read-only	Upgrade information for floating point units.
[3:0]	FPU version		Version information for the floating point unit.

12.3.4.3 Floating Point Control Register (FPU Control Register, FCR)

The floating point control register is readable and writable, and is responsible for configuring the exception care and execution mode of the floating point unit.

The reserved bits of FCR are initialized to 0.

The bits will not be modified by mistake. The code recommends reading first, then modifying the read value, and finally writing it back to FCR.

Failure to follow this approach may result in unexpected errors in subsequent programs.

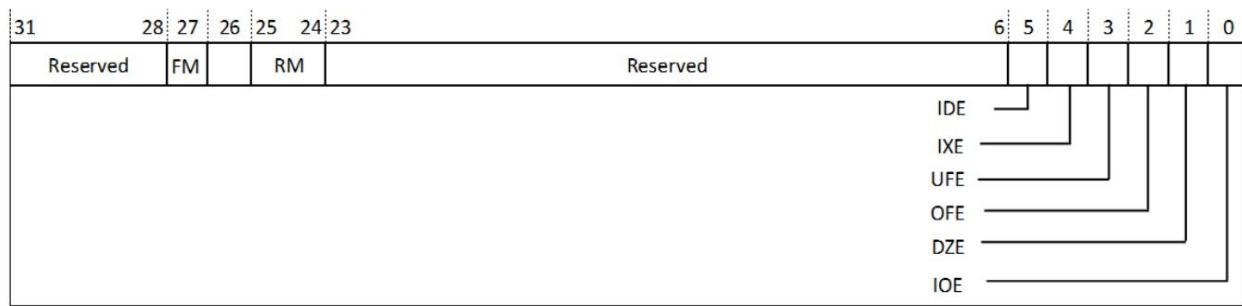


Figure 12.4: Floating-point control registers

Reset value: 0x00000000.

The definition of each bit in the FCR register is shown in Table 12.3 .

Table 12.3: Floating point control register descriptions

Bit Function	Read/Write	Description
[31:28]	Reserved	Read-only All 0

Continue on next page

Table 12.3 – Continued from previous page

[27]	FM	Reading/writing	this field determines the exception enable bit for input non-standard number/overflow exception enable bit for operation result. When IDE/OFE is not enabled, the denormalized number of processing methods. 1: Convert the denormalized number to a signed minimum normal number. 0: flush denormalized numbers to signed zero.
[25:24] RM		Reading/writing	this field determines the rounding mode for floating-point calculations. 00: Round to the Nearest 01: Round towards 0 10: Round towards +∞ 11: Round towards -∞
[23:6] Reserved	Read-only	All 0	
[5]	IDE read/write	input denormalized number exception enable bit (IDE, Input Denormalized trap Enable bit).	 1: Enable. In this case, if an input non-standard number is encountered, the processor will terminate the program execution and throw an abnormal The specification number input is abnormal. 0: Disable. At this time, the processor will process the input non-standard number according to the FM indicator bit. And continue the operation.
[4]	IXE Read/write	IneXact trap Enable bit (IXE, IneXact trap Enable bit).	 1: Enable. At this time, if the calculation result is not accurate, the processor will terminate the program execution and throw an inaccurate If an exception is detected, no result will be written back. 0: Disable. At this time, the processor will round the operation result according to the RM indicator bit. And continue the operation.
[3]	UFE reads/writes	the UnderFlow trap Enable bit (UFE, UnderFlow trap Enable bit).	 1: Enable. At this time, if the operation result overflows, the processor will terminate the program execution and throw an underflow If an exception occurs, no result will be written back. 0: Disable. At this time, the processor will process the underflow result according to the FM indicator bit. And continue the operation.
[2]	OFE read/write	overflow exception enable bit (OFE, OverFlow trap Enable bit).	 1: Enable. At this time, if the operation result overflows, the processor will terminate the program execution and throw an overflow If an exception occurs, no result will be written back. 0: Disable. At this time, the processor will process the overflow result according to the RM indicator bit. And continue the operation.
[1]	DZE Read/write	Division by Zero trap Enable bit (DZE, Division by Zero trap Enable bit).	 1: Enable. In this case, if a division by zero operation occurs, the processor will terminate the program execution and throw a division by zero error. Abnormal, no result is written back. 0: Disable. In this case, the processor will use positive/negative infinite floating point numbers as the calculation results and continue to operate. Calculate.

Continue on next page

Table 12.3 – Continued from previous page

[0]	IOE read/write illegal operation exception enable bit (IOE, Invalid Operation trap Enable bit).	
	1: Enable. At this time, if an illegal operand occurs, the processor will terminate the program execution and throw an illegal operation exception. The operation is abnormal and no result is written back.	
	0: Disabled. In this case, the processor will use QNaN as the operation result and continue the operation.	

12.3.4.4 Floating Point Exception Status Register (FESR)

The FESR provides floating-point exception status information. Each bit indicates the type of floating-point exception that occurred. The FESR is accessible only in Supervisor mode. ask.

The various regions of FESR are shown in Figure 12.5:

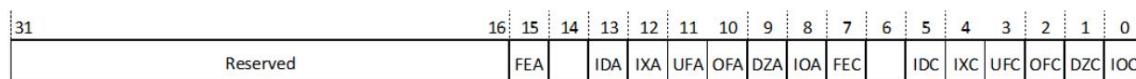


Figure 12.5: Floating Point Exception Status Register

Table 12.4 shows the bit definitions of the floating point exception status register.

Table 12.4: Floating point exception status register description

Bit Function	Read/Write Description	
[29:16] Reserved, read-only, all 0s.		
[15]	FEA Read/write floating point exception accumulation bits. This bit will be set when any floating point exception occurs and the corresponding enable bit is not turned on. 1.	
[14] Reserved	Read-only All 0	
[13]	IDA Read/write Input Denormalized Accumulative bit (IDA, Input Denormalized Accumulative bit). This bit is set to 1 when an input denormal exception occurs and the IDE bit is 0.	
[12]	IXA Read/write Inexact Accumulative bit (IXA, Inexact Accumulative bit). This bit is set to 1 when an inexact exception occurs and the IXE bit is 0.	
[11]	UFA reads/writes the UnderFlow Accumulative bit (UFA, UnderFlow Accumulative bit). This bit is set to 1 when an underflow exception occurs and the UFE bit is 0.	
[10]	OFA Read/write overflow exception accumulation bit. (OFA, OverFlow Accumulative bit). This bit is set to 1 when an overflow exception occurs and the OFE bit is 0.	
[9]	DZA Read/write Division by Zero Accumulative bit. This bit is set to 1 when a divide-by-0 exception occurs and the DZE bit is 0.	
[8]	IOA Read/write illegal operation number exception accumulation bit. (IOA, Invalid Operation Accumulative bit). This bit is set to 1 when an illegal operand exception occurs and the IOE bit is 0.	
[7]	FEC Read/Write This bit will be set to 1 when any floating point exception occurs in the current floating point instruction, otherwise it will be cleared to 0.	
[6]	Reserved for read-only all 0s.	

Continue on next page

Table 12.4 – Continued from previous page

[5]	IDC read/write input denormalized current bit (IDC, Input Denormalized Current bit).	When a non-standard number input exception occurs in the current floating-point instruction, this bit is set to 1, otherwise it is cleared to 0.
[4]	IXC Read/write inexact exception indication bit (IXC, InExact Current bit).	This bit is set to 1 when an inexact exception occurs in the current floating-point instruction, otherwise it is cleared to 0.
[3]	UFC reads/writes the overflow exception indication bit (UFC, UnderFlow Current bit)	When an underflow exception occurs in the current floating-point instruction, this bit is set to 1, otherwise it is cleared to 0.
[2]	OFC read/write overflow exception indication bit (OFC, OverFlow Current bit)	When an overflow exception occurs in the current floating-point instruction, this bit is set to 1, otherwise it is cleared to 0.
[1]	DZC Read/write Division by Zero Current bit (DZC, Division by Zero Current bit).	This bit is set to 1 when a division by zero exception occurs in the current floating-point instruction, otherwise it is cleared to 0.
[0]	IOC reads/writes the illegal operation exception indication bit (IOC, Invalid Operation Current bit).	When an illegal operand exception occurs in the current floating-point instruction, this bit is set to 1, otherwise it is cleared to 0.

12.3.5 Data size

The data endianness is proposed relative to the format of data storage in the memory. The high address byte is stored in the low bit of the physical memory and is defined as the big endian; the high address byte is stored in the low bit of the physical memory and is defined as the big endian.

The high order bits of address bytes stored in physical memory are defined as little endian.

The memory access unit of the E804 floating point is the same as the memory access unit of the shaping pipeline, supporting the little-endian format and the V1 version big-endian (CSKY).

The CPU has designed two versions of the big-endian and small-endian formats, V1 and V2, and does not support unaligned access. In the little-endian mode, regardless of the access instruction

The size of the address is byte/halfword/word, and the lowest byte data always corresponds to the lowest bit of the address; in the big-endian mode of V1 version, it is necessary to

The access size (byte/halfword/word) of the access instruction determines the data. Since the floating-point access instruction accesses in units of words, the little-endian and V1

The storage access mode of floating-point access instructions in big-endian mode is the same as that of the previous version.

Table 12.5: Little-endian/big-endian memory access modes

Access size	Access address	Little	V1 Big Endian
	A	D3D2D1D0 D3D2D1D0	
endian word	Half	D1D0	D3D2
word A	Half word A+2	D3D2	D1D0
Byte A	Byte A+1	D0	D3
Byte A+2	Byte A+3	D1	D2
		D2	D1
		D3	D0

12.4 Exception Handling

This chapter describes the exception handling of floating-point operations. It includes the following contents.

- Introduction to floating point exceptions;

- Input Denormalized Exception;
- Invalid Operation Exception;
- Divide by Zero Exception;
- Underflow Exception;
- Overflow Exception;
- Inexact Exception;
- Exception priority.

12.4.1 Introduction to floating point exceptions

Floating-point instructions generate 6 types of floating-point arithmetic exceptions, including the non-standard input exception in addition to the 5 floating-point operation exceptions defined by the floating-point protocol standard (ANSI/IEEE Std 754). The exception response process of floating-point instructions is similar to the mainstream pipeline. All 6 types of floating-point arithmetic exceptions share the same exception vector number 30.

After the processor responds to a floating-point arithmetic exception, the software will determine the exception that occurred in the current instruction by querying the indicator bit in the floating-point exception register, namely the Current bit, and process it in order according to the floating-point exception priority.

Floating-point arithmetic exceptions include:

- Input Denormalized Exception;
- Invalid Operation Exception;
- Divide by Zero Exception;
- Underflow Exception;
- Overflow Exception;
- Inexact Exception.

12.4.2 Abnormal Input of Non-standard Numbers

When a floating-point operand is a denormalized number, the IDC flag (FESR[5]) is set to 1.

The exception handling method is divided into two cases: exception enabled and exception not enabled.

12.4.2.1 Exception Enable

When the IDE bit is 1, the out-of-specification input exception is enabled. When the input operand is an out-of-specification number, the processor will terminate normal program execution and throw exception 30. At the same time, the FEC bit (FESR[7]) and IDC bit (FESR[5]) are set to 1 for query and processing by the exception service routine. The source register and destination register of the exception instruction will not be changed.

12.4.2.2 Exception Disable

When the IDE bit is 0, the out-of-specification input exception is disabled. The floating point unit will replace the out-of-specification number with an approximate number based on the FM bit.

If the FM bit is 1, the non-standard number will be flushed to a minimum floating point number, and the sign remains unchanged; if FM is 0, the non-standard number will be flushed to a minimum floating point number.

The number will be flushed to 0, with the same sign as before. The floating point unit will then use this approximate number to complete the instruction operation and write the result back to the register.

FEA (FESR[15]), IDA (FESR[13]), FEC (FESR[7]), and (IDCFESR[5]) will be set to 1.

12.4.3 Illegal Operation Exception

When an operand of an operation is illegal, an illegal operation exception will be triggered. Table 12.6 lists the operations and their corresponding operations that generate illegal operation exceptions .

The handling of illegal operation exceptions is divided into two cases: exception enabled and exception not enabled.

Table 12.6: Illegal Operation Exception

All	Illegal operation exception
floating point instructions (FABS, FNEG, FMOV, outside)	The operand contains sNaN
FADDS	(+INF) + (-INF) or (-INF) + (+INF)
FSUBS	(+INF) - (+INF) or (-INF) - (-INF)
FCMPHSS FCMPLTS FCMPZHSS FCMPZLSS	Any NaN operand
FSTOSI	Any NaN operand or result overflow
FSTOUI	Any NaN operand or result overflows or underflows
FMULS FNMULS	(Zero \times ±INF) or (\pm INF \times Zero) :sup:a
FDIVS	Zero/zero or INF/INF.a
FSQRTS	The operand is less than 0

Note: If the FM bit is 0 and the input operand is a denormalized number, the denormalized number will be treated as

0, it is also possible to meet the IO exception conditions.

12.4.3.1 Exception Enable

When the IOE bit (FCR[0]) is 1, the illegal operation exception is enabled. If an illegal operation exception occurs, the processor will terminate execution and generate a Generate exception 30 and set FEC (FESR[7]) and IOC (FESR[0]) to 1 for the exception service routine to query and process.

Neither the register nor the destination register will be changed.

12.4.3.2 Exception Disable

When the IOE bit is 0, floating-point instructions (except integer conversion operations) will generate qNaN as the result of the operation and write it back to the destination. Registers. FEA (FESR[15]), IOA (FESR[8]), FEC (FESR[7]), and IOC (FESR[0]) will be set to 1. For format

The default result of the conversion instruction when an illegal operation occurs is shown in Table 4-2.

Table 12.7: Default results for illegal conversion results

Command	Input value	Default output
FSTOSI NaN		0x00000000
	+INF or x231	0x7fff
	-INF or x-231	0x80000000
FSTOUI NaN		0x00000000
	+INF or x232	0xffff
	-INF or x<0	0x00000000

12.4.4 Division by Zero Exception

When the dividend is a non-zero number and the divisor is 0, a divide-by-zero exception occurs. 0, the divisor will be refreshed to 0, which will also cause a division by zero exception.

12.4.4.1 Exception Enable

If the DZE bit (FCR[1]) is set to 1, the divide-by-zero exception is enabled. If a divide-by-zero exception occurs, the processor will terminate program execution. Exception No. 30 is thrown. FEC (FESR[7]) and DZC (FESR[1]) will be set to 1 for the exception service routine to query and process. Neither the register nor the destination register will be changed.

12.4.4.2 Exception Disable

If the DZE bit is set to 0, the Division by Zero exception is not enabled. If this exception occurs, a Signed infinity. FEA (FESR[15]), DZA (FESR[9]), FEC (FESR[7]), and DZC (FESR[1]) are set to 1.

12.4.5 Overflow Exception

The floating point unit of the E804 will detect overflow exceptions after rounding. When the number is finite, an overflow exception will occur. Because when an overflow exception occurs, the result cannot be accurately represented, so overflow is always the same as Inexact exceptions are also raised.

12.4.5.1 Exception Enable

If the OFE bit (FCR[2]) is 1, the overflow exception is enabled. If the operation result overflows, the processor will terminate the program. The instruction is executed and exception 30 is thrown, and FEC (FESR[7]), IXC (FESR[4]), and OFC (FESR[2]) are set to 1.

Neither the register nor the destination register will be changed.

12.4.5.2 Exception Disable

If the OFE bit is 0, the overflow exception is not enabled. If this exception occurs, the floating-point unit takes a signed infinite number or a maximum finite number as the operation result and writes it to the destination register according to different rounding modes. FEA (FESR[15]), IXA (FESR[12]), OFA (FESR[10]), FEC (FESR[7]), IXC (FESR[4]), and OFC (FESR[2]) will be set to 1.

12.4.6 Underflow Exception

E804 The floating point unit will detect if an underflow exception occurs after rounding the result of an operation. If the result is less than the minimum finite number of the corresponding format, An Underflow exception will be generated. An Underflow exception is always generated at the same time as an Inexact exception.

12.4.6.1 Exception Enable

If the UFE bit (FCR[3]) is 1, the underflow exception is enabled. If the operation result has an underflow exception, the processor will terminate the program execution and throw exception 30, and set FEC (FESR[7]), IXC (FESR[4]), and UFC (FESR[3]) to 1. The source register and destination register of the instruction will not be changed.

12.4.6.2 Exception Disable

If the UFE bit is 0, the underflow exception is not enabled. If this exception occurs, when the FM bit is 1, the result will be flushed to the smallest finite number of the corresponding format with the same sign, otherwise the result will be a 0 with the same sign. At the same time, FEA (FESR[15]), IXA (FESR[12]), UFA (FESR[11]), FEC (FESR[7]), IXC (FESR[4]), and UFC (FESR[3]) will be set for software to query.

12.4.7 Imprecise Exceptions

The exponent field or mantissa field of the floating-point operation result will cause an inexact exception. For specific definitions, please refer to the floating-point protocol standard (ANSI/IEEE Std 754). The following situations will cause an imprecise exception:

- The results before and after rounding are inconsistent;
- An overflow exception occurs;
- An underflow exception occurs.

Note: Inaccurate exceptions frequently occur in floating-point operations. If the IXE bit is enabled, the floating-point operation performance will be greatly reduced. If the application itself does not require high precision, it is recommended to turn off the accurate exception enable bit.

12.4.7.1 Exception Enable

If the IXE bit (FCR[4]) is 1, the inexact exception is enabled. If an inexact exception is generated by a floating-point operation, the processor terminates program execution and throws exception 30, and sets FEC (FESR[7]) and IXC (FESR[4]) to 1 for the exception service routine to query. The source register and destination register of the instruction will not be changed.

12.4.7.2 Exception Disable

If the IXE bit is 0, the inexact exception is disabled. The floating point unit will round the floating point operation result according to the rounding mode and write the rounded result back to the destination register. FEA (FESR[15]), IXA (FESR[12]), FEC (FESR[7]), and IXC (FESR[4]) will be set to 1.

12.4.8 Exception Priority

During the operation of floating-point instructions, several exceptions may be generated at the same time. The exceptions that may be generated at the same time are as follows:

- Non-standard number input anomaly and other 5 types of anomalies;
- Overflow exception and inexact exception;
- Underflow exception and inexact exception.

Take a floating-point multiplication instruction as an example, one operand is a non-standard number, the other operand is a minimum standard number, the non-standard number input exception is not enabled (IDE is 0), and FM = 1. This instruction will generate three exceptions, namely, non-standard number input exception, underflow exception and inexact exception. The floating-point exception register will record all exceptions that occur during the floating-point instruction processing for software query and processing.

The priority relationship of floating-point exceptions is as follows (recommended for floating-point exception programs):

1. Abnormal input of non-standard numbers;
2. Illegal operation exception;
3. Division by zero exception;
4. Overflow/underflow exception;
5. Imprecise exceptions.

12.4.9 Exceptions to special instructions

For the multiply-accumulate/multiply-accumulate-subtract instructions, the floating-point unit considers them as two independent floating-point operations, and the floating-point exception status register records the multiplication and addition/multiplication and subtraction instructions.

All exceptions that occur during addition and subtraction operations are recorded to help the software record and handle exceptions to the greatest extent possible.

12.5 Instruction Set

12.5.1 Floating-point instruction list

This section mainly introduces the E804 floating-point instruction set. The floating-point instructions of E804 are all 32-bit instructions. According to the functions implemented by the instructions, they can be divided into for:

- Data operation instructions;
- Transmission instructions;
- Memory access instructions.

12.5.1.1 Data Operation Instructions

Table 12.8: List of single-precision data operation instructions

FSTOSI Convert single-precision floating-point number to signed integer	
FSTOUI	Converts a single-precision floating point number to an unsigned integer
FSITOS	Convert signed integer to single-precision floating point number
FUITOS	Convert unsigned integer to single-precision floating point number
FCMPZHSS	Single precision floating point greater than or equal to 0
FCMPZLSS	Single precision floating point less than or equal to 0
FCMPZNES	Single precision floating point not equal to 0
FCMPZUOS	Single precision floating point single operand is not a number
FCMPHSS	single-precision floating point greater than or equal to
FCMPLTS	Single-precision floating point less than
FCMPNES	Single precision floating point not equal
FCMPUOS	Checks whether the double operand is not a number.
FMOVIS	Single-precision floating-point immediate value generation
FMOVS	single-precision floating-point number transfer between floating-point registers
FABSS	single-precision floating point number absolute value
FNEGS	Negate a single-precision floating point number
FADDS	Single-precision floating-point addition
FSUBS	Single-precision floating-point subtraction
FMULS	Single-precision floating-point multiplication
FNMULS	Single-precision floating-point multiplication and negation
FMACS	single-precision floating-point multiply-accumulate
FMSCS	single-precision floating-point multiply-accumulate-subtract
FNMACS	Single-precision floating-point multiplication and negation accumulation
FNMSCS	Single-precision floating-point multiplication, inverse subtraction,
FDIVS	single-precision floating-point division
FSQRTS	Single-precision floating-point number square root
FRECIPS	single-precision floating point number reciprocal

12.5.1.2 Transfer instructions

Table 12.9: Data transfer instruction list

FMTVRL floating point register word write transfer instruction	
FMFVRL Floating point register word read transfer instruction	

12.5.1.3 Memory Access Instructions

Table 12.10: List of memory access instructions

FLDS single precision floating point load instruction	
FLDRS Register shift addressing single precision floating point load instruction	
FLDMS Continuous single-precision floating-point load instruction	
FLRWS Single precision floating point immediate memory read instruction	
FSTS single-precision floating-point store instruction	
FSTRS Register shift addressing single precision floating point storage instruction	
FSTMS Continuous single-precision floating-point store instruction	

12.5.2 Floating-point instruction execution delay

The E804 floating-point unit has a strict design for the execution delay of instructions, which reduces the floating-point delay while meeting the requirements of low cost and low power consumption.

Instruction execution delay: Software developers can use instruction delay information to adjust and optimize the software to improve the execution efficiency of floating-point programs.

Table 12.11: Floating-point instruction execution delay list

	Execution cycle	Remark
Instruction data operation instruction		
FSTOSI	1	-
FSTOUI	1	-
FSITOS	1	-
FUITOS	1	-
FCMPZHSS	1	-
FCMPZLSS	1	-
FCMPZNES	1	-
FCMPZUOS	1	-
FCMPHSS	1	-
FCMPLTS	1	-
FCMPNES	1	-
FCMPUOS	1	-

Continue on next page

Table 12.11 – Continued from previous page

FMOVIS	1	-
FMOVS	1	-
FABSS	1	-
FNEGS	1	-
FADDS	2	-
FSUBS	2	-
FMULS	2	-
FNMULS	2	-
FMACS	4	Split into FMULS and FADDS instructions.
FMSCS	4	Split into FMULS and FSUBS instructions.
FNMACS	4	Split into FNMULS and FADDS instructions.
FNMSCS	4	Split into FNMULS and FSUBS instructions.
FDIVS	2-28	-
FSQRTS	2-28	-
FRECIPS	2-28	-
Transmission instructions		
FMTVRL	1	-
FMFVRL	1	-
Memory access instructions		
FLDS	1	-
FLDRS	3	Split into IXW and FLDS instructions.
FLDMS	N	Split into N FLDS instructions.
FLRWS	1	-
FSTS	1	-
FSTRS	3	Split into IXW and FSTS instructions.
FSTMS	N	Split into N FSTS instructions.

Note: Indicates memory access related instructions. The instruction completion cycle depends on the bus delay or cache hit situation. The values listed in the table are the fastest Condition.*

Chapter 13 Initialization Reference Code

13.1 MPU Setting Example

```

/* Set the attributes of area 0: non-executable, non-cacheable, readable and writable by superusers and ordinary users.

mfcr r10,cr<19,0> bclri
r10,0 //Set the control register bit 0 to 0, and area 0 is not executable. bseti r10,8 //
Set the control register bits 8 and 9 to 1, and area 0 is readable and writable by both super users and ordinary users.
bseti r10,9 bclri
r10,24 //Set the 24th bit of the control register to 0, and area 0 cannot be cached.
mtcr r10,cr<19,0>

/* Set area 1 attributes, executable, cacheable, readable and writable by superusers and ordinary users

mfcr r10,cr<19,0> bseti
r10,1 //Set the control register bit 1 to 1, and area 1 is executable bseti r10,10 //
Set the control register bits 10 and 11 to 1, and area 1 is readable/writable by both super users and ordinary users.
bseti r10,11
bseti r10,25 //Set the 25th bit of the control register to 1, and area 1 can be cached.
mtcr r10,cr<19,0>

/* Set the properties of area 2: executable, non-cacheable, readable and writable by superusers, and read-only by ordinary users.

mfcr r10,cr<19,0> bseti
r10,2 //Set the second bit of the control register to 1, and area 2 is executable.
bclri r10,12 //Set bit 12 to 0, bit 13 to 1, the super user can read and write, and ordinary users can read only.
bseti r10,13
bclri r10,26 //Set the 26th bit of the control register to 0, and area 2 cannot be cached.
mtcr r10,cr<19,0>

/* Set the attributes of area 3: non-executable, cacheable, readable and writable by both superusers and ordinary users.

```

(Continued on next page)

(Continued from previous page)

```

mfcr r10,cr<19,0> bclri r10,3 //  

Set the third bit of the control register to 0, and area 3 is not executable. bseti r10,14 //Set bits  

14 and 15 to 1, and area 3 is readable and writable by both super users and ordinary users.  

bclri r10,15  

bseti r10,27 //Set the 27th bit of the control register to 1, and area 3 can be cached.  

mtcr r10,cr<19,0>  
  

/* The attribute settings of regions 4~7 are the same as those of regions 0, 1, and 2.  
  

// The executable attribute of area 0~7, set the [7:0] bits of the control register CR<19,0>.  
  

//Access rights for area 0~7, set bits [23:12] of control register CR<19,0>.  
  

// The cacheable permissions of regions 0~7 are set by setting bits [31:24] of control register CR<19,0>.  
  

/* Set protection area 0, base address and protection area size.  
  

movi r10,0 mtcr  

r10,cr<21,0> //Select protection area 0. movi r10,0x0 //Set  

the base address of the protection area to 0x00000000. ori r10,r10,0x3f //  

Set the size of the protection area to 4G. mtcr r10,cr<20,0> //  

Set the base address of the protection area to 0 and the size of the protection area to 4G.  
  

/* Set protection area 1, base address and protection area size.  
  

movi r10,1  

mtcr r10,cr<21,0> //Select protection area 1. movih  

r10,0x2800 //Set the base address of the protection area to 0x28000000. ori  

r10,r10,0x2f //Set the size of the protection area to 16M. mtcr  

r10,cr<20,0> //Set the address range of the protection area to 0x28000000 ~ 0x29000000.  
  

/* Set protection area 2, base address and protection area size.  
  

movi r10,2 mtcr  

r10,cr<21,0> //Select protection area 2. movih r10,0x2800 //  

Set the base address of the protection area to 0x28000000. ori r10,r10,0x27 //  

Set the size of the protection area to 1M. mtcr r10,cr<20,0> //  

Set the address range of the protection area to 0x28000000 ~ 0x28100000

```

(Continued on next page)

(Continued from previous page)

```

/* Set protection area 3, base address and protection area size.

    movi r10,3
    mtcr r10,cr<21,0> //Select protection area 2. movih
    r10,0x28f0 //Set the base address of the protection area to 0x28f00000. ori
    r10,r10,0x27 //Set the size of the protection area to 1M. mtcr
    r10,cr<20,0> //Set the address range of the protection area to 0x28f00000 ~ 0x29000000.

    //The base address and protection area size settings of protection areas 3~7 are the same as those of areas 0, 1, and 2.

    //Set the control register CR<21,0> to select the protection area.

    //Write the base address and size of the protection zone into the control register CR<20,0>.

/* Enable MPU

    mfcr r7, cr<18,0> //Select MUP enable control register. bseti r7, 0 //Set the
    lowest two bits of control register CR<18,0> to 2'b01, enable MPU.
    bclri r7, 1 mtcr
    r7, cr<18,0> //Write the preset value into the MPU enable control register to turn on the MPU.

```

13.2 Cache Setting Example

Before enabling the cache, you need to invalidate the entire cache, and then configure CER and enable CACHE according to actual application requirements.

```

// Cache invalidation.

    lrw r1, 1 // Preset bit 0 INV_ALL to 1. lrw r2, 0xe000f004 // Preset
    the address of CIR. st.w r1, (r2) // Write the preset value to CIR, and
    the invalidation operation begins.

// Set up cacheable area crcrc0.

    lrw r1, 0x00000039 // Preset the 512M address space with the starting address of 0x0 to be cacheable. lrw r2,
    0xe000f008 // Preset the address of CRCR0. st.w r1, (r2) // Write the preset
    value to CIR, and the invalidation operation begins.

//The settings of the bufferable area crcrc1~3 are the same as crcrc0, and the cacheable start address, size and enable bit are written separately.

```

(Continued on next page)

(Continued from previous page)

```
//crcr1 address 0xe000f00c,
//crcr2 address 0xe000f010
//crcr3 address 0xe000f014

// Enable cache

lw r1, 0x0 // The preset value is 0x0000000.
bseti r1, 0 // Set Cache enable to open. bclri r1,
1 // Set, both instructions and data can be cached.
bseti r1, 2 // Set Cache to write back mode. bseti r1, 4 //
Set Cache to be writable. bclri r1, 5 //
Set Cache to close write allocate. lw r2, 0xe000f000 //
The preset address of CER. st.w r1, (r2) // Write the
preset value to CER, cache enable.
```

//At this time, the cache is writable and in write-back mode, with four-way groups connected, instruction and data caches, and the write allocate function is disabled.

13.3 Interrupt Enable Initialization

After configuring the interrupt controller and interrupt vector table (refer to Tightly Coupled [IP](#) for details), you need to turn on the interrupt enable bit. The specific settings are as follows:

```
psrset ee,ie
```

13.4 General Register Initialization Example

```
//Initialize general registers r0~r15 and r28 to 0.

movi r0, 0 //Initialize general register 0 to 0.
movi r1,0
movi r2,0
movi r3,0
movi r4,0
movi r5,0
movi r6,0
movi r7,0
```

(Continued on next page)

(Continued from previous page)

```
movi r8,0
movi r9,0
movi r10,0
movi r11,0
movi r12,0
movi r13,0
movi r14,0
movi r15,0
movi r28,0
```

13.5 Stack Pointer Initialization Example

The setting of the stack pointer is related to the current state of the program as shown below. In the supervisor state, the stack pointer is initialized to Example 1. In the user state, the stack pointer is initialized to Example 2.

The lower stack pointer is initialized as in Example 2.

Example 1:

```
//In superuser mode, set the superuser mode and user mode stack pointers

// In superuser mode, r14 is mapped to the stack pointer register of superuser mode

//In super user mode, the user mode stack pointer register is CR<14,1>

lwz r14, 0x01000000 //Set the super user state pointer lwz r0,
0x02000000 // mtcr r0, cr<14,1> //

Set the user state stack pointer
```

Example 2:

```
//In user mode, only user mode pointers can be set:

// In user mode, r14 is mapped to the stack pointer register in user mode

lwz r14, 0x02000000 //Set user mode pointer
```

13.6 Example of setting the entry address of exception and interrupt service routines

The entry address setting of exception and interrupt service routines is divided into two steps:

Step 1: Set the exception vector table address register VBR. According to the vector number, each exception vector number corresponds to the exception vector table address, VBR + (vector number << 2);

Step 1: Write the entry address of the exception service program to the exception vector table address corresponding to the exception vector number in Step 1.

An example of an access error exception with exception vector number 2 is as follows:

```
//Set the exception vector table entry address, VBR corresponds to the control register cr<1,0>

Lrw r2, 0 mtcr r2,
cr<1,0> //Set the address of the exception vector table to 0

//Exception/interrupt service program entry address setting

lw r1, ACCERR_ERROR_BEGIN //Set the exception service program entry address lw r2, 0 //VBR
address movi r3, 0x2 //Exception vector
number
lsli r3, r3, 2 addu r2, r2, r3 //

Calculate the exception vector number corresponding to the exception vector table address st.w
r1, (r2, 0x0) //Store the exception service program entry address in the corresponding exception vector table address
br START

//Access error exception service program set by the user

label ACCERR_ERROR_BEGIN

/* User's exception service program */

label ACCERR_ERROR_END
label START
```

13.7 FPU Initialization Example

When initializing floating-point operations, you need to set the rounding mode of floating-point operations, the processing of abnormal data, etc.

```
// Clear floating point exception status handling register

lw r1,0x0 mtcr
r1,cr<2,2> //Clear the exception flag register.

//Set the floating point control register
```

(Continued on next page)

(Continued from previous page)

```
lw r1,0x0 bclri  
r1,27 //Set the processing mode of denormal numbers, treat them as 0. bclri  
r1,25 //Set bits 25 and 24 to 0, rounding mode round to nearest even. bclri r1,24 // bclri r1,5 //Set denormal  
numbers, do not trigger  
exceptions. bclri r1,4 //Set the result to be inaccurate, do not  
enter the exception, and process it according to the rounding mode. bclri r1,3 //Set the result to  
overflow, do not trigger an exception, and process it according to the rounding mode. bclri r1,2 //Set  
the result to overflow, do not trigger an exception, and process it according to the rounding mode.  
bclri r1,1 //Set the division by 0 operation to trigger an exception, do not write back the result, and jump to the  
division by 0 exception. bclri r1,0 //Set if NaN appears in the input and output, do not trigger an  
exception, and output QNaN. mtc r1,cr<1,2> //Write the prefabricated value to the floating point control register FCR.
```

```
//At this time, the control state of FPU is that it switches to exception handling only when a division by 0 exception occurs, and does not trigger exceptions when  
other exceptions occur. //Non-standard numbers are treated as 0, and the rounding mode is round to nearest even.
```

Chapter 14 Appendix A Basic Command Glossary

The following is a detailed description of each E804-implemented Xuantie V2 instruction. Each instruction is described in alphabetical order.

bright.

Each Black Iron V2 instruction mnemonic ends with the number "32" or "16" to indicate the instruction bit width. For example, "addc32" means that the instruction is 32 "addc16" indicates that the instruction is a 16-bit unsigned addition instruction with carry.

If you omit the instruction width in the mnemonic (such as "addc"), the system will automatically assemble it into the most optimized instruction.

Among them, the instructions with # in the Chinese name are pseudo instructions.

14.1 ABS—Absolute value instruction

Unified instructions	
Syntax	abs rZ, rx
Operation	RZ \leftarrow RX
The compiled result	contains only 32-bit instructions. abs32 rZ, rx
illustrate	Take the absolute value of the RX value and store the result in RZ. Note that the result of operand 0x80000000 is 80000000.
Affects flag bit	No effect
abnormal	none

32-bit instructions	
Operation	RZ \leftarrow RX
Syntax	abs32 rZ, rx
Description	Take the absolute value of the RX value and store the result in RZ. Note that the result of operand 0x80000000 is 80000000.
Affects flag bit	No effect
Abnormal	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	00000	RX	000000	10000	RZ

14.2 ADDC——Unsigned Add with Carry Instruction

Unified instruction		
	addc rz, rx	addc rz, rx, ry
	RZ \ddot{y} RX + C, C \ddot{y} carry is compiled into	RZ \ddot{y} RX + RY + C, C \ddot{y} carry is compiled into
syntax operation compilation results	the corresponding 16-bit or 32-bit instruction according to the range of the register. if ($x < 16$) and ($z < 16$), then addc16 else addc32 rz, rz, rx; Add the	the corresponding 16-bit or 32-bit instruction according to the range of the register. if ($y == z$) and ($x < 16$) and ($z < 16$), then addc16 rz, rx; else addc32 rz, rx, ry;
	values of RZ/RY, RX and C bit, and store the result in RZ, and the carry in C bit.	
	C \ddot{y} Carry	
Description:	Affects the flag bit abnormality	

16-bit	
	RZ \ddot{y} RX + C, C \ddot{y} carry addc16 rz, rx Add the
	values of RZ, RX
	and C, and store the result in RZ, with the carry in C.
instruction operation syntax	
	The register range is r0-r15.
description	Affected flag bit C \ddot{y} Carry limit exception

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11000		RZ		RX	0	1	

Figure 14.1: ADDC-1

32-bit	
	RZ \leftarrow RX + RY + C, C \leftarrow carry addc32 rz,
	rx, ry Add the values
	of RX, RY and C and store the result in RZ, with the carry in C.
instruction operation syntax	
description	Affected flag bit C \leftarrow Carry exception

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RY	RX	000000	00010	RZ

Figure 14.2: ADDC-2

14.3 ADDI—Unsigned Immediate Addition Instruction

Unified command			
language	addi rz, oimm12	addi rz, rx, oimm12	addi rz, r28, oimm18
Law			
	RZ \leftarrow RZ + zero_extend(OIMM12) compiles to	RZ \leftarrow RX + zero_extend(OIMM12) compiles to	RZ \leftarrow R28 + zero_extend(OIMM18)
Operation	the corresponding 16-bit or 32-bit instruction based on the range of the immediate value and register. if compilation results (z<8) and (oimm12<257), addi16 rz, oimm8; else addi32 rz, rz, oimm12;		
	the corresponding 16-bit or 32-bit instruction based on the range of the immediate value and register. if (z<8) and (oimm12<257), addi16 rz, oimm8; else addi32 rz, rz, oimm12;		
	Compile to the corresponding 16-bit or 32-bit instruction according to the range of the immediate value and register. if (oimm12<9) and (z<8) and (x<8), addi16 rz, rx, oimm3; elseif (oimm12<257) and (x==z) and (z<8), addi16 rz, oimm8; else addi32 rz, rx, oimm12;		
	The immediate value with offset 1 is zero-extended to 32 bits and then added to the value of RX/RZ and the result is stored in RZ.		
	No impact		
Description	If the source register is R28, the range of the immediate value is 0x1-0x40000. If the target register is not R28, the range of the immediate value is 0x1-0x1000.		

16-bit instruction	
1 Operation	RZ \leftarrow RZ + zero_extend(OIMM8) Syntax addi16
rz, oimm8	Description Zero-extend
	the 8-bit immediate value (OIMM8) with offset 1 to 32 bits, then add it to the value of RZ and store the result in RZ.
	NOTE: Binary operand IMM8 is equal to OIMM8 - 1. No effect
The range of the	
affected flag limit	register is r0-r7; the range of the immediate value is 1-256 .

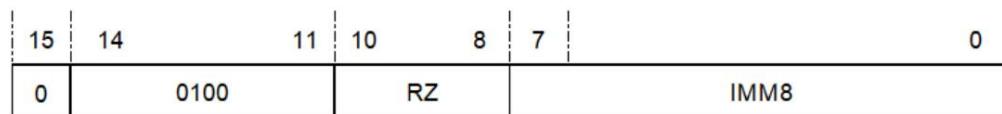
16-bit instruction format1:

Figure 14.3: ADDI-1

IMM8 domain:

Specifies the value of an immediate value without an offset.

Note: The value added to register OIMM8 must be offset by 1 compared to the binary operand IMM8.

00000000:

Add 1

00000001:

Add 2

...

11111111:

Add 256

16-bit instruction	
2 Operation	RZ \leftarrow RX + zero_extend(OIMM3) Syntax addi16
rz, rx, oimm3	Description Zero-extend the
	3-bit immediate value (OIMM3) with offset 1 to 32 bits, then add it to the value in RX and store the result in RZ.
	NOTE: Binary operand IMM3 is equal to OIMM3 - 1. No effect
The range of the	
affected flag limit	register is r0-r7; the range of the immediate value is 1-8 .

16-bit instruction format 2:

15	14	11	10	8	7	5	4	2	1	0
0	1011	RX	RZ	IMM3	1	0				

Figure 14.4: ADDI-2

IMM3 domain:

Specifies the value of an immediate value without an offset.

Note: The value added to register OIMM3 must be offset by 1 compared to the binary operand IMM3.

000:

Add 1

001:

Add 2

...

111:

Add 8

32-bit instruction	
1 Operation	RZ \leftarrow RX + zero_extend(OIMM12) Syntax addi32 rz,
rx, oimm12	Description Zero-extend the 12-bit immediate value (OIMM12) with offset 1 to 32 bits, then add it to the value in RX and store the result in RZ.
	NOTE: Binary operand IMM12 is equal to OIMM12 - 1. No effect
The flag bit	
affects	the immediate value range to 0x1-0x1000.

32-bit instruction format1:

31	30	26	25	21	20	16	15	12	11	0
1	11001	RZ	RX	0000						IMM12

Figure 14.5: ADDI-3

IMM12 domain:

Specifies the value of an immediate value without an offset.

Note: The value added to register OIMM12 must be offset by 1 compared to the binary operand IMM12.

000000000000:

Add 0x1

000000000001:

Add 0x2

...

111111111111:

Add 0x1000

32-bit	
instruction 2 Operation RZ + R28 + zero_extend(OIMM18)	
Syntax	addi32 rz, r28, oimm18
Description	Zero-extend the 18-bit immediate value (OIMM18) with offset 1 to 32 bits, then add it to the value in R28 and store the result in RZ. NOTE: Binary operand IMM18 is equal to OIMM18 - 1. No
The flag bit	effect
affects	the immediate value range to 0x1-0x40000.

32-bit instruction format 2:

Figure 14.6: ADDI-4

IMM18 domain:

Specifies the value of an immediate value without an offset.

Note: The value added to register OIMM18 must be offset by 1 compared to the binary operand IMM18.

0000000000000000:

Add 0x1

0000000000000001:

Add 0x2

...

1111111111111111:

Add 0x40000

14.4 ADDI(SP) – Unsigned (Stack Pointer) Immediate Addition Instruction

The unified		
	addi rZ, sp, imm	addi sp, sp, imm SP
instruction syntax	operation RZ + SP + zero_extend(IMM)	
compilation result	only has 16-bit instructions. addi rZ, sp, imm Zero-	
illustrate	extends the immediate value (IMM) to 32 bits, adds it to the value of the stack pointer (SP), and stores the result in RZ or SP.	
Affects flag bit	No impact	
	The register range is r0-r7; the immediate range is 0x0-0x3fc.	
restriction exception		

16-bit instruction	
1 Operation	RZ + SP + zero_extend(IMM) Syntax addi16 rZ,
sp, imm8	Description Zero-extend the
	immediate value (IMM) to 32 bits, then add it to the value of the stack pointer (SP) and store the result in RZ. Note: The immediate value (IMM) is equal to the binary operand IMM8 << 2. No effect
The range of the	
affected flag limit	register is r0-r7; the range of the immediate value is (0x0-0xf) << 2 .

16-bit instruction format1:

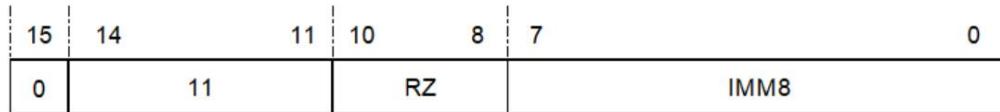


Figure 14.7: ADDI(SP)-1

IMM8 domain:

Specifies the value without the shift immediate.

Note: The value added to register IMM is shifted left 2 bits compared to the binary operand IMM8.

00000000:

Add 0x0

00000001:

Add 0x4

...

11111111:

Add 0x3fc

16-bit	
instruction 2 Operation	SP \leftarrow SP + zero_extend(IMM)
Syntax	addi16 sp, sp, imm
Description	Zero-extend the immediate value (IMM) to 32 bits, then add it to the value of the stack pointer (SP) and store the result in RZ. Note: The immediate value (IMM) is equal to the binary operand {IMM2, IMM5} $\ll 2$.
The source	No effect
and destination registers of the affected flag bit limit	are both the stack pointer register (R14); the range of the immediate value is $(0x0\text{-}0x7f) \ll 2$.

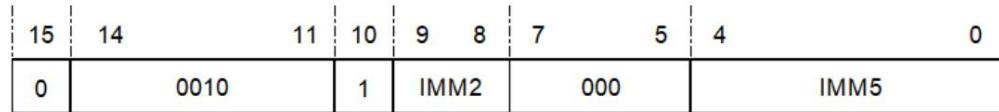
16-bit instruction format 2:

Figure 14.8: ADDI(SP)-2

IMM Domain:

Specifies the value without the shift immediate.

Note: The value added to register IMM is shifted left 2 bits compared to the binary operand {IMM2, IMM5}.

{00, 00000}

Add 0x0

{00, 00001}

Add 0x4

...

{11, 11111}

Add 0x1fc

14.5 ADDU - Unsigned Addition Instruction

Unified instruction		
	addu rz, rx	addu rz, rx, ry
	RZ $\ddot{+}$ RZ + RX	RZ $\ddot{+}$ RX + RY
syntax operation compilation results	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (z<8) and (x<8) and (y<8), then addu16 else addu32 rz, rx, ry;	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (z<8) and (x<8) and (y<8), then addu16 rz, rx, ry; elseif (y==z) and (x<16) and (z<16), then addu16 rz, rx; else addu32 rz, rx, ry; Add the
	value of RZ/RY to the value of RX and store the result in RZ.	
	No impact	
Description:	Affects the flag bit abnormality	

16-bit instruction	
1	RZ $\ddot{+}$ RZ + RX
Operation: Syntax	addu16 rz, rx Add the
	value of RZ and RX and store the result in RZ. Description:
Affected flags:	None Restricted
register range is r0-r15. Exceptions	
	none

16-bit instruction format1:

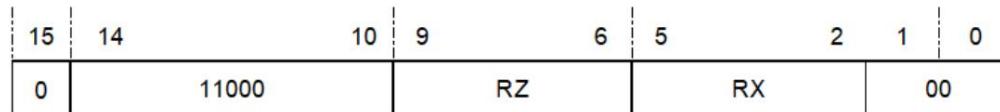


Figure 14.9: ADDU-1

16-bit instructions	
	RZ \ddot{y} RX + RY
	addu16 rz, rx, ry
	Add the values of RX and RY and store the result in RZ.
Operation Syntax	Description Impact Flags No Impact
Limit	registers to the range r0-r7.
abnormal	none

16-bit instruction format 2:

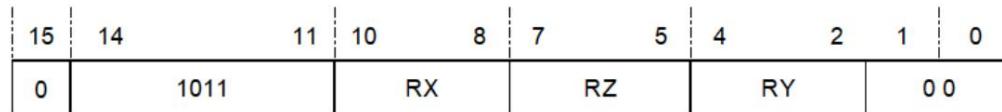


Figure 14.10: ADDU-2

32-bit instructions	
	RZ \ddot{y} RX + RY
	addu32 rz, rx, ry
	Add the values of RX and RY and store the result in RZ.
Operation Syntax	Description Impact Flags No Impact
Abnormal	

32-bit instruction format:

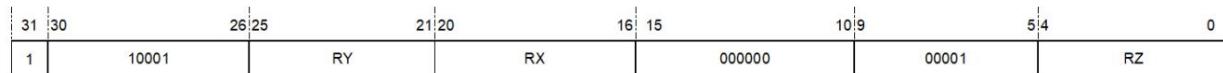


Figure 14.11: ADDU-3

14.6 AND——Bitwise AND Instruction

Unified instruction		
	and rz, rx	and rz, rx, ry
	RZ $\ddot{\wedge}$ RZ and RX	RZ $\ddot{\wedge}$ RX and RY
syntax operation and compilation results	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 16$) and ($z < 16$), then else and32 rz, rz, rx; Perform	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($y == z$) and ($x < 16$) and ($z < 16$), then and16 rz, rx; else and32 rz, rx, ry;
	bitwise AND operation on the value of RZ/RY and RX, and store the result in RZ.	
	No impact	
	none	
Description: Affects the flag bit abnormality		

16-bit	
	RZ $\ddot{\wedge}$ RZ and RX
	and16 rz, rx
	Perform bitwise AND of the values in RZ and RX and store the result in RZ.
instruction operation	syntax
	The register range is r0-r15.
description affected	flags no impact limit exception

16-bit instruction format:

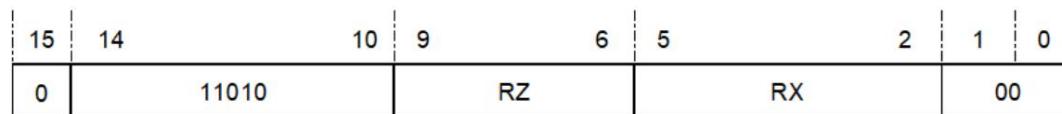


Figure 14.12: AND-1

32-bit instructions	
	RZ \leftarrow RX and RY
	and32 rz, rx, ry
	Perform bitwise AND of the values in RX and RY and store the result in RZ.
Operation Syntax Description Impact Flags No Impact	
abnormal	none

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10001	RY	RX		001000	00001	RZ

Figure 14.13: AND-2

14.7 ANDI—Bitwise AND Immediate Value Instruction

Unified instructions	
Syntax	andi rz, rx, imm16
Operations	RZ \leftarrow RX and zero_extend(imm12)
The compiled result	contains only 32-bit instructions. andi32 rz, rx, imm12 Zero-extend the 12-bit immediate value to 32 bits, then bitwise AND it with the value in RX and store the result in RZ.
Description: The impact flag	has no impact
Limit	the range of immediate values to 0x0-0xFFFF.
Abnormal	

32-bit instructions	
	RZ \leftarrow RX and zero_extend(imm12)
	andi32 rz, rx, imm12
	Zero-extend the 12-bit immediate value to 32 bits, then bitwise AND it with the value in RX and store the result in RZ.
Operation Syntax Description Impact Flags No Impact	
Limit	the range of immediate values to 0x0-0xFFFF.
Abnormal	

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	11001		RZ		RX		0010			IMM 12

Figure 14.14: ANDI

14.8 ANDN——Bitwise AND Instruction

Unified instruction		
	andn rZ, rx	andn rZ, rx, ry
	RZ ∽ RZ and (!RX)	RZ ∽ RX and (!RY)
syntax operation	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (z<16), then andn16 else andn32 rZ, rZ, rx; For andn	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x==z) and (y<16) and (z<16), then andn16 rZ, ry; else andn32 rZ, rZ, rx;
	rz, rx, perform a bitwise AND operation on the value of RZ and the negative value of RX, and store the result in RZ; For andn rZ, rx, ry, perform a bitwise AND operation on the value of RX and the negative value of RY, and store the result in	
	RZ. No effect	
Description:	Affects the flag bit abnormality	

16-bit	
	RZ ∽ RZ and (!RX) andn16 rZ, rx
	Perform a bitwise
	AND of the value of RZ and the negated value of RX, and store the result in RZ.
instruction operation syntax	
Description Affects flags	None Restrictions The range of
registers is r0-r15. Exceptions	

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11010		RZ		RX		01	

Figure 14.15: ANDN-1

32-bit instructions	
	RZ \bar{y} RX and (IRY)
	andn32 rz, rx, ry
	Perform a bitwise AND operation on the value of RX and the negated value of RY, and store the result in RZ.
Operation Syntax Description Impact Flags No Impact	
Abnormal	

32-bit instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10001		RY		RX		001000		00010		RZ	

Figure 14.16: ANDN-2

14.9 ANDNI—Bitwise AND Immediate Instruction

Unified instructions	
Syntax	andni rz, rx, imm16
Operation	RZ \bar{y} RX and !(zero_extend(imm12))
The compiled result	contains only 32-bit instructions. andni32 rz, rx, imm12
	Zero-extend the 12-bit immediate value to 32 bits and negate it, then perform a bitwise AND operation with the value in RX and store the result in RZ.
Description: The impact flag	has no impact
Limit	the range of immediate values to 0x0-0xFFFF.
abnormal	none

32-bit instructions	
	RZ \bar{y} RX and !(zero_extend(imm12))
	andni32 rz, rx, imm16
	Zero-extend the 12-bit immediate value to 32 bits and negate it, then perform a bitwise AND operation with the value in RX and store the result in RZ.
Operation Syntax Description Impact Flags No Impact	
Limit	the range of immediate values to 0x0-0xFFFF.
abnormal	none

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	11001	RZ		RX		0011		IMM12		

Figure 14.17: ANDNI

14.10 ASR—Arithmetic Shift Right Instruction

Unified instruction		
	asr rz, rx	asr rz, rx, ry
	RZ $\ddot{\vee}$ RZ >>> RX[5:0]	RZ $\ddot{\vee}$ RX >>> RY[5:0]
syntax operator <small>asr16 compilation results</small>	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x<16$) and ($z<16$), then else asr32 rz, rz, rx; asr32 rz, rx, ry; For asr rz, rx, perform arithmetic right shift on the value of RZ (shift the original value right, and shift a copy of the original sign bit on the left), and store the result in RZ. The number of right shifts is determined by the value of the lower 6 bits of RX (RX[5:0]); if the value of RX[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of the original value of RZ; For asr rz, rx, ry, perform arithmetic right shift on the value of RX (shift the original value right, and shift a copy of the original sign bit on the left), and store the result in RZ. The number of right shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of RX. No effect	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x==z$) and ($y<16$) and ($z<16$), then asr16 rz, ry; else asr32 rz, rz, rx; asr32 rz, rx, ry; For asr rz, rx, perform arithmetic right shift on the value of RZ (shift the original value right, and shift a copy of the original sign bit on the left), and store the result in RZ. The number of right shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of RY. No effect
explain bright	original value right, and shift a copy of the original sign bit on the left), and store the result in RZ. The number of right shifts is determined by the value of the lower 6 bits of RX (RX[5:0]); if the value of RX[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of the original value of RZ; For asr rz, rx, ry, perform arithmetic right shift on the value of RX (shift the original value right, and shift a copy of the original sign bit on the left), and store the result in RZ. The number of right shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of RY. No effect	none
Affects flag bit exception		

16-bit instruction	
operation	RZ $\ddot{\vee}$ RZ >> RX[5:0] Syntax asr16 rz, rx
Description	Perform arithmetic right shift on the value of RZ (the original value is shifted right, and a copy of the original sign bit is shifted into the left side), and the result is stored in RZ. The number of right shifts is determined by the value of the lower 6 bits of RX (RX[5:0]); if the value of RX[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of the original
The range of	value of RZ. No effect
the affected flag	limit register is r0- r15 .

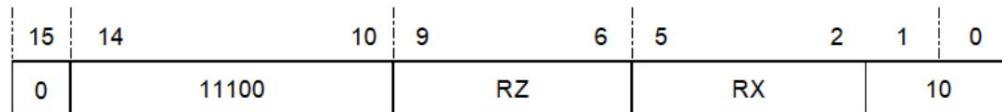
16-bit instruction format:

Figure 14.18: ASR-1

32-bit	
instruction operation	RZ $\ddot{\vee}$ RX >> RY[5:0] Syntax
asr32 rz, rx, ry	Description Perform arithmetic right shift on the value of RX (the original value is shifted right, and a copy of the original sign bit is shifted into the left side), and the result is stored in RZ. The number of right shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 30, then the value of RZ (0 or -1) is determined by the sign bit of RX. No effect
Affects flag bit	the sign bit of RX. No effect
abnormality	

32-bit instruction format:

Figure 14.19: ASR-2

14.11 ASRC - Immediate Arithmetic Shift Right to C Instruction

Unified instruction	
language	asrc rz, rx, oimm5 method
	RZ \leftarrow RX >>> OIMM5, C \leftarrow RX[OIMM5 - 1] Only 32-bit
	instructions exist: asrc32 rz, rx, oimm5
operation <small>combination result description</small>	Perform arithmetic right shift on the value of RX (the original value is shifted right, and a copy of the original sign bit is shifted in on the left), store the last bit bit C, and store the shift result in RZ. The right shift bit is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then the condition bit C is the sign bit (highest bit) of RX, and the value of RZ (0 or -1) is determined by the sign bit of RX.
	C \leftarrow RX[OIMM5 - 1]
	The range of immediate value is 1-32.
Affects flag bit restriction exception	none

32-bit instruction	
	RZ \leftarrow RX >>> OIMM5, C \leftarrow RX[OIMM5 - 1]
	asrc32 rz, rx, oimm5
operation syntax exception	Shift the value of RX right arithmetic (the original value is shifted right, and a copy of the original sign bit is shifted in on the left), store the last bit shifted , bit C, and store the shift result in RZ. The right shift bit is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then condition bit C is the sign bit (most significant bit) of RX, and the value of RZ (0 or -1) is determined by the sign bit of RX. Note: The binary operand IMM5 is equal to OIMM5 - 1.
	C \leftarrow RX[OIMM5 - 1]
	The range of immediate value is 1-32.
	none
Affects flag bit restriction exception	

32-bit instruction format:

31	30	25	25	21	20	16	15	10	9	5	4	0
1		10001		IMM5		RX		010011		00100		RZ

Figure 14.20: ASRC

IMM5 domain:

Specifies the value of an immediate value without an offset.

Note: The shift value OIMM5 must be offset by 1 compared to the binary operand IMM5.

00000:

Shift 1 bit

00001:

Shift 2 places

...

11111:

Shift 32 bits

14.12 ASRI - Arithmetic Shift Right Immediate Instruction

Unified	
instruction syntax	asri rz, rx, imm5
operation	RZ \leftarrow RX >>> IMM5 compilation
result	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 8$) and ($z < 8$), then asri16 rz, rx, imm5; else asri32 rz, rx, imm5;
Description	For asri rz, rx, imm5, perform arithmetic right shift on the value of RX (the original value is shifted right, and a copy of the original sign bit is shifted into the left side), and the result is stored in RZ. The number of right shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is equal to 0, then the value of RZ will be the same
Affects	as RX. No effect
flag bit	
abnormality	

16-bit instruction	
operation	RZ \leftarrow RX >>> IMM5 Syntax asri16 rz, rx,
imm5 Description	Perform an arithmetic right
	shift on the value of RX (the original value is shifted right, and a copy of the original sign bit is shifted into the left side). The result is stored in RZ. The number of right shifts is determined by the value of the 5-bit immediate value (IMM5) is determined; if the value of IMM5 is equal to 0, the value of RZ will remain unchanged.
The range of the	No effect
affected flag limit	register is r0-r7; the range of the immediate value is 0-31 .

16-bit instruction format:

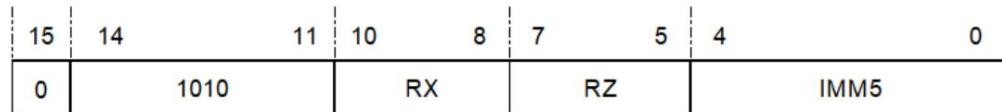


Figure 14.21: ASRI-1

32-bit instruction	
operation	RZ \leftarrow RX >>> IMM5 Syntax asri32 rz, rx,
imm5 Description	Perform arithmetic right shift on the value of RX (the original value is shifted right, and a copy of the original sign bit is shifted into the left side), and the result is stored in RZ. The number of right shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is equal to 0, then the value of RZ will be the same
The flag bit	as RX. No effect
affects	immediate value range to 0-31.

32-bit instruction format:

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10001	IMM5	RX	010010	00100	RZ	

Figure 14.22: ASRI-2

14.13 BCLRI—Digital Clear Immediately Instruction

Unified instruction		
	bclri rz, imm5	bclri rz, rx, imm5
	RZ \leftarrow RZ[IMM5] clear	RZ \leftarrow RX[IMM5]
syntax operation	(z == x) ? bclri16 rz, imm5; else bclri32 rz, rz, imm5; bclri32 rz, rx, imm5; Clear the bit indicated by the IMM5 field value in the value of RZ/	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x==z) and (z<8), then bclri16 rz, imm5; else
	RX, keep the other bits unchanged, and store the cleared result in RZ.	
	No impact	
Description	The range of immediate values is 0-31. Affects flag bit restrictions	

16-bit instructions	
	RZ \leftarrow RZ[IMM5] clear
	bclri16 rz, imm5
	Clear the bit indicated by the IMM5 field value in the value of RZ, keep the other bits unchanged, and store the cleared result in RZ.
Operation Syntax Description	None Impact Flags
Bit	
limit	The register range is r0-r7; The range of immediate values is 0-31.
abnormal	none

16-bit instruction format:

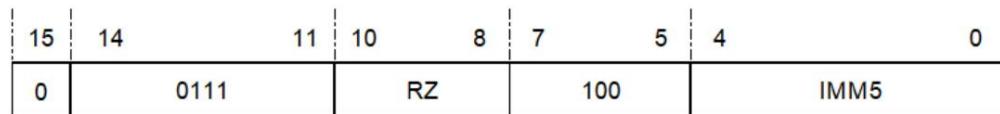


Figure 14.23: BCLRI-1

32-bit instructions	
	RZ \leftarrow RX[IMM5] cleared
	bclri32 rz, rx, imm5
	Clear the bit indicated by the IMM5 field value in the RX value, keep the other bits unchanged, and store the cleared result in RZ.
Operation Syntax Description	None Impact Flags
Bit	
Limit	The range of immediate values is 0-31.
exception	none

32-bit instruction format:

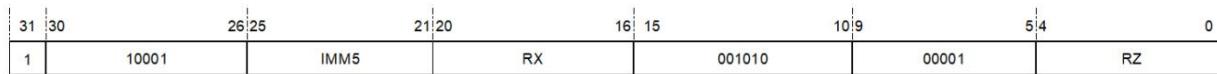


Figure 14.24: BCLRI-2

14.14 BEZ - Branch if Register Equals Zero

Unified	
instruction syntax	bez rx,
label operation	register equals zero then program transfer if ($RX == 0$) $PC \rightarrow PC + sign_extend(offset << 1)$ else $PC \rightarrow PC + 4$
The	Only 32-bit instructions exist bez32 rx, label
compilation result shows that	if register RX is equal to zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, $PC \rightarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BEZ instruction is $\pm 64KB$ address
Affects	space. No effect
flag	bit abnormality

32-bit	
instruction operation	register is equal to zero, then the program transfer if ($RX == 0$) $PC \rightarrow PC + sign_extend(offset << 1)$ else $PC \rightarrow PC + 4$
Syntax	bez32 rx, label
Description	If register RX is equal to zero, the program will jump to label; otherwise, the program will execute the next instruction, that is, $PC \rightarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BEZ instruction is $\pm 64KB$ address
Affects	space. No effect
flag	bit abnormality

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		01000		RX		Offset	

Figure 14.25: BEZ

14.15 BF——C is 0 branch instruction

Unified	
instruction syntax	bf
label operation	If C is equal to zero, the program will jump. if ($C==0$) PC \rightarrow PC + sign_extend(offset << 1); else PC \rightarrow next PC; Compile
Compilation result	to the corresponding 16-bit or 32-bit instruction according to the jump range. If offset < 1KB, then bf16 label; else bf32 label;
If	If the condition flag C is equal to zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction. Label is obtained by adding the current program PC to the relative offset shifted 1 bit to the left and extending it to 32 bits. The transfer range of the BF instruction is $\pm 64KB$ address space. No
Affects flag bit	effect
abnormality	

16-bit instruction	
operation	C is equal to zero, the program will be transferred. if ($C == 0$) PC \rightarrow PC + sign_extend(offset << 1) else PC \rightarrow PC + 2
Syntax	bf16 label
Description	If the conditional flag C is equal to 0, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, PC \rightarrow PC + 2. Label is obtained by adding the current program PC to the 10-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BF16 instruction is $\pm 1KB$ address space.
Affects	No effect
Flags	bit abnormality

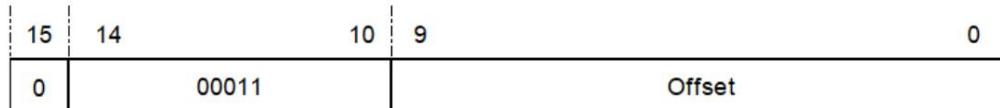
16-bit instruction format:

Figure 14.26: BF-1

32-bit instruction	
operation	C is equal to zero, then the program transfers if ($C == 0$) PC \rightarrow PC + sign_extend(offset << 1) else PC \rightarrow PC + 4
Syntax	b32 label
Description	If the conditional flag C is equal to zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, PC \rightarrow PC + 4. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BF instruction is $\pm 64KB$ address space.
Affects	No effect
Flags	bit abnormality

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		00010		00000			Offset

Figure 14.27: BF-2

14.16 BGENI——Generate Bit Immediately Instruction

Unified	
instruction syntax	bgeni rz, imm5
Operation	RZ \leftarrow (2) IMM5 compilation
only has 32-bit instructions. Result	
bgeni32 rz, imm5 Description	Set the bit of RZ (RZ[IMM5]) determined by the 5-bit immediate value and clear the other bits of RZ.
	Note that if IMM5 is less than 16, this instruction is a pseudo-instruction of movi rz, (2)IMM5 ; if IMM5 is greater than or equal to 16, this instruction is a pseudo-instruction of movih rz, (2)IMM5 . No effect
The flag bit	
affects	the immediate value range to 0-31.

32-bit	
instruction operation	RZ \leftarrow (2) IMM5;
Syntax	bgeni32 rz, imm5 Description
Set the bit of RZ (RZ[IMM5]) determined by the 5-bit immediate value and clear the other bits of RZ.	
	Note that if IMM5 is less than 16, this instruction is a pseudo-instruction of movi32 rz, (2)IMM5 ; if IMM5 is greater than or equal to 16, this instruction is a pseudo-instruction of movih32 rz, (2)IMM5 . No effect
The flag bit	
affects	the immediate value range to 0-31.

32-bit instruction format:

If IMM5 is less than 16:

31	30	26	25	21	20	16	15	0
1	11010		10000		RZ		(2) IMM5	

Figure 14.28: BGENI-1

If IMM5 is greater than or equal to 16:

31 30	26 25	21 20	16 15	0
1	11010	10000	RZ	(2) IMM5

Figure 14.29: BGENI-2

14.17 BGENR - Register Bit Generate Instruction

Unified	
instruction syntax	bgenr rz,
rx operation	If (RX[5] == 0) , then RZ \leftarrow 2RX[4:0]; else RZ \leftarrow 0;
Compilation result	only exists for 32-bit instructions bgenr32 rz,
rx Description	If RX[5] is 0, then set the register bit of RZ determined by the lower five bits of RX (RX[4:0]) and clear all other bits of RZ; otherwise, clear RZ. No
Affects flag	effect
bit abnormality	

32-bit	
instruction operation	If (RX[5] == 0), then RZ \leftarrow 2RX[4:0]; else RZ \leftarrow 0;
Syntax	bgenr32 rz, rx
Description	If RX[5] is 0, then set the RZ register bit determined by the RX lower five bits (RX[4:0]) and clear all other RZ bits; otherwise, clear RZ to 0. No effect
Affects flag	
bit abnormality	

32-bit instruction format:

31 30	25 25	21 20	16 15	10 9	5 4	0
1	10001	00000	RX	010100	00010	RZ

Figure 14.30: BGENR

14.18 BHSZ—Branch if register is greater than or equal to zero

Unified instruction	
syntax	bhsz rx, label operation
register	is greater than or equal to zero, then the program will be transferred if($RX \geq 0$) PC \leftarrow PC + sign_extend(offset << 1); else PC \leftarrow PC + 4; only
The	32-bit instructions exist bhsz32 rx, label
compilation result shows that	if the register RX is greater than or equal to zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, PC \leftarrow PC+4. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BHSZ instruction is $\pm 64KB$ address space.
	No effect
Affects	flag bit abnormality

32-bit instruction	
operation	register is greater than or equal to zero, then the program will be transferred if($RX \geq 0$) PC \leftarrow PC + sign_extend(offset << 1) else PC \leftarrow PC + 4
Syntax	bhsz32 rx, label Description
If register	RX is greater than or equal to zero, the program will jump to label for execution; otherwise, the program will execute the next instruction, that is, PC \leftarrow PC+4. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BHSZ instruction is $\pm 64KB$ address space.
	No effect
Affects	flag bit abnormality

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		01101		RX		Offset	

Figure 14.31: BHSZ

14.19 BHZ - Branch if Register Greater Than Zero

Unified	
instruction syntax	bhz rx,
label operation	register is greater than zero, then the program transfer if ($RX > 0$) $PC \leftarrow PC + sign_extend(offset \ll 1)$ else $PC \leftarrow PC + 4$
The	Only 32-bit instructions exist bhz32 rx, label
compilation result shows that	if the register RX is greater than zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, $PC \leftarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BHZ instruction is $\pm 64KB$ address
Affects	space. No effect
flag	
bit	
abnormality	

32-bit instructions	
If the operation register is greater than zero, the program will be transferred.	<pre>if(RX > 0) PC ← PC + sign_extend(offset << 1) else PC ← PC + 4</pre>
Syntax bhz32 rx, label	
If the register RX is greater than zero, the program will transfer to label; otherwise, the program will execute the next instruction, that is, PC ← PC + 4.	<p>Label is obtained by adding the current program PC to the 16-bit relative offset shifted left by 1 bit and signed extended to 32 bits.</p> <p>The instruction transfer range is ±64KB address space.</p>
Influence	No impact
Logo	
Bit	
Abnormal	

32-bit instruction format:

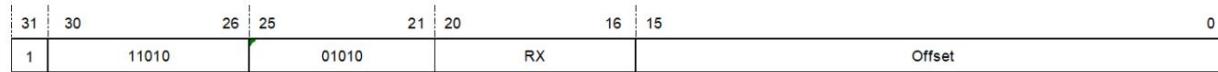


Figure 14.32: BHZ

14.20 BKPT—Breakpoint Instruction

Unified instructions	
	Causes a breakpoint exception or enters debug mode
Operations	are always compiled as 16-bit instructions.
	bkpt16
Description	Breakpoint instruction
Affects flag bit	No effect
Exception	breakpoint exception

16-bit instructions	
The operation	causes a breakpoint exception or enters debug mode
Grammar	bkpt16
	Breakpoint instructions
Description:	The impact flag has no impact
Exception	breakpoint exception

16-bit instruction format:



Figure 14.33: BKPT

14.21 BLSZ——Branch if register is less than or equal to zero

Unified	
instruction syntax	blsz rx,
label operation	register is less than or equal to zero if the program transfer if($RX \leq 0$) $PC \rightarrow PC + \text{sign_extend}(\text{offset} \ll 1)$ else $PC \rightarrow PC + 4$
The	Only 32-bit instructions exist: blsz32 rx, label
compilation result shows that	if the register RX is less than or equal to zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, $PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BLSZ instruction is $\pm 64KB$ address space.
Affects	No effect
flag	bit abnormality

32-bit instruction	
operation	register is less than or equal to zero, then the program will be transferred if($RX \leq 0$) PC \leftarrow PC + sign_extend(offset $\ll 1$) else PC \leftarrow PC + 4
Syntax	bisz32 rx, label
Description	If register RX is less than or equal to zero, the program will jump to label for execution; otherwise, the program will execute the next instruction, i.e. PC PC+4. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BLSZ instruction is $\pm 64KB$ address space.
Affects	No effect
flag	bit abnormality

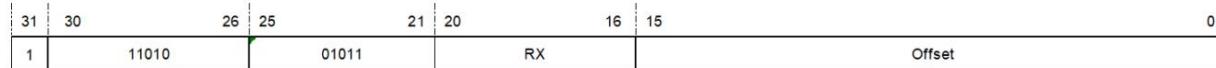
32-bit instruction format:

Figure 14.34: BLSZ

14.22 BLZ——Branch if Register is Less Than Zero

Unified	
instruction syntax	blz rx,
label operation	register is less than zero, then the program will jump if ($RX < 0$) $PC \rightarrow PC + sign_extend(offset \ll 1)$ else $PC \rightarrow PC + 4$
The	Only 32-bit instructions exist blz32 rx, label
compilation result shows that	if the register RX is less than zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, $PC \rightarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BLZ instruction is $\pm 64KB$ address
Affects	space. No effect
flag	bit abnormality

32-bit instruction	
operation	register is less than zero, then the program will jump if ($RX < 0$) $PC \rightarrow PC + sign_extend(offset \ll 1)$ else $PC \rightarrow PC + 4$
Syntax	blz32 rx, label
Description	If register RX is less than zero, the program will jump to label; otherwise, the program will execute the next instruction, that is, $PC \rightarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BLZ instruction is $\pm 64KB$ address
Affects	space. No effect
flag	bit abnormality

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		01100		RX		Offset	

Figure 14.35: BLZ

14.23 BMASKI—Immediate Digital Mask Generate Instruction

Unified instruction	
	bmaski rz, oimm5
	RZ ý (2)OIMM5 - 1
	Only 32-bit instructions exist: bmaski32 rz, oimm5
syntax operation compilation result bright	the value with consecutive low bits as 1 and high bits as 0, and store the immediate value in RZ. The immediate value OIMM5 with offset specifies the number of consecutive low bits (RX[OIMM5-1:0]) to be set to 1, and the remaining high bits are cleared to 0. When OIMM5 is 0 or 32, all RX bits are set to 1. Note that when OIMM5 is 1-16, it is executed by the movi instruction. No effect
	The range of immediate values is 0, 17-32;
	none
Affects flag bit restriction exception	

32-bit instructions	
	RZ ý (2) OIMM5-1 Operation
	bmask32 rz, oimm5
Syntax Description	Generates an immediate value with consecutive low bits as 1 and high bits as 0, and stores the immediate value in RZ. The immediate value OIMM5 specifies the number of consecutive low bits (RX[OIMM5:1:0]) to be set to 1, and the remaining high bits are cleared to 0. When OIMM5 is 0 or 32, all RX bits are set to 1. Note that when OIMM5 is 1-16, it is executed by the movi instruction; the binary operand IMM5 is equal to OIMM5
	- 1. No effect
	The range of immediate values is 0, 17-32;
Affects flag bit restriction exception	none

32-bit instruction format:

Figure 14.36: BMASKI

IMM5 domain:

Specifies the highest bit of consecutive low bits to be set.

Note: The immediate value OIMM5 must be offset by 1 compared to the binary operand IMM5.

10000:

0-16 Position

10001:

0-17 Position

...

11111:

0-31 Position

14.24 BNEZ - Branch if register is not equal to zero

Unified	
instruction syntax	bnez rx,
label operation	register is not equal to zero, then the program will jump if ($RX \neq 0$) $PC \leftarrow PC + sign_extend(offset \ll 1)$ else $PC \leftarrow PC + 4$
The	Only 32-bit instructions exist. bnez32 rx, label
compilation result shows that	if the register RX is not equal to zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, $PC \leftarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BNEZ instruction is $\pm 64KB$ address space.
Affects	No effect
flag	bit abnormality

32-bit instruction	
operation	register is not equal to zero, then the program transfer if ($RX \neq 0$) $PC \leftarrow PC + sign_extend(offset \ll 1)$ else $PC \leftarrow PC + 4$
Syntax	bnez32 rx, label
Description	If register RX is not equal to zero, the program will jump to label for execution; otherwise, the program will execute the next instruction, that is, $PC \leftarrow PC + 4$. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BNEZ instruction is $\pm 64KB$ address space.
Affects	No effect
flag	bit abnormality

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		01001		RX		Offset	

Figure 14.37: BNEZ

14.25 BNEZAD - Branch if register is decremented greater than zero

The unified	
instruction syntax is bnezad rx, label. The	
register is decremented by 1 and then judged. If it is greater than zero, the program will be transferred. The decrement result will unconditionally update the register.	<pre> RX = RX-1 if(RX > 0) PC = PC + sign_extend(offset << 1) else PC = PC + 4 </pre>
Compilation results	Only 32-bit instructions exist. bnez32 rx, label

illustrate:	Register RX is decremented by 1 and then judged. If the result is greater than zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, PC = PC + 4. The decrement result will unconditionally update register RX. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BNEZAD instruction is ±64KB address
	space. No effect
Affected flag: Abnormal:	none

32	Bit
instruction	If the register is not equal to zero, the program will be transferred RX > RX-1 if($RX > 0$) $PC \leftarrow PC + \text{sign_extend}(\text{offset} \ll 1)$ else $PC \leftarrow PC + 4$
language Law:	bnezad32 rx, label
illustrate	Register RX is decremented by 1 and then judged. If the result is greater than zero, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, $PC \leftarrow PC + 4$. The decrement result will unconditionally update register RX. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BNEZAD instruction is $\pm 64KB$ address
	space. No effect
Affected flag: Abnormal:	none

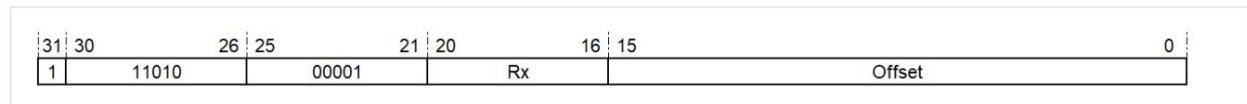
Instruction format:

Figure 14.38: BNEZAD

14.26 BR——Unconditional Jump Instruction

Unified	
instruction syntax	br label
operation	PC \leftarrow PC + sign_extend(offset << 1) Compilation result Compiled
	into corresponding 16-bit or 32-bit instructions according to the jump range if(offset<1KB), then br16 label; else br32 label;
indicates that	the program jumps unconditionally to label for execution. Label is obtained by adding the current program PC to the relative offset shifted left by 1 bit and extending it to 32 bits. No
Affects flag bit	effect
abnormality	

16-bit	
instruction	operation PC \leftarrow PC + sign_extend(offset << 1) Syntax br16
label Description	The
	program jumps unconditionally to label for execution. Label is obtained by adding the current program PC to the 10-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The jump range of the BR16 instruction is $\pm 1KB$ address space.
Affects flag bit	No effect
abnormality	

16-bit instruction format:



Figure 14.39: BR-1

32-bit	
instruction	operation PC \leftarrow PC + sign_extend(offset << 1) Syntax br32
label	Description The program jumps unconditionally to label for execution.
	Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The jump range of the BR instruction is $\pm 64KB$ address space. No
Affects	effect
flag bit	
abnormality	

32-bit instruction format:

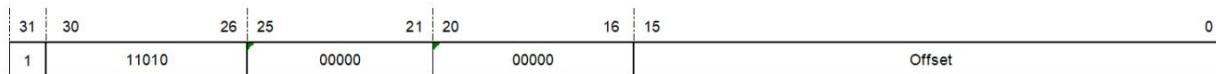


Figure 14.40: BR-2

14.27 BREV——Bit Reverse Instruction

Unified	
instruction	syntax: brev rz,
rx operation	for i=0 to 31 RZ[i] \leftarrow RX[31-i]; Only 32-bit instructions exist in the compilation .
The result is	brev32 rz, rx, which means to reverse the value of RX bit by bit and store the result in RZ.
	If the value of RX is "abc defghijklmnopqrstuvwxyz012345", after reversing the bit order, the value of RZ becomes "543210zyxwvutsrqponmlkjihgfedcba". No effect
Affects	
flag bit	
abnormality	

32-bit	
instruction	operation for $i=0$ to 31 RZ[i] \leftarrow RX[31-i];
Syntax	brev32 rz, rx
Description	Reverse the value of RX bit by bit and store the result in RZ. If the value of RX is "abc defghijklmnopqrstuvwxyz012345", after reversing the bit order, the value of RZ becomes "543210zyxwvutsrqponmlkjihgfedcba". No effect
Affects flag bit	
abnormality	

Instruction format:

Figure 14.41: BREV

14.28 BSETI – Set Bit Immediately Instruction

Unified instruction		
	bseti rz, imm5	bseti rz, rx, imm5
	RZ ѕ RZ[IMM5] is set	RZ ѕ RX[IMM5] is set
syntax operation	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if bseti16 rz, imm5; else bseti32 rz, rz, imm5; Set the	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if((x==z) and (z<8), then bseti16 rz, imm5; else bseti32 rz, rx, imm5;
	position indicated by the IMM5 field value in the value of RZ/RX to 1, and keep the other bits unchanged, and store the set result in RZ.	
	No impact	
	The range of immediate values is 0-31.	
	none	
Description: Affects the flag bit limit exception		

16-bit instruction	
	RZ ѕ RZ[IMM5] Set bseti16 rz,
	imm5 Set the position
	indicated by the IMM5 field value in the value of RZ to 1, and keep the other bits unchanged, and store the result after setting in RZ. No
operation syntax	effect
	The range of registers is r0-r7; the range of immediate values is 0-31.
	description affects the flag bit limit exception

16-bit instruction format:

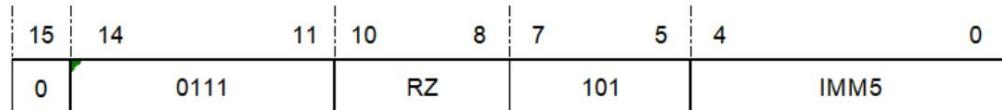


Figure 14.42: BSETI-1

32-bit instructions	
	RZ \leftarrow RX[IMM5] is set
	bseti32 rz, rx, imm5
	Set the position indicated by the IMM5 field value in the RX value to 1, and keep the other bits unchanged. Store the set result in RZ.
Operation Syntax Description Impact Flags	
Bit	
Limit	The range of immediate values is 0-31.
exception	none

32-bit instruction format:



Figure 14.43: BSETI-2

14.29 BSR - Jump to Subroutine Instruction

Unification refers to make	
Syntax	bsr label
operations	to link and jump to subroutines: R15 \leftarrow next PC PC \leftarrow PC + sign_extend(offset << 1)
The compilation result	is compiled into the corresponding 16-bit or 32-bit instructions according to the jump range. if(0<offset<1KB), then bsr16 label; else bsr32 label;

Description:	Subroutine jump, save the subroutine return address (the PC of the next instruction) in the link register R15, and the program is transferred to label. Label is obtained by adding the current program PC to the relative offset shifted left by 1 bit and extending it to 32 bits. No
Affected flag:	effect
Abnormal:	None

16 Bit	
instruction operation:	Link and jump to a subroutine: $R15 \leftarrow PC + 2$ $PC \leftarrow PC + \text{sign_extend}(\text{offset} \ll 1)$ bsr16 label
Syntax : Description:	The subroutine jumps and saves the return address of the subroutine (the PC of the next instruction, that is, the current PC + 2) in the link register R15, and the program is transferred to the label for execution. Label is obtained by adding the current program PC to the 10-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The jump range of the BSR16 instruction is $\pm 1KB$ address
Affected flags:	space. No effect
	The jump target of a BSR16 instruction cannot be the BSR16 instruction itself.
	none

Instruction format:

Figure 14.44: BSR-1

Offset field - specifies the relative offset of the jump.

Note: The relative offset (Offset) of the jump cannot be 0x0.

32	
Bit	
	<p>Link and jump to a subroutine:</p> <p>instruction operation: R15, PC+4</p> <p>PC = PC + sign_extend(offset << 1)</p>
	bsr32 label
	<p>The subroutine jumps and saves the return address of the subroutine (the PC of the next instruction, that is, the current PC+4) in the link register R15, and the program is transferred to the label for execution.</p> <p>Label is obtained by adding the current program PC to the 26-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The jump range of the BSR instruction is ±64MB address space. No effect</p>
	Affected flag: Abnormal: none

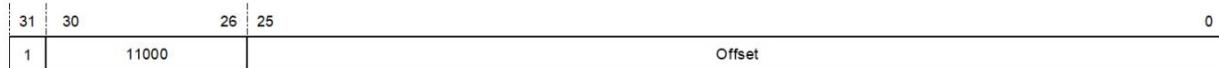
Instruction format:

Figure 14.45: BSR-2

14.30 BT——C is 1 branch instruction

Unified	
instruction syntax	bt
Description	<p>label operation if($C == 1$)</p> <p style="padding-left: 2em;">PC \leftarrow PC + sign_extend(offset << 1);</p> <p style="padding-left: 2em;">else</p> <p style="padding-left: 2em;">PC \leftarrow next PC; Compile</p>
Compilation result	<p>to the corresponding 16-bit or 32-bit instruction according to the jump</p> <p>if (offset<1KB), then bt16</p> <p style="padding-left: 2em;">label;</p> <p style="padding-left: 2em;">else</p> <p style="padding-left: 2em;">bt32 label;</p>
If	<p>If the condition flag C is equal to 1, the program will transfer to label for execution; otherwise, the program will execute the next instruction.</p> <p>Label is obtained by adding the current program PC to the relative offset shifted left by 1 bit and extending it to 32 bits. The transfer range of the BT instruction is $\pm 64KB$ address</p>
Affects	space. No effect
flag bit	
abnormality	

16-bit	
instruction operation	C is equal to a
	program transfer if ($C == 1$)
	PC \leftarrow PC + sign_extend(offset << 1)
	else
	PC \leftarrow PC + 2
Syntax	bt16 label
Description	<p>If the conditional flag C is equal to 1, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, PC \leftarrow PC + 2.</p> <p>Label is obtained by adding the current program PC to the 10-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of BT16 instruction is $\pm 1KB$ address</p>
Affects	space. No effect
flag	
bit	
abnormality	

16-bit instruction format:

15	14	10	9	0
0	00010		Offset	

Figure 14.46: BT-1

32-bit	
instruction operation	C is equal to a program transfer if(C == 1) PC → PC + sign_extend(offset << 1) else PC → PC + 4
Syntax	bt32 label
Description	If the conditional flag C is equal to 1, the program will transfer to label for execution; otherwise, the program will execute the next instruction, that is, PC → PC + 4. Label is obtained by adding the current program PC to the 16-bit relative offset shifted 1 bit to the left and signed extended to 32 bits. The transfer range of the BT instruction is ±64KB address space.
Affects flag	No effect
bit abnormality	

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010	00011		00000		Offset		

Figure 14.47: BT-2

14.31 BTSTI——Bit Test Immediate Instruction

The unified	
	btsti rx, imm5
	C → RX[IMM5]
instruction syntax	operation compilation result only has 32-bit instructions btsti32 rx, imm5 Test
	the RX bit (RX[IMM5]) determined by IMM5 and make the value of the condition bit C equal to the value of this bit. Description
Affects the flag bit	C → RX[IMM5] Limits
	the range of the immediate value to 0-31. Exception
	None

32-bit instructions	
Operation	C \neq RX[IMM5]
Syntax	btsti32 rx, imm5
	The RX bit determined by IMM5 (RX[IMM5]) is tested and the value of condition bit C is equal to the value of this bit.
Description	Affects flag bit C \neq RX[IMM5]
Limits	the immediate value to the range 0-31.
Abnormal	none

32-bit instruction format:

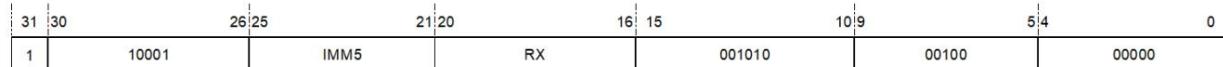


Figure 14.48: BTSTI

14.32 CLRF——C is 0 clear instruction

Unified instructions	
Syntax	clrf
Operations	if C==0, then RZ \leftarrow 0; else RZ \leftarrow RZ;
The compiled result	contains only 32-bit instructions. clrf32 rz
	If C is 0, register RZ is cleared; otherwise, register RZ remains unchanged.
Description:	The impact flag has no impact
Abnormal	

32-bit instructions	
operate	if C==0, then RZ \leftarrow 0; else RZ \leftarrow RZ;
	clrf32 rz
	If C is 0, register RZ is cleared; otherwise, register RZ remains unchanged.
Syntax Description Affects flags	No effect
Abnormal	

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RZ	00000	001011	00001	00000

Figure 14.49: CLRF

14.33 CLRT——C is 1 clear instruction

Unified instructions	
Syntax	clrt
Operations	if C==1, then RZ \leftarrow 0; else RZ \leftarrow RZ;
The compiled result	contains only 32-bit instructions. clrt32 rz
	If C is 1, register RZ is cleared; otherwise, register RZ remains unchanged.
Description: The impact flag	has no impact
Abnormal	

32-bit instructions	
operate	if C==1, then RZ \leftarrow 0; else RZ \leftarrow RZ;
	clrt32 rz
	If C is 1, register RZ is cleared; otherwise, register RZ remains unchanged.
Syntax Description Affects flags	No effect
abnormal	none

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RZ	00000	001011	00010	00000

Figure 14.50: CLRT

14.34 CMPHS - Unsigned Greater Than or Equal Comparison Instruction

The unified	
instruction syntax	cmphs rx, ry
operation	<p>performs an unsigned comparison between RX and RY.</p> <p>If RX >= RY, then C = 1;</p> <p>else</p> <p>C = 0;</p>
Compilation result	<p>Compile to the corresponding 16-bit or 32-bit instruction according to the range</p> <p>if (x<16) and (y<16), then</p> <p>cmphs16 rx, ry;</p> <p>else</p> <p>cmphs32 rx, ry;</p>
Description	<p>Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. cmphs performs unsigned comparison, that is, the operands are considered to be unsigned numbers. If RX is greater than or equal to RY, that is, the subtraction result is greater than or equal to 0, then set the conditional bit C; otherwise, clear the conditional</p>
Affects	bit C. Set the conditional bit C according to the comparison result
flag bit	
abnormality	

The 16-bit	
instruction	<p>performs an unsigned comparison between RX and RY.</p> <p>If RX >= RY, then C = 1;</p> <p>else</p> <p>C = 0;</p>
Syntax	cmphs16 rx, ry
Description	<p>Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. cmphs16 performs unsigned comparison, that is, the operands are considered to be unsigned numbers. If RX is greater than or equal to RY, that is, the subtraction result is greater than or equal to 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set</p>
The range of	the conditional bit C according to the comparison result
the affected flag limit	register is r0- r15 .

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11001		RY		RX		00	

Figure 14.51: CMPHS-1

The 32-bit	
instruction	performs an unsigned comparison between RX and RY. If RX >= RY, then C = 1; else C = 0;
Syntax cmphs32 rx, ry	Description Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. cmphs32 performs unsigned comparison, that is, the operands are considered to be unsigned numbers. If RX is greater than or equal to RY, that is, the subtraction result is greater than or equal to 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set
Affects flag bit	the conditional bit C according to the comparison result
abnormality	

32-bit instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10001		RY		RX		000001		00001		00000	

Figure 14.52: CMPHS-2

14.35 CMPHSI - Immediate Unsigned Greater Than or Equal Comparison Instruction

Unified instruction	
language	cmhsr rx, oimm16 method
operation	<p>RX performs an unsigned comparison with the immediate value.</p> <p>If RX >= zero_extend(OIMM16), C = 1;</p> <p>else</p> <p>C = 0;</p>
Compilation results	<p>Compile to the corresponding 16-bit or 32-bit instruction according to the range of the immediate value and register. if (oimm16<33) and (x<8),then cmphsi16</p> <p>rx, oimm5;</p> <p>cmphsi32 rx, oimm16; Zero-extend</p>
illustrate	<p>the 16-bit immediate value (OIMM16) with offset 1 to 32 bits, then subtract the 32-bit value from the value of RX, compare the result with 0, and update the C bit. cmphsi performs an unsigned comparison, that is, the operands are considered unsigned. If RX is greater than or equal to the zero-extended OIMM16, that is, the subtraction result is greater than or equal to 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set the conditional bit C according to the comparison result</p>
Impact Mark	
Zhi	
Bit	
	The range of immediate value is 0x1-0x10000.
Limit Exception	none

16-bit	
	instruction RX compares the immediate value with an unsigned value. Operation If RX >= zero_extend(OIMM5), then C = 1; else C = 0;
	cmphsi16 rx, oimm5
Syntax	Zero-extend the 5-bit immediate value (OIMM5) with bias 1 to 32 bits, then subtract the 32-bit value from the value of RX, compare the result with 0, and update the C bit. cmphsi16 performs an unsigned comparison, that is, the operands are considered unsigned. If RX is greater than or equal to the zero-extended OIMM5, that is, the subtraction result is greater than or equal to 0, then set the condition bit C; otherwise, clear the condition bit C. Note: The binary operand IMM5 is equal to OIMM5 - 1. Set the
	condition bit C according to the comparison result
Impact Mark	
Zhi Bit	
	The range of registers is r0-r7; the range of immediate values is 1-32.
Limit Exception	none

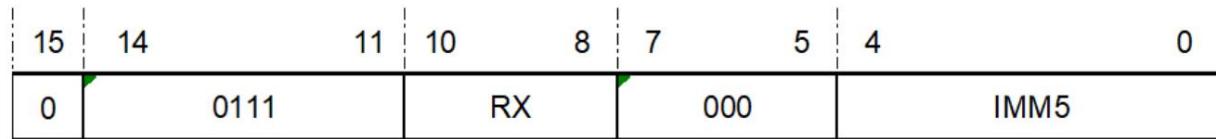
16-bit instruction format:

Figure 14.53: CMPHSI-1

IMM5 domain:

Specifies the value of an immediate value without an offset.

Note: The immediate value OIMM5 involved in the comparison needs to be offset by 1 compared to the binary operand IMM5.

00000:

Compare with 1

00001:

Compare with 2

...

11111:

Compare with 32

32-bit	
instruction operation RX and immediate data	for unsigned comparison. If RX >= zero_extend(OIMM16), then C = 1; else C = 0;
	cmphsi32 rx, oimm16
Syntax	Zero-extend the 16-bit immediate value (OIMM16) with bias 1 to 32 bits, then subtract the 32-bit value from the value of RX, compare the result with 0, and update the C bit. cmphsi32 performs an unsigned comparison, that is, the operands are considered unsigned. If RX is greater than or equal to the zero-extended OIMM16, that is, the subtraction result is greater than or equal to 0, then set the condition bit C; otherwise, clear the condition bit C. Note: The binary operand IMM16 is equal to OIMM16 - 1. Set the
Impact flag	condition bit C according to the comparison result
limit	The range of immediate value is 0x1-0x10000.
	none
Control Abnormality	

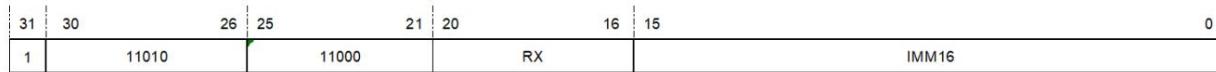
32-bit instruction format:

Figure 14.54: CMPHSI-2

IMM16 domain:

Specifies the value of an immediate value without an offset.

Note: The immediate value OIMM16 involved in the comparison needs to be offset by 1 compared to the binary operand IMM16.

0000000000000000:

Compare with 0x1

0000000000000001:

Compare with 0x2

...

1111111111111111:

Compare with 0x10000

14.36 CMPLT - Signed Less Than Compare Instruction

The unified	
instruction syntax	cmplt rx, ry
operation	<p>performs a signed comparison between RX and RY.</p> <p>If RX < RY, then C = 1;</p> <p>else</p> <p>C = 0;</p>
Compilation result	<p>Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 16$) and ($y < 16$),</p> <p>then cmplt16 rx, ry;</p> <p>else</p> <p>cmplt32 rx, ry;</p>
Description	<p>Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. cmplt performs a signed comparison, that is, the operands are considered to be signed numbers in the form of two's complement. If RX is less than RY, that is, the subtraction result is less than 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set the</p>
Affects flag bit	conditional bit C according to the comparison result
abnormality	

The 16-bit	
instruction	<p>performs a signed comparison between RX and RY.</p> <p>If RX < RY, then C = 1;</p> <p>else</p> <p>C = 0;</p>
Syntax	cmplt16 rx, ry
Description	<p>Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. cmplt16 performs a signed comparison, that is, the operands are considered to be signed numbers in the form of two's complement. If RX is less than RY, that is, the subtraction result is less than 0, then set the conditional bit C; otherwise, clear the conditional bit</p>
The range of	C. Set the conditional bit C according to the comparison result
the affected flag limit	register is r0- r15 .

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11001		RY		RX		01	

Figure 14.55: CMPLT-1

The 32-bit	
instruction	performs a signed comparison on RX and RY. If RX < RY, then C = 1; else C = 0;
Syntax cmplt32 rx, ry	Description Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. cmplt32 performs a signed comparison, that is, the operands are considered to be signed numbers in the form of two's complement. If RX is less than RY, that is, the subtraction result is less than 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set the
Affects flag bit	conditional bit C according to the comparison result
abnormality	

32-bit instruction format:

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RY	RX	000001	00010	00000	

Figure 14.56: CMPLT-2

14.37 CMPLTI - Immediate Signed Less Than Compare Instruction

Unified instruction	
syntax	<code>cmplti rx, oimm16 method</code>
operation	<p>RX performs a signed comparison with the immediate value.</p> <p>If RX < zero_extend(OIMM16), C = 1;</p> <p>else</p> <p>C = 0;</p>
Compilation results	<p>Compile to the corresponding 16-bit or 32-bit instruction according to the range of the immediate value and register. if (x<8) and (oimm16<33), then</p> <p><code>cmplti16 rx, oimm5;</code></p> <p><code>cmplti32 rx, oimm16; Zero-extends</code></p>
illustrate	<p>the 16-bit immediate value (OIMM16) with offset 1 to 32 bits, then subtracts the 32-bit value from the value of RX, compares the result with 0, and updates the C bit. cmplti performs a signed comparison, that is, the value of RX is considered to be a signed number in two's complement form. If RX is less than the zero-extended OIMM16, that is, the subtraction result is less than 0, then set the condition bit C; otherwise, clear the condition bit C.</p>
Impact flag	Set condition bit C according to the comparison result
limit	The range of immediate value is 0x1-0x10000.
Control Abnormality	none

16-bit	
	instruction RX compares the immediate value with a signed value. Operation If RX < zero_extend(OIMM5), then C = 1; else C = 0;
	cmplti16 rx, oimm5
Syntax	Zero-extend the 5-bit immediate value (OIMM5) with bias 1 to 32 bits, then subtract the 32-bit value from the value of RX, compare the result with 0, and update the C bit. cm plti16 performs a signed comparison, that is, the value of RX is considered to be a signed number in two's complement form. If RX is less than the zero-extended OIMM5, that is, the subtraction result is less than 0, then set the condition bit C; otherwise, clear the condition bit C. Note: Binary operand IMM5 is equal to OIMM5 - 1. Condition bit C is
Impact	set based on the comparison result.
flag	
limit	The range of registers is r0-r7; the range of immediate values is 1-32.
Control Abnormality	none

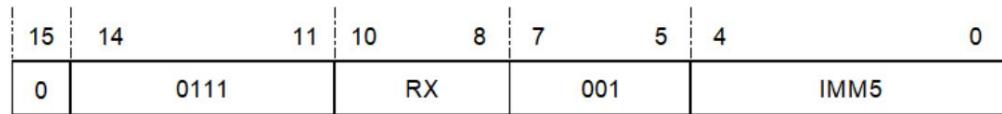
16-bit instruction format:

Figure 14.57: CMPLT-1

IMM5 domain:

Specifies the value of an immediate value without an offset.

Note: The immediate value OIMM5 involved in the comparison needs to be offset by 1 compared to the binary operand IMM5.

00000:

Compare with 1

00001:

Compare with 2

...

11111:

Compare with 32

32-bit instruction	
operation	RX performs a signed comparison with the immediate value. If RX < zero_extend(OIMM16), then C = 1; else C = 0;
	cmplti32 rx, oimm16
Syntax	Zero-extend the 16-bit immediate value (OIMM16) with bias 1 to 32 bits, then subtract the 32-bit value from the value of RX, compare the result with 0, and update the C bit. cm plti32 performs a signed comparison, that is, the value of RX is considered to be a signed number in two's complement form. If RX is less than the zero-extended OIMM16, that is, the subtraction result is less than 0, then set the condition bit C; otherwise, clear the condition bit C. Note: The binary operand IMM16 is equal to OIMM16 - 1. Set the condition bit C according to the
Impact flag	comparison result
limit	The range of immediate value is 0x1-0x10000.
	none
Control Abnormality	

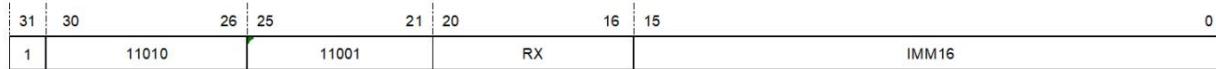
32-bit instruction format:

Figure 14.58: CMPLT-2

IMM16 domain:

Specifies the value of an immediate value without an offset.

Note: The immediate value OIMM16 involved in the comparison needs to be offset by 1 compared to the binary operand IMM16.

0000000000000000:

Compare with 0x1

0000000000000001:

Compare with 0x2

...

1111111111111111:

Compare with 0x10000

14.38 CMPNE—Not Equal Comparison Instruction

The unified	
instruction	syntax cmpne rx, ry
operation	compares RX with RY. If RX != RY, then C = 1; else C = 0;
Compilation results	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (y<16), then cmpne16 rx, ry; else cmpne32 rx, ry;
Description:	Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. If RX is not equal to RY, that is, the subtraction result is not equal to 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set
Affects flag	the conditional bit C according to the comparison result
bit abnormality	

The 16-bit	
instruction	operates by comparing RX with RY. If RX != RY, then C = 1; else C = 0;
Syntax	cmpne16 rx, ry
Description	Subtract the value of RY from the value of RX, compare the result with 0, and update the C bit. If RX is not equal to RY, the value is subtracted. If the comparison result is not equal to 0, set condition bit C; otherwise, clear condition bit C. Set
The range	condition bit C according to the comparison result
of the affected flag limit	register is r0- r15 .

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11001		RY		RX		1 0	

Figure 14.59: CMPNE-1

The 32-bit	
instruction	operates by comparing RX with RY. If RX != RY, then C = 1; else C = 0;
Syntax	<code>cmpne32 rx, ry</code> Description
Subtract	the value of RY from the value of RX, compare the result with 0, and update the C bit. If RX is not equal to RY, that is, the subtraction result is not equal to 0, set the conditional bit C; otherwise, clear the conditional bit C. Set the
Affects flag	conditional bit C according to the comparison result
bit abnormality	

32-bit instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10001		RY		RX		000001		00100		00000	

Figure 14.60: CMPNE-2

14.39 CMPNEI——Immediate Not Equal Comparison Instruction

Unified	
instruction syntax	cmpnei rx, imm16
operation	<p>RX compares with immediate value.</p> <p>If RX != zero_extend(imm16), C = 1;</p> <p>else</p> <p>C = 0;</p>
Compilation results	<p>Compile to the corresponding 16-bit or 32-bit instruction according to the range of the immediate value and register. if (x<7) and (imm16<33), then cmpnei16 rx, imm5;</p> <p>else</p> <p>cmpnei32 rx, imm16;</p>
Description:	<p>Subtract the value of the 16-bit immediate value that is zero-extended to 32 bits from the value of RX, compare the result with 0, and update the C bit.</p> <p>If RX is not equal to the zero-extended IMM16, that is, the subtraction result is not equal to 0, then set the condition bit C; otherwise, clear the condition bit C.</p>
The flag bit	Set condition bit C according to the comparison result
affects	the immediate value range to 0x0-0xFFFF.

The 16-bit	
instruction operation	<p>RX compares with the immediate value.</p> <p>If RX != zero_extend(IMM5), then C = 1;</p> <p>else</p> <p>C = 0;</p>
Syntax	cmpnei16 rx, imm5
Description	<p>Subtract the value of the 5-bit immediate value that is zero-extended to 32 bits from the value of RX, compare the result with 0, and update the C bit. If RX is not equal to the zero-extended IMM5, that is, the subtraction result is not equal to 0, then set the conditional bit C; otherwise, clear the conditional bit C. Set the conditional bit C according to the comparison result</p>
The range of the affected flag limit	register is r0-r7; the range of the immediate value is 0-31 .

16-bit instruction format:

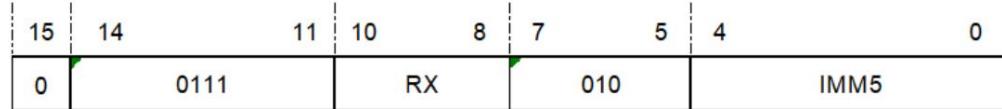


Figure 14.61: CMPNEI-1

The 32-bit	
instruction	operation RX compares with the immediate value. If RX != zero_extend(imm16), then C = 1; else C = 0;
Syntax	cmpnei rx, imm16 Description Subtract the value of the 16-bit immediate value that is zero-extended to 32 bits from the value of RX, compare the result with 0, and update the C bit. If RX is not equal to the zero-extended IMM16, that is, the subtraction result is not equal to 0, then set the condition bit C; otherwise, clear the condition bit C.
The flag bit	Set condition bit C according to the comparison result
affects	the immediate value range to 0x0-0xFFFF.

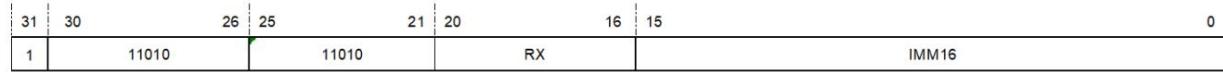
32-bit instruction format:

Figure 14.62: CMPNEI-2

14.40 DECF——C is 0 immediate subtraction instruction

Unified	
instruction syntax	decf rz, rx, imm5
operation if C==0, then	<p>RZ \leftarrow RX - zero_extend(Imm5);</p> <p>else</p> <p>RZ \leftarrow RZ; The</p>
compilation result	only has 32-bit
instruction result	decf32 rz, rx, imm5. If the
condition bit C is 0, extend the 5-bit immediate value to 32 bits, subtract the 32-bit value from the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain	
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit	
instruction operation	if C==0, then
	<p>RZ \leftarrow RX - zero_extend(Imm5);</p> <p>else</p> <p>RZ \leftarrow RZ;</p>
Syntax	decf32 rz, rx, imm5 Description If
the condition bit C is 0, zero-extend the 5-bit immediate value to 32 bits, subtract the 32-bit value from the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain	
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit instruction format:

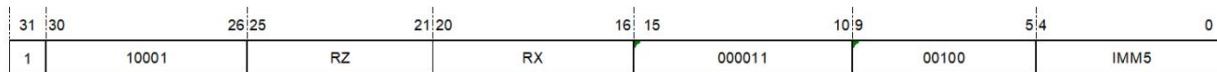


Figure 14.63: DECF

14.41 DECGT——Subtraction greater than zero set C instruction

Unified	
instruction syntax	decgt rz, rx, imm5 operation RZ - RX - zero_extend(IMM5); If RZ > 0, then C = 1; else C = 0;
Compilation result	Only 32-bit instructions exist: decgt32 rz, rx, imm5
shows	that the 5-bit immediate value is zero-extended to 32 bits, and the result of subtracting the 32-bit value from RX is stored in RZ. The subtraction result is considered to be a two's complement. If the result is greater than 0, set condition bit C; otherwise, clear condition bit C. If the
The flag bit	subtraction result is greater than 0, set condition bit C; otherwise, clear condition bit C.
affects	the immediate value range to 0-31.

32-bit	
instruction operation	RZ - RX - zero_extend(IMM5); If RZ > 0, then C = 1; else C = 0;
Syntax	decgt32 rz, rx, imm5 Description Zero-
extend	the 5-bit immediate value to 32 bits, and store the result of subtracting the 32-bit value from RX in RZ. The subtraction result is considered to be the two's complement. If the result is greater than 0, set condition bit C; otherwise, clear condition bit C. If the
The flag bit	subtraction result is greater than 0, set condition bit C; otherwise, clear condition bit C.
affects	the immediate value range to 0-31.

32-bit instruction format:

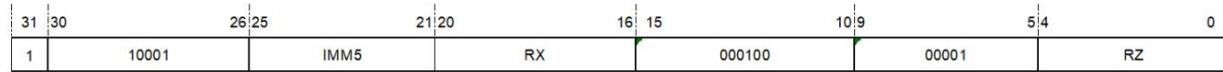


Figure 14.64: DECGT

14.42 DECLT - Set C if subtraction is less than zero

Unified	
instruction syntax	dect rz, rx, imm5 operation RZ - RX - zero_extend(IMM5); If RZ < 0, then C = 1; else C = 0;
Compilation result	only 32-bit instructions exist: declt32 rz, rx, imm5
shows	that the 5-bit immediate value is zero-extended to 32 bits, and the result of subtracting the 32-bit value from RX is stored in RZ. The subtraction result is considered to be a two's complement. If the result is less than 0, set condition bit C; otherwise, clear condition bit C. If the
The flag bit	subtraction result is less than 0, set condition bit C; otherwise, clear condition bit C.
affects	the immediate value range to 0-31.

32-bit	
instruction operation	RZ - RX - zero_extend(IMM5); If RZ < 0, then C = 1; else C = 0;
Syntax	declt32 rz, rx, imm5 Description Zero-
extend	the 5-bit immediate value to 32 bits, and store the result of subtracting the 32-bit value from RX in RZ. The subtraction result is considered to be the two's complement. If the result is less than 0, set condition bit C; otherwise, clear condition bit C. If the
The flag bit	subtraction result is less than 0, set condition bit C; otherwise, clear condition bit C.
affects	the immediate value range to 0-31.

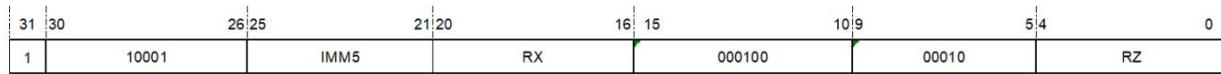
32-bit instruction format:

Figure 14.65: DECLT

14.43 DECNE - Set C if subtraction is not equal to zero

Unified	
instruction syntax	decne rz, rx, imm5
operation	RZ \leftarrow RX - zero_extend(imm5); If RZ \neq 0, then C \leftarrow 1; else C \leftarrow 0;
Compilation	only 32-bit instructions exist: decne32 rz, rx, imm5
result Description	Extend the 5-bit immediate value to 32 bits, and store the result of subtracting the 32-bit value from RX in RZ. If the result is not equal to 0, set Set condition bit C; otherwise, clear condition bit
The flag bit	C. If the subtraction result is not equal to 0, set condition bit C; otherwise, clear condition bit C.
affects	the immediate value range to 0-31.

32-bit	
instruction operation	RZ \leftarrow RX - zero_extend(imm5); If RZ \neq 0, then C \leftarrow 1; else C \leftarrow 0;
Syntax	decne32 rz, rx, imm5
Description	Zero-extend the 5-bit immediate value to 32 bits, and store the result of subtracting the 32-bit value from RX in RZ. If the result is not equal to 0, set Set condition bit C; otherwise, clear condition bit
The flag bit	C. If the subtraction result is not equal to 0, set condition bit C; otherwise, clear condition bit C.
affects	the immediate value range to 0-31.

32-bit instruction format:



Figure 14.66: DECNE

14.44 DECT——C is 1 immediate subtraction instruction

Unified	
instruction syntax	dect rz, rx, imm5
operation if C==1, then	<p>RZ \leftarrow RX - zero_extend(imm5);</p> <p>else</p> <p>RZ \leftarrow RZ; The</p>
compilation result	only has 32-bit instructions. The result is dect32 rz, rx,
	imm5. If the condition bit C is 1, zero-extend the 5-bit immediate value to 32 bits, subtract the 32-bit value from the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit	
instruction operation	if C==1, then
	<p>RZ \leftarrow RX - zero_extend(imm5);</p> <p>else</p> <p>RZ \leftarrow RZ;</p>
Syntax	dect32 rz, rx, imm5 Description If
the condition bit C is 1, zero-extend the 5-bit immediate value to 32 bits, subtract the 32-bit value from the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain	
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit instruction format:

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RZ	RX	000011	01000	IMM5	

Figure 14.67: DECT

14.45 DIVS -- Signed Divide Instruction

Unified	
instruction syntax	divs rz, rx, ry
operation	signed division RZ = RX / RY Only 32-bit
	bit instruction divs32 rz, rx, ry exists
Compilation Result Description The signed register division instruction divides the value of register RX by the value of register RY, and the quotient is stored in RZ. The values of RY and RZ are both considered 32-bit signed numbers. Note that for 0x80 000000 divided by 0xffff, the result is undefined.	
Affects flag bit	
exception	Division by zero exception

32-bit	
instruction operation	Signed division RZ = RX / RY
Syntax	divs32 rz, rx, ry Description
The signed register division instruction divides the value of register RX by the value of register RY and stores the quotient in RZ. The values of RY and RZ are both considered 32-bit signed numbers. Note that for the case where 0x80000000 is divided by 0xffff, the result is undefined.	
Affects flag bit	
exception	Division by zero exception

32-bit instruction format:

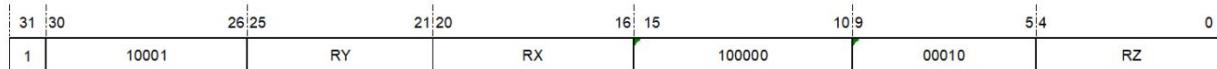


Figure 14.68: DIVS

14.46 DIVU -- Unsigned Divide Instruction

Unified instruction	
syntax	divu rz, rx, ry operation
unsigned division	RZ = RX / RY indicates
that only 32-bit instructions exist	divu32 rz, rx, ry
Description	The unsigned register division instruction divides the value of register RX by the value of register RY and stores the quotient in RZ. The values of RY and RZ are considered to be 32-bit unsigned
Affects flag bit	numbers.
exception	Division by zero exception

32-bit	
instruction operation	Unsigned division RZ = RX / RY
Syntax	divu32 rz, rx, ry Description The
unsigned register division instruction	divides the value of register RX by the value of register RY and stores the quotient in RZ. The values of RY and RZ are considered to be 32-bit unsigned
Affects flag bit	numbers.
exception	Division by zero exception

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RY	RX	100000	00001	RZ

Figure 14.69: DIVU

14.47 DOZE——Enter low power sleep mode instruction

Unified	
instruction syntax	
doze operation	enters low-power sleep mode
Compilation result	Only 32-bit instructions doze32
exist Result Description	This instruction causes the processor to enter low-power sleep mode and wait for an interrupt to exit this mode. At this time, the CPU clock stops If the device is stopped, the corresponding peripheral devices
Affects flag bit	are also stopped.
exception privilege violation exception	

32-bit	
instruction operation	to enter low-power sleep
mode Syntax	doze32
Attributes:	Privileged instruction
Description	This instruction causes the processor to enter low-power sleep mode and wait for an interrupt to exit this mode. At this time, the CPU clock stops and the corresponding peripheral devices are
Affects flag bit	also stopped. No impact
exception privilege violation exception	

32-bit instruction format:

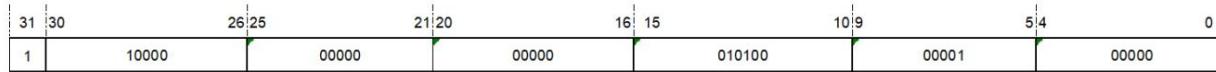


Figure 14.70: DOZE

14.48 FF0——Quickly find 0 instruction

Unified	
instruction syntax	f0
rz, rx Operation	RZ \leftarrow fnd_frst_0(RX); Compiled
only 32-bit instruction result	f0.32
rz, rx Description	Search
	for the first 0 bit in RX and return the search result to RZ. The search order is from the highest bit to the lowest bit in RX. If the highest bit in RX (RX[31]) is 0, the value returned in RZ is 0. If there is no 0 bit in RX, the value returned in RZ is 32. No effect
Affects	
flag bit	
abnormality	

32-bit	
instruction operation	RZ \leftarrow fnd_frst_0(RX);
Syntax	f0.32 rz, rx
Description	Search for the first 0 bit in RX and return the search result to RZ. The search order is from the highest bit to the lowest bit in RX. If the highest bit in RX (RX[31]) is 0, the value returned in RZ is 0. If there is no 0 bit in RX, the value returned in RZ is 32. No effect
Affects	
flag bit	
abnormality	

Instruction format:

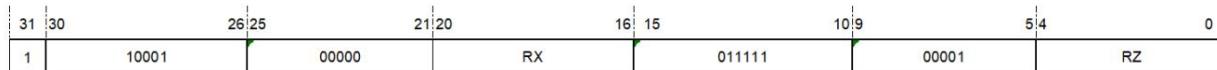


Figure 14.71: FF0

14.49 FF1—Quick Find 1 Command

Unified	
instruction syntax	f1
rz, rx Operation	RZ \leftarrow fnd_frst_1(RX); Compiled
only 32-bit instruction result	f1.32
rz, rx Description	Search
	for the first 1 bit in RX and return the search result to RZ. The search order is from the highest bit to the lowest bit in RX. If the highest bit in RX (RX[31]) is 1, the value returned in RZ is 0. If there is no 1 bit in RX, the value returned in RZ is 32. No effect
Affects	
flag bit	
abnormality	

32-bit	
instruction operation	RZ \leftarrow fnd_frst_1(RX);
Syntax	f1.32 rz, rx
Description	Search for the first 1 bit in RX and return the search result to RZ. The search order is from the highest bit to the lowest bit in RX. If the highest bit in RX (RX[31]) is 1, the value returned in RZ is 0. If there is no 1 bit in RX, the value returned in RZ is 32. No effect
Affects	
flag bit	
abnormality	

Instruction format:

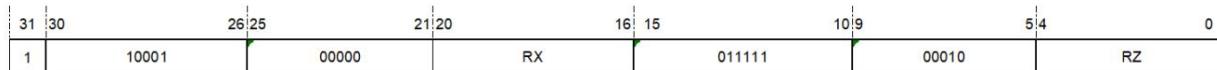


Figure 14.72: FF1

14.50 GRS——Symbol Generation Instructions

Unified	
command syntax	grs rz, label grs rz, imm32
Operation	RZ ѕ PC + sign_extend(offset << 1); Only 32-bit instructions
exist in the compilation . Result	grs32 rz, label grs32 rz, imm32
Instructions	Generate a symbol value, which is determined by the location of label or a 32-bit immediate value (IMM32). The symbol value is obtained by adding the current program PC to the 18-bit relative offset shifted left by 1 bit and signed extended to 32 bits. The valid range of the symbol value is ±256KB address
Affects	space. No effect
flag bit	
abnormality	

32-bit	
instruction operation	RZ ѕ PC + sign_extend(offset << 1); Syntax
grs rz, label grs rz, imm32	
Instructions	Generate a symbol value, which is determined by the location of label or a 32-bit immediate value (IMM32). The symbol value is obtained by adding the current program PC to the 18-bit relative offset shifted left by 1 bit and signed extended to 32 bits. The valid range of the symbol value is ±256KB address
Affects	space. No effect
flag bit	
abnormality	

Instruction format:

31	30	26	25	21	20	18	17	0
1	10011		RZ	011				Offset

Figure 14.73: GRS

14.51 IDLY——Interrupt Identification Disable Instruction

Unified instruction	
syntax	idly method
	Disable interrupt recognition 4 instructions
	There is only a 32-bit instruction idly32
	The 4 instructions following the idly instruction disable interrupt recognition, thus allowing an uninterruptible instruction sequence to be executed in a multitasking environment.
	Flag bit C is cleared after the idly instruction is executed. If an exception (including trace or breakpoint exception) occurs within 4 instructions after the idly instruction is executed, bit C is set to 1 and the interrupt instruction sequence can be observed.
operation	completion result description impact flag limit The instructions following the idly instruction can only be single-clock cycle arithmetic, logic instructions, ld, st, or branch instructions. In order to minimize the impact of some potential operation consequences, the idly instruction sequence is not to be interrupted. If there is another idly instruction in the instruction sequence following idly, it will also be ignored. However, if it is rte, rf, doze, wait, stop, etc., they will cause the idly instruction sequence to be terminated. The idly instruction is not allowed to appear in a loop with less than 8 instructions. None
Exception Notes	If the idly counter stops at a non-zero state, the interrupt will be masked. If a breakpoint exception or a trace exception occurs in the idly instruction sequence, bit C will be set to 1, and the sequence will fail. During the exception handling process, the interrupt mask is invalidated, so that the counter is cleared. The idly counter does not change during debugging using the HAD debug port. Once the processor is released from debug mode to normal operation, the counting will continue.

32-bit instruction	
operation	disable interrupt recognition 4 instruction <code>disable_int_in_following(4); Syntax description</code>
affect flag	bit limit
	The 4 instructions following the idly instruction disable interrupt recognition, thus allowing an uninterruptible instruction sequence to be executed in a multitasking environment.
	Flag bit C is cleared after the idly instruction is executed. If an exception (including trace or breakpoint exception) occurs within 4 instructions after the idly instruction is executed, bit C is set to 1 and the interrupt instruction sequence can be observed.
	The instructions following the idly instruction can only be single-clock cycle arithmetic, logic instructions, ld, st, or branch instructions. In order to minimize the impact of some potential interrupts, other instructions cannot be guaranteed not to be interrupted. If there is another idly instruction in the instruction sequence following idly, it will also be ignored. However, if it is rte, rf, doze, wait, stop, etc., they will cause the idly instruction sequence to be terminated. The idly instruction is not allowed to appear in a loop with less than 8 instructions. None
Exception Notes	If the idly counter stops at a non-zero state, the interrupt will be masked. If a breakpoint exception or a trace exception occurs in the idly instruction sequence, bit C will be set and the sequence will fail. During the exception handling process, the interrupt mask is invalidated, so that the counter is cleared. The idly counter does not change during debugging using the HAD debug port. Once the processor is released from debug mode to normal operation, the counting will continue.

32-bit instruction format:

Figure 14.74: IDLY

14.52 INCF——C is 0 immediate addition instruction

Unified	
instruction syntax	incf rz, rx, imm5
operation if C==0, then	<p>RZ \leftarrow RX + zero_extend(Imm5);</p> <p>else</p> <p>RZ \leftarrow RZ; The</p>
compilation result	only exists in 32-bit instruction
bit instruction result	incf32 rz, rx, imm5.
If the condition bit C is 0,	extend the 5-bit immediate value to 32 bits, add the 32-bit value to the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit	
instruction operation if C==0, then	<p>RZ \leftarrow RX + zero_extend(Imm5);</p> <p>else</p> <p>RZ \leftarrow RZ;</p>
Syntax	incf32 rz, rx, imm5 Description
If the condition bit C is 0,	zero-extend the 5-bit immediate value to 32 bits, add the 32-bit value to the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit instruction format:

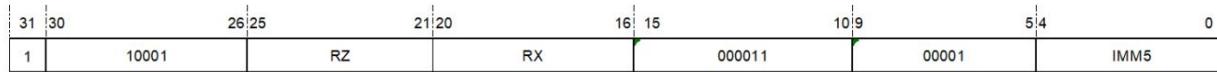


Figure 14.75: INCF

14.53 INCT——C is 1, immediate addition instruction

Unified	
instruction syntax	inct rz, rx, imm5
operation	<p>if C==1, then</p> <p>RZ \leftarrow RX + zero_extend(Imm5);</p> <p>else</p> <p>RZ \leftarrow RZ; The</p>
compilation result	only has 32-bit
instruction result	inct32 rz, rx, imm5. If
the condition	bit C is 1, extend the 5-bit immediate value to 32 bits, add the 32-bit value to the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit	
instruction operation	<p>if C==1, then</p> <p>RZ \leftarrow RX + zero_extend(Imm5);</p> <p>else</p> <p>RZ \leftarrow RZ;</p>
Syntax	inct32 rz, rx, imm5 Description
If the condition	bit C is 1, zero-extend the 5-bit immediate value to 32 bits, add the 32-bit value to the value of RX, and store the result in RZ; otherwise, the values of RZ and RX remain
The flag bit	unchanged. No effect
affects	the immediate value range to 0-31.

32-bit instruction format:

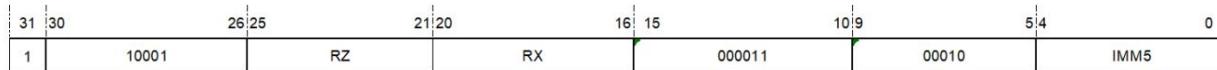


Figure 14.76: INCT

14.54 INS——Bit Insert Instruction

Unified instruction	
language	ins rz, rx, msb, lsb method
	RZ[MSB:LSB] \leftarrow RX[MSB:LSB:0]
	Only 32-bit instructions exist: ins32 rz, rx, msb, lsb
operation	Insert a continuous low-bit range of RX into a continuous bit range of RZ (RZ[MSB: LSB]) specified by two 5-bit immediate values (MSB, LSB). The other bits in RZ remain unchanged. The width of the continuous low-bit range of RX is specified by MSB and LSB (ie, RX[MSB:LSB:0]). If MSB is equal to 31 and LSB is equal to 0, the value of RZ is the same as RX. If MSB is equal to LSB, the MSB (ie, LSB) bit of RZ is the lowest bit of RX, and the other bits remain unchanged. If MSB is less than LSB, the behavior of this instruction is unpredictable. No
	effect
	The range of MSB is 0-31, the range of LSB is 0-31, and MSB should be greater than or equal to LSB.
	none
Affects flag bit restriction exception	

32-bit instruction	
	RZ[MSB:LSB] \leftarrow RX[MSB:LSB:0]
	ins32 rz, rx, msb, lsb
operation style	Insert a continuous low-bit range of RX into a continuous bit range of RZ (RZ[MSB: LSB]) specified by two 5-bit immediate values (MSB, LSB). The other bits of RZ remain unchanged. The width of the continuous low-bit range of RX is specified by MSB and LSB (ie, RX[MSB:LSB:0]). If MSB is equal to 31 and LSB is equal to 0, the value of RZ is the same as RX. If MSB is equal to LSB, the MSB (ie, LSB) bit of RZ is the lowest bit of RX, and the other bits remain unchanged. If MSB is less than LSB, the behavior of this instruction is unpredictable. No
	effect
	The range of MSB is 0-31, the range of LSB is 0-31, and MSB should be greater than or equal to LSB.
Affects flag bit restriction exception	none

32-bit instruction format:

31	30	25	25	21	20	16	15	10	9	5	4	0
1		10001		RZ		RX		010111		SIZE		LSB

Figure 14.77: INS

SIZE field:

Specifies the width of the insertion point.

Note: The SIZE of a binary operand is equal to MSB - LSB.

00000:

1

00001:

2

...

11111:

32

LSB field:

Specifies the position where the insertion ends.

00000:

0 position

00001:

1 bit

...

11111:

31 bits

14.55 IPOP——Interrupt Pop Instruction

Unified	
instruction syntax	ins rz, rx, msb, lsb
operations	RZ[MSB:LSB] \leftarrow RX[MSB-LSB:0] The
compilation result	only has 32-bit instructions ins32 rz, rx, msb, lsb

Description:	Load the interrupt general register context {R0~R3, R12, R13} from the stack pointer register, and then update the stack pointer register to the top of the stack memory. Use the stack pointer register direct addressing mode.
Affected flags:	No effect
Exceptions:	Access error exception, misalignment exception

16-bit	
instruction operation:	Load the interrupt general register context {R0~R3, R12, R13 } from the stack pointer register, and then update the stack pointer register to the top of the stack memory; {R0~R3,R12,R13} \leftarrow MEM[SP]~MEM[SP+20]; SP \leftarrow SP+24;
Syntax:	ipop16
Description:	Load the interrupt general register context {R0~R3, R12, R13 } from the stack pointer register, and then update the stack pointer register. The stack pointer register is used to directly address the memory.
Affected flags :	
Exceptions:	Access error exception, misalignment exception

Instruction format:

15	14	10	9	8	7	5	4	0
1	00101	0	0	011		00011		

Figure 14.78: IPOP

14.56 IPUSH——Interrupt Push Instruction

The unified	
instruction syntax	
ipush operation	stores the interrupted general register scene {R0~R3, R12, R13 } into the stack memory, and then updates the stack pointer register to the top of the stack memory; MEM[SP-4]~MEM[SP-24] ý(R13,R12,R3~R0); SPýSP-24; Only
Compilation results	16-bit instruction ipush exists;

Description:	Save the interrupted general register scene {R0~R3, R12, R13} to the stack memory, and then update the stack pointer register to the top of the stack memory. Use the stack pointer register direct addressing mode. No impact
Affected flags:	
Exceptions:	Access error exception, misalignment exception

16-bit	
instruction operation:	Store the interrupted general register context {R0~R3, R12, R13 } into the stack memory, and then update the stack pointer register to the top of the stack memory; MEM[SP-4]~MEM[SP-24] ý(R13,R12,R3~R0); SPýSP-24;
Syntax:	ipush16
Description:	Save the interrupt general register context {R0~R3, R12, R13} to the stack memory, and then update the stack pointer register. The stack pointer register is used to directly address the memory.
Affected flags :	
Exceptions:	Access error exception, misalignment exception

Instruction format:

15	14	10	9	8	7	5	4	0
0	00101		0	0	011		00010	

Figure 14.79: IPUSH

14.57 IXH -- Index Halfword Instruction

Unified instructions	
The	ixh rz, rx, ry
syntax	RZ \ddot{y} RX + (RY << 1)
operation compilation result	contains only 32-bit instructions. ixh32 rz, rx, ry
	Shift the value of RY one bit to the left, add the value of RX, and store the result in RZ.
Description: The impact flag	has no impact
abnormal	none

32-bit instructions	
	RZ \ddot{y} RX + (RY << 1)
	ixh32 rz, rx, ry
	Shift the value of RY one bit to the left, add the value of RX, and store the result in RZ.
Operation Syntax Description Impact Flags	No Impact
Abnormal	

32-bit instruction format:

31	30	25	25	21	20	16	15	10	9	5	4	0
1	10001		RY		RX		000010		00001		RZ	

Figure 14.80: IXH

14.58 IXW - Index Word Instruction

Unified instructions	
The	ixw rz, rx, ry
syntax	RZ \ddot{y} RX + (RY << 2)
operation compilation result	contains only 32-bit instructions. ixw32 rz, rx, ry
	Shift the value of RY left by two bits, add the value of RX, and store the result in RZ.
Description:	The impact flag has no impact
Abnormal	

32-bit instructions	
	RZ \ddot{y} RX + (RY << 2)
	ixw32 rz, rx, ry
	Shift the value of RY left by two bits, add the value of RX, and store the result in RZ.
Operation Syntax	Description Impact Flags No Impact
abnormal	none

32-bit instruction format:

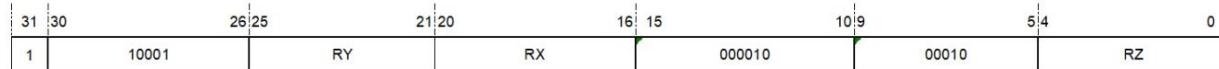


Figure 14.81: IXW

14.59 IxD -- Index Double Word Instruction

Unified instructions	
The	idx rz, rx, ry
syntax	RZ \ddot{y} RX + (RY << 3)
operation compilation result	contains only 32-bit instructions. idx32 rz, rx, ry
	Shift the value of RY left by three bits, add the value of RX, and store the result in RZ.
Description:	The impact flag has no impact
Abnormal	

32-bit instructions	
	RZ \leftarrow RX + (RY \ll 3)
	idx32 rz, rx, ry
	Shift the value of RY left by three bits, add the value of RX, and store the result in RZ.
Operation Syntax Description Impact Flags	No Impact
abnormal	none

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RY	RX	000010	00100	RZ

Figure 14.82: IXD

14.60 JMP——Jump to Register Instruction

Unified instructions	
Syntax	jmp rx
The operation	jumps to the location specified by the register PC \leftarrow RX & 0xffffe
The compilation result	is compiled into corresponding 16-bit or 32-bit instructions according to the range of registers. if (x<16), then jmp16 rx; else jmp32 rx;
Indicates that	the program jumps to the location specified by register RX, and the lowest bit of RX is ignored. The jump range of the JMP instruction is the entire 4GB Address space.
Impact Signs	No impact
Bit	
abnormal	none

16-bit	
instruction operation	jumps to the location specified by the register PC \ddagger RX & 0xffffe
Syntax jmp16 rx	Description
	The program jumps to the location specified by register RX, and the lowest bit of RX is ignored. The jump range of the JMP instruction is the entire 4GB Address space.
Affects flag bit	No impact
abnormality	

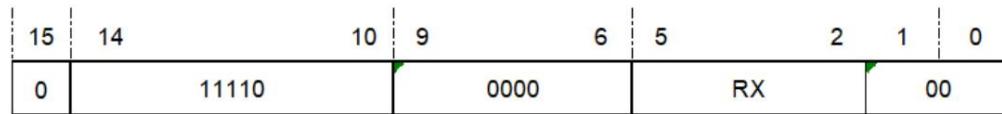
16-bit instruction format:

Figure 14.83: JMP-1

32-bit	
instruction operation	jumps to the location specified by the register PC \ddagger RX & 0xffffe
Syntax jmp32 rx	Description
	The program jumps to the location specified by register RX, and the lowest bit of RX is ignored. The jump range of the JMP instruction is the entire 4GB Address space.
Affects flag bit	No impact
abnormality	

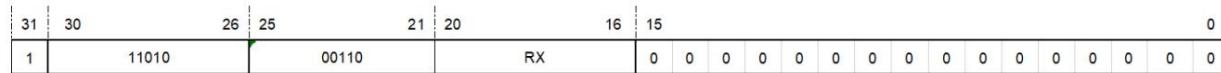
32-bit instruction format:

Figure 14.84: JMP-2

14.61 JMPI——Jump Indirect Instruction

Unified	
instruction	syntax jmpi
label	operation program jumps to the specified location in the memory PC \leftarrow MEM[(PC + zero_extend(offset << 2)) & 0xffffc] Only 32-bit instructions exist. jmpi32
The	label
compilation result shows	that the program jumps to the location of label, which is loaded from the memory. The memory address is unsignedly extended to 32 bits based on the PC plus the 16-bit relative offset shifted left by two bits, and then the lowest two bits are forced to zero. The jump range of the JMPI instruction is the entire
Affects flag bit	4GB address space. No impact
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit	
instruction	to jump to a specified location in memory PC \leftarrow MEM[(PC + zero_extend(offset << 2)) & 0xffffc]
Syntax	jmpi32 label
Description	The program jumps to the location of label, which is loaded from the memory. The memory address is unsignedly extended to 32 bits based on the PC plus the 16-bit relative offset shifted left by two bits, and then the lowest two bits are forced to zero. The jump range of the JMPI instruction is the entire 4GB address space. No effect
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

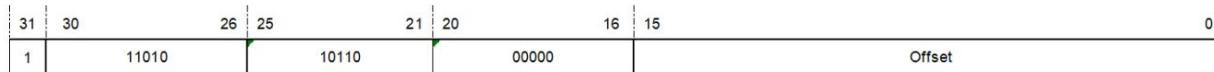


Figure 14.85: JMPI

14.62 JSR——Jump from Register to Subroutine Instruction

Unified	
instruction syntax	<code>jsr rx</code> operation link and jump to the subroutine location specified by the register R15 \leftarrow PC + 4, PC \leftarrow RX & 0xffffe
Compilation result	Compile to the corresponding 16-bit or 32-bit instruction according to the register if ($x < 16$), then <code>jsr16 rx;</code> <code>else</code> <code>jsr32 rx;</code>
Indicates	a subroutine register jump. The return address of the subroutine (the PC of the next instruction, i.e., the current PC+4) is saved in the link register R15. The program jumps to the subroutine location specified by the content of register RX for execution. The lowest bit of RX is ignored. The jump range of the JSR instruction is the entire 4GB address space.
Affects	No effect
flag bit	
abnormality	

16-bit	
instruction operations	link and jump to the subroutine location specified by the register R15 \leftarrow PC + 4, PC \leftarrow RX & 0xffffe Syntax jsr16 rx Description
Subroutine register	jump, save the subroutine return address (the PC of the next instruction, that is, the current PC + 4) in the link register R15, and the program jumps to the subroutine location specified by the content of register RX for execution. The lowest bit of RX is ignored. The jump range of the JSR instruction is the entire 4GB address space. No effect
Affects	
flag bit	
abnormality	

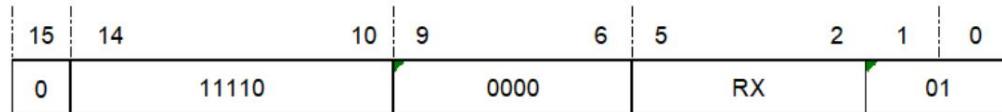
16-bit instruction format:

Figure 14.86: JSR-1

32-bit instruction	
operations	link and jump to the subroutine location specified by the register R15 \neq PC + 4, PC \neq RX & 0xffffe Syntax jsr32 rx Description
Subroutine register jump,	
save the subroutine return address (the PC of the next instruction, that is, the current PC + 4) in the link register R15, and the program jumps to the subroutine location specified by the content of register RX for execution. The lowest bit of RX is ignored. The jump range of the JSR instruction is the entire 4GB address space.	
No effect	
Affects	
flag bit	
abnormality	

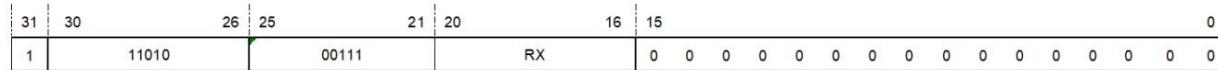
32-bit instruction format:

Figure 14.87: JSR-2

14.63 JSRI -- Jump Indirectly to Subroutine Instruction

Unified command	
language	jsri label method
operation	<p>The program jumps to the subroutine location specified in the memory</p> <p>R15 → next PC, PC →</p> <p>MEM[(PC + zero_extend(offset << 2)) & 0xffffc] Only 32-bit instructions exist. jsri32 label;</p>
Compilation Result	Subroutine indirect jump, save the return address of the subroutine (the PC of the next instruction) in the link register R15, and the program jumps to the location of the label for execution. The label is loaded from the memory. The memory address is unsigned extended to 32 bits based on the PC plus the 16-bit relative offset shifted left by two bits, and then the lowest two bits are forced to zero. The jump range of the JSRI instruction is the entire 4GB address space. No impact
Affects	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

32-bit instruction	
operation	program jumps to the subroutine location specified in the memory $R15 \leftarrow PC + 4, PC \leftarrow MEM[(PC + \text{zero_extend}(offset \ll 2)) \& 0xffffc] jsri32 \text{ label method description}$
	Subroutine indirect jump, save the return address of the subroutine (the PC of the next instruction, that is, the current PC+4) in the link register R15, and the program jumps to the location of the label for execution. The label is loaded from the memory. The memory address is unsignedly extended to 32 bits based on the PC plus the 16-bit relative offset shifted left by two bits, and then the lowest two bits are forced to zero. The jump range of the JSRI instruction is the entire 4GB address space. No impact
Affects	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
	flag bit exception

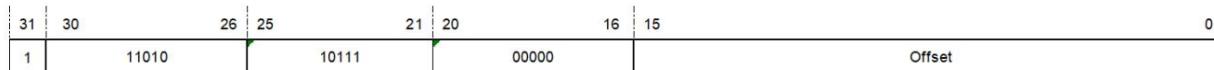
32-bit instruction format:

Figure 14.88: JSRI

14.64 LD.B -- Load Unsigned Byte Extension Instruction

Unified instruction	
syntax	ld.b rz, (rx, disp) method to
	RZ := zero_extend(MEM[RX + zero_extend(offset)])
	Compile to the corresponding 16-bit or 32-bit instruction according to the offset and register range. if (disp<32)and(x<7) and (z<7), then ld16.b rz, (rx, disp); operate the 0x compilation result ld32.b rz, (rx, disp); The byte
illustrate	loaded from the memory is zero-extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the value of the 12-bit relative offset after unsigned extension to 32 bits. The LD.B instruction can address +4KB address space. Note that the offset DISP is the binary operand Ofset. No effect
Affects	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

16-bit	
	instruction loads a byte from memory into a register, unsigned extension operation RZ ý zero_extend(MEM[RX + zero_extend(offset)]) Syntax ld16.b rz, (rx, disp) Description
	The bytes loaded from the memory are zero-extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the 5-bit relative offset unsigned extended to 32 bits. The LD16.B instruction can address the +32B address space. Note that the offset DISP is the binary operand Offset. No effect
	The register range is r0-r7.
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit restriction exception

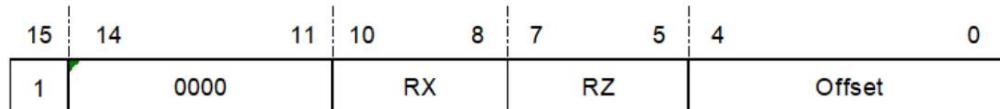
16-bit instruction format:

Figure 14.89: LD.B-1

32-bit instruction	
operation	to load a byte from memory into a register, without a sign-extend
Syntax	RZ \leftarrow zero_extend(MEM[RX + zero_extend(offset)])
Description	
	The bytes loaded from the memory are zero-extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the value of the 12-bit relative offset after unsigned extension to 32 bits. The LD32.B instruction can address +4KB address space. Note that the offset DISP is the binary operand Offset. No effect
Affects	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	10110		RZ		RX		0000		Offset	

Figure 14.90: LD.B-2

14.65 LD.BS – Load Sign-Extended Byte Instruction

Unified instruction	
syntax	ld.bs rz, (rx, disp) method operation
	RZ \leftarrow sign_extend(MEM[RX + zero_extend(offset)])
	Only 32-bit instructions exist. ld32.bs rz, (rx, disp)
compilation note	The byte loaded from the memory is signed extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the value of the 12-bit relative offset unsigned extended to 32 bits. The LD.BS instruction can address +4KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit exception

32-bit instruction	
operation	Load byte from memory into register with sign extension RZ \leftarrow sign_extend(MEM[RX + zero_extend(offset)]) Syntax ld32.bs rz, (rx, disp)
	The bytes loaded from the memory are signed extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the value of the 12-bit relative offset after unsigned extension to 32 bits. The LD32.BS instruction can address +4KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit exception

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	10110		RZ		RX		0100		Offset	

Figure 14.91: LD.BS

14.66 LD.H -- Unsigned Halfword Load Instruction

Unified instruction	
	ld.h rz, (rx, disp)
	RZ = zero_extend(MEM[RX + zero_extend(offset << 1)])
	Compile to the corresponding 16-bit or 32-bit instruction according to the offset and register range. if (disp<64)and(x<7) and (z<7), then ld16.h rz, (rx, disp); syntax operation compilation results else ld32.h rz, (rx, disp); The halfword
explain bright	loaded from memory is zero-extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX plus the 12-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LD.H instruction can address +8KB address space. Note: The offset DISP is obtained by shifting the binary operand Offset left by 1 bit. No effect
	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit exception

16-bit	
	instruction loads halfword from memory into register, unsigned extension operation RZ ý zero_extend(MEM[RX + zero_extend(offset << 1)]) Syntax ld16.h rz, (rx, disp) Description
	The halfword loaded from the memory is zero-extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the 5-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LD16.H instruction can address +64B of the address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by 1 bit. No effect
	The register range is r0-r7.
	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit restriction exception

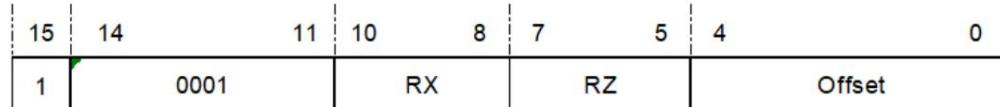
16-bit instruction format:

Figure 14.92: LD.H-1

32-bit instruction	
	loads a halfword from memory into a register, without a sign-extend
operation	RZ \leftarrow zero_extend(MEM[RX + zero_extend(offset << 1)])
	Syntax ld32.h rz, (rx, disp) Description
	The halfword loaded from the memory is zero-extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the 12-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LD32.H instruction can address +8KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by 1 bit. No effect
	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Affects	flag bit exception

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	10110		RZ		RX		0001		Offset	

Figure 14.93: LD.H-2

14.67 LD.HS – Load Sign-Extended Halfword Instruction

Unified instruction	
syntax	<code>ld.hs rz, (rx, disp) method operation</code>
	<code>RZ ← sign_extend(MEM[RX + zero_extend(offset << 1)])</code>
	<p>Only 32-bit instructions exist.</p> <p><code>ld32.hs rz, (rx, disp)</code></p>
compilation details description	<p>The halfword loaded from the memory is signed extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the 12-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LD.HS instruction can address +8KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by 1 bit. No effect</p>
	<p>Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception</p> <p>Affects flag bit exception</p>

32-bit instruction	
operation	Load halfword from memory into register with sign extension <code>RZ ← sign_extend(MEM[RX + zero_extend(offset << 1)])</code>
	Syntax <code>ld32.hs rz, (rx, disp)</code> Description
	<p>The halfword loaded from the memory is signed extended to 32 bits and stored in register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX to the 12-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LD32.HS instruction can address +8KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by 1 bit. No effect</p>
	<p>Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception</p> <p>Affects flag bit exception</p>

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	10110	RZ	RX	0101					Offset	

Figure 14.94: LD.HS

14.68 LD.W - Word Load Instruction

Unified instruction	
syntax	ldw rz, (rx, disp) method to operate
the	RZ \leftarrow MEM[RX + zero_extend(offset << 2)]
compilation	Compile to the corresponding 16-bit or 32-bit instruction according to the offset and register range. if (x=sp) and (z<7) and (disp < 1024), ld16.w rz, (sp, disp); else if (disp<128) and (x<7) and (z<7), ld16.w rz, (rx, disp); else ld32.w rz, (rx, disp); Load word
explain	from memory into register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX plus the 12-bit relative offset shifted left by two bits and unsigned extended to 32 bits. The LD.W instruction can address +16KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by two bits. No effect
bright	
Affects	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

16-bit	
	instruction load word from memory to register RZ register RZ \leftarrow MEM[RX + sign_extend(offset << 2)] ld16.w rz , (rx, disp) loads word from memory to register RZ. Register plus immediate offset
	addressing is used. When RX is SP, the effective address of the memory descriptor is obtained by adding the base register RX plus the 8-bit relative offset shifted left by 2 bits and unsigned extended to 32 bits. When rx is any other register, the effective address of the memory is obtained by adding the base register RX plus the 5-bit relative offset shifted left by 2 bits and unsigned extended to 32 bits. The LD16.W instruction can address +1KB of address space. Note that the offset DISP is obtained by shifting the binary operand IMM5 left by two bits. When the base register RX is SP, the offset DISP is obtained by shifting the binary operand {IMM3, IMM5} left by two bits. No effect
	Impact flag
limit	The register range is r0-r7.
Control Abnormality	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception.

16-bit instruction format:

ld16.w rz, (rx, disp)

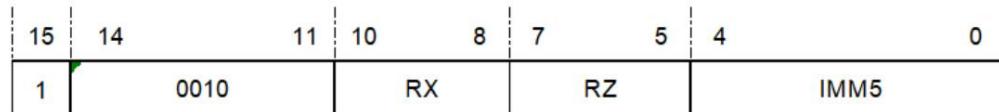


Figure 14.95: LD.W-1

ld16.w rz, (sp, disp)

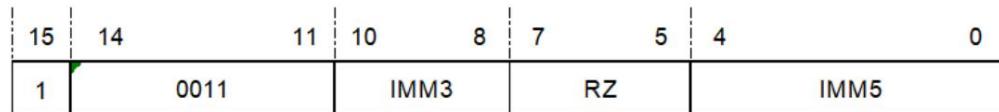


Figure 14.96: LD.W-2

32-bit instruction	
operation	load word from memory to register RZ ý MEM[RX + zero_extend(offset << 2)] Syntax ld32.w rz, (rx, disp)
Description	
	Load word from memory into register RZ. Register plus immediate offset addressing is used. The effective address of the memory is obtained by adding the base register RX plus the 12-bit relative offset shifted left by two bits and unsigned extended to 32 bits. The LD32.W instruction can address +16KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by two bits. No effect
Affects	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	10110		RZ		RX		0010		Offset	

Figure 14.97: LD.W-3

14.69 LDM——Continuous Load Multiple Words Instruction

Unified instruction	
syntax	ldm ry-rz, (rx) method operation
	<p>Load consecutive words from memory into a consecutive register file dst \leftarrow Y; addr \leftarrow RX; for (n = 0; n <= (ZY); n++) { Rdst \leftarrow MEM[addr]; dst \leftarrow dst + 1; addr \leftarrow addr + 4; } Only 32-bit instructions exist.</p> <p>ldm32 ry-rz, (rx);</p>
Compilation <small>Result Description</small>	Load multiple consecutive words from the memory into a continuous register stack starting from register RY, that is, load the first word starting from the specified address of the base register RX. No effect
Impact flag	
limit	RZ should be greater than or equal to RY. The base register RX should not be included in the RY-RZ range, otherwise the result is unpredictable. Unaligned
<small>Control Abnormality</small>	access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB read invalid exception

32-bit	
	instruction to load consecutive words from memory into a consecutive register file operation dst \ddagger Y; addr \ddagger RX; for (n = 0; n <= IMM5; n++) { Rdst \ddagger MEM[addr]; dst \ddagger dst + 1; addr \ddagger addr + 4; } ldm32 ry-rz, (rx)
Syntax	Load multiple consecutive words from the memory into a continuous register stack starting from register RY, that is, load the first word starting from the specified address of the memory into register RY, the second word into register RY+1, and so on, and the last word into register RZ. The effective address of the memory is determined by the content of the base register RX.
Impact flag	No effect
Control Abnormality	unpredictable. Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB read invalid exception

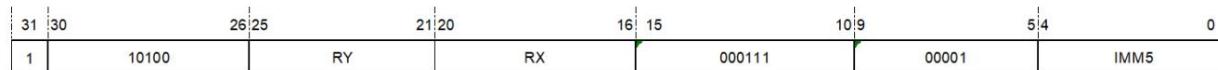
32-bit instruction format:

Figure 14.98: LDM

IMM5 domain:

Specify the number of target registers, IMM5 = Z - Y.

00000:

1 destination register

00001:

2 destination registers

...

11111:

32 destination registers

14.70 LDQ -- Continuous Four-Word Load Instruction

Unified instruction	
syntax	<code>ldq r4-r7, (rx) method</code>
operation	<p>Load four consecutive words from memory into registers R4-R7 dst \ddagger 4; addr \ddagger RX; for $(n = 0; n <= 3; n++) \{ Rdst \ddagger MEM[addr];$ $dst \ddagger dst + 1; addr \ddagger addr + 4; \}$ Only 32-bit instructions exist. <code>ldq32 r4-r7, (rx);</code></p>
Compilation Rule	Load the first word into register R5, the second word into register R6, the third word into register R5, and the fourth word into register R7. The effective address of the memory is determined by the content of the base register RX. Note that this instruction is a pseudo-instruction of <code>ldm r4-r7, (rx)</code> . No effect
	The base register RX should not be included in the range R4-R7, otherwise the results are unpredictable.
Affects flag bit restriction exception	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit	
Operation	instruction loads four consecutive words from memory into registers R4-R7 Operation dst \ddagger 4; addr \ddagger RX; for (n = 0; n \leq 3; n++) { Rdst \ddagger MEM[addr]; dst \ddagger dst + 1; addr \ddagger addr + 4; } ldq32 r4-r7, (rx)
Syntax	Load 4 consecutive words from memory into register file [R4, R7] (including boundaries), that is, load the first word starting from the specified address of memory into register R4, the second word into register R5, the third word into register R6, and the fourth word into register R7. The effective address of the memory is determined by the content of the base register RX. Note: This instruction is a pseudo-instruction of ldm32 r4-r7, (rx). No effect
	The base register RX should not be included in the range R4-R7, otherwise the results are unpredictable.
Affects	flag bit restriction exception Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

Figure 14.99: LDQ

14.71 LDR.B – Shift Register Addressing Unsigned Extended Byte Load Instruction

Unified	
Instruction Syntax	ldr.b rz, (rx, ry << 0) ldr.b rz, (rx, ry << 1) ldr.b rz, (rx, ry << 2) ldr.b rz, (rx, ry << 3) Operation Load byte from memory into
register, no sign extension	RZ Ѽ zero_extend(MEM[RX + RY << IMM2]) Only 32-bit instructions exist.
Compilation	ldr32.b rz, (rx, ry << 0) ldr32.b rz, (rx, ry << 1) ldr32.b rz, (rx, ry << 2) ldr32.b rz, (rx, ry << 3) Description The byte loaded from the memory is zero-extended to 32
bits and stored in register RZ. Register plus register shift addressing is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted 2 bits to the left by the immediate value IMM2. The default value of IMM2 is 0. No effect	
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit	
instruction operation	loads bytes from memory into registers, without sign extension RZ Ѽ zero_extend(MEM[RX + RY << IMM2]) Syntax ldr32.b rz, (rx, ry << 0)
ldr32.b rz, (rx, ry << 1) ldr32.b rz, (rx, ry << 2) ldr32.b rz, (rx, ry << 3) Description The byte loaded from the memory is zero-extended to 32 bits and stored	
in register	RZ. Register plus register shift addressing is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted 2 bits to the left by the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

ldr32.b rz, (rx, ry << 0)

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10100	RY	RX	000000	00001	RZ	

Figure 14.100: LDR.B-1

ldr32.b rz, (rx, ry << 1)

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10100	RY	RX	000000	00010	RZ	

Figure 14.101: LDR.B-2

ldr32.b rz, (rx, ry << 2)

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10100	RY	RX	000000	00100	RZ	

Figure 14.102: LDR.B-3

ldr32.b rz, (rx, ry << 3)

14.72 LDR.BS – Shift Register Addressed Sign-Extended Byte Load Instruction

Unified	
Instruction Syntax	ldr.bs rz, (rx, ry << 0) ldr.bs rz, (rx, ry << 1) ldr.bs rz, (rx, ry << 2) ldr.bs rz, (rx, ry << 3) Operation Load byte from
memory	into register, sign-extend RZ = sign_extend(MEM[RX + RY << IMM2]) Only 32-bit instructions
Compilation results	exist. ldr32.bs rz, (rx, ry << 1) ldr32.bs rz, (rx, ry << 2) ldr32.bs rz, (rx, ry << 3) Description The byte loaded from the memory is sign-
extended	to 32 bits and stored in register RZ. Register plus register shift addressing is used. The effective address of the memory is obtained by adding the base register RX to the value of the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000000	01000	RZ

Figure 14.103: LDR.B-4

32-bit	
instruction operation	loads byte from memory into register, sign-extended
	RZ \leftarrow sign_extend(MEM[RX + RY << IMM2]) Syntax ldr32.bs rz, (rx, ry << 0) ldr32.bs rz, (rx, ry << 1) ldr32.bs rz, (rx, ry << 2) ldr32.bs rz, (rx, ry << 3) Description The byte loaded from the memory is sign-
extended	extended to 32 bits and stored in register RZ. Register plus register shift addressing is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

ldr32.bs rz, (rx, ry<< 0)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000100	00001	RZ

Figure 14.104: LDR.BS-1

ldr32.bs rz, (rx, ry<< 1)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000100	00010	RZ

Figure 14.105: LDR.BS-2

ldr32.bs rz, (rx, ry<< 2)

ldr32.bs rz, (rx, ry<< 3)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000100	00100	RZ

Figure 14.106: LDR.BS-3

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000100	01000	RZ

Figure 14.107: LDR.BS-4

14.73 LDR.H -- Shift Register Unsigned Halfword Load Instruction

Unified	
Instruction Syntax	ldr.h rz, (rx, ry << 0) ldr.h rz, (rx, ry << 1) ldr.h rz, (rx, ry << 2) ldr.h rz, (rx, ry << 3) Operation Load halfword from memory into
register, no sign extension.	RZ \leftarrow zero_extend(MEM[RX + RY << IMM2]) Only 32-bit instructions exist.
Compilation note	ldr32.h rz, (rx, ry << 0) ldr32.h rz, (rx, ry << 1) ldr32.h rz, (rx, ry << 2) ldr32.h rz, (rx, ry << 3) Description The half-word loaded from the memory is zero-extended
to 32 bits	and stored in register RZ. Register plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit	
instruction	operation loads halfword from memory into register, without sign extension RZ \leftarrow zero_extend(MEM[RX + RY << IMM2]) Syntax ldr32.h rz, (rx, ry)
<< 0)	ldr32.h rz, (rx, ry << 1) ldr32.h rz, (rx, ry << 2) ldr32.h rz, (rx, ry << 3) Description The half-word loaded from the memory is zero-extended
to 32 bits	and stored in register RZ. Register plus register shift addressing is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits and the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

ldr32.h rz,(rx, ry << 0)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000001	00001	0001	RZ

Figure 14.108: LDR.H-1

ldr32.h rz,(rx, ry << 1)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000001	00010	00100	RZ

Figure 14.109: LDR.H-2

ldr32.h rz,(rx, ry << 2)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000001	00100	00100	RZ

Figure 14.110: LDR.H-3

ldr32.h rz,(rx, ry << 3)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000001	01000	RZ

Figure 14.111: LDR.H-4

14.74 LDR.HS – Shift Register Sign-Extend Load Halfword Instruction

Unified	
Instruction Syntax	ldr.hs rz, (rx, ry << 0) ldr.hs rz, (rx, ry << 1) ldr.hs rz, (rx, ry << 2) ldr.hs rz, (rx, ry << 3)
Operation	Load halfword from memory into register, sign-extend RZ = sign_extend(MEM[RX + RY << IMM2]) Only 32-bit instructions exist.
Compilation result	ldr32.hs rz, (rx, ry << 0) ldr32.hs rz, (rx, ry << 1) ldr32.hs rz, (rx, ry << 2) ldr32.hs rz, (rx, ry << 3) Description The half-word loaded from the memory is sign-extended to
Affects	32 bits and stored in register RZ. Register plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit	
	instruction operation loads halfword from memory into register, sign-extended RZ ý sign_extend(MEM[RX + RY << IMM2]) Syntax ldr32.hs rz, (rx, ry << 0) ldr32.hs rz, (rx, ry << 1) ldr32.hs rz, (rx, ry << 2) ldr32.hs rz, (rx, ry << 3) Description The half-word loaded from the memory is sign-extended
to 32 bits	and stored in register RZ. Register plus register shift addressing is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted 2 bits to the left by the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

ldr32.hs rz, (rx, ry << 0)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000101	00001	RZ	

Figure 14.112: LDR.HS-1

ldr32.hs rz, (rx, ry << 1)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000101	00010	RZ	

Figure 14.113: LDR.HS-2

ldr32.hs rz, (rx, ry << 2)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000101	00100	RZ	

Figure 14.114: LDR.HS-3

ldr32.hs rz, (rx, ry << 3)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000101	01000	RZ

Figure 14.115: LDR.HS-4

14.75 LDR.W – Register Shift Addressed Word Load Instruction

Unified	
Instruction Syntax	ldr.w rz, (rx, ry << 0) ldr.w rz, (rx, ry << 1) ldr.w rz, (rx, ry << 2) ldr.w rz, (rx, ry << 3)
Operation	Load word from memory into register RZ \ddagger MEM[RX + RY << IMM2] Only 32-bit instructions exist. ldr32.w rz, (rx, ry << 0) ldr32.w rz, (rx, ry Compiling results) ldr32.w rz, (rx, ry << 2) ldr32.w rz, (rx, ry << 3) Description Load word from memory into register RZ. Register plus register shift addressing
	is used. The effective address of the memory is obtained by adding the base address register RX to the offset register RY and shifting the immediate value IMM2 left by 2 bits. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit	
instruction operation	Load word from memory to register RZ \ddagger MEM[RX + RY << IMM2] Syntax
ldr32.w rz, (rx, ry << 0) ldr32.w rz, (rx, ry << 1) ldr32.w rz, (rx, ry << 2) ldr32.w rz, (rx, ry << 3) Description Load word from memory to register RZ. Register	
plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect	
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

ldr32.w rz, (rx, ry << 0)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000010	00001	RZ	

Figure 14.116: LDR.W-1

ldr32.w rz, (rx, ry << 1)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000010	00010	RZ	

Figure 14.117: LDR.W-2

ldr32.w rz, (rx, ry << 2)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000010	00100	RZ	

Figure 14.118: LDR.W-3

ldr32.w rz, (rx, ry << 3)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10100	RY	RX	000010	01000	RZ

Figure 14.119: LDR.W-4

14.76 LRS.B - Byte Signed Load Instruction

Unified instruction	
	lrs.b rz, [label]
	Load byte from memory into register RZ zero_extend(MEM[R28 + zero_extend(offset)]) Only 32-bit instructions exist. lrs32.b rz,
	[label]
	Load the byte symbol at the location of label, zero-extend to 32 bits and store it in the destination register RZ. Use register plus immediate offset addressing. The syntax operation, address, offset, exception is obtained by adding the base register R28 to the value of the 18-bit relative offset after unsigned extension to 32 bits. The LRS.B instruction can address +256KB address space. Note: The offset DISP is the binary operand Offset. No effect
Affects	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

32-bit instruction	
RZ	loads a byte from memory into a register, unsigned extension operation ÿ zero_extend(MEM[R28 + zero_extend(offset)]) Syntax Description lrs32.b rz, [label]
	Load the byte symbol at the location of label, zero-extend to 32 bits and store it in the destination register RZ. Use register plus immediate offset addressing. The effective address of the memory is obtained by adding the base register R28 to the value of the 18-bit relative offset after unsigned extension to 32 bits. The LRS.B instruction can address +256KB address space. Note: The offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Affects	flag bit exception

32-bit instruction format:

Figure 14.120: LRS.B

14.77 LRS.H -- Load Half-Word Instruction

Unified instruction	
syntax	lrs.h rz, [label] method
	RZ ← zero_extend(MEM[R28 + zero_extend(offset << 1)])
	Only 32-bit instructions exist. lrs32.h rz, [label]
operation	Load the half-word number at the location of label, zero-extend it to 32 bits and store it in the destination register RZ. Use register plus immediate offset addressing. The effective address of the memory is obtained by adding the base register R28 plus the 18-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LRS.H instruction can address +512KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit exception

32-bit instruction	
operation	loads a half-word from memory into a register, unsigned extension RZ ← zero_extend(MEM[R28 + zero_extend(offset << 1)]) Syntax Description
	Load the half-word number at the location of label, zero-extend it to 32 bits and store it in the destination register RZ. Use register plus immediate offset addressing. The effective address of the memory is obtained by adding the base register R28 plus the 18-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The LRS.H instruction can address +512KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit exception

32-bit instruction format:

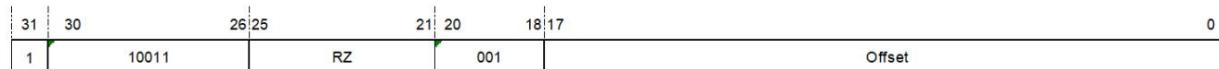


Figure 14.121: LRS.H

14.78 LRS.W——Word Load Instruction

Unified instruction	
	lrs.w rz, [label]
	RZ \leftarrow zero_extend(MEM[R28 + zero_extend(offset << 2)])
	Only 32-bit instructions exist. lrs32.w rz, [label]
<small>syntax operand order: base register, offset, disp, result</small>	Load the word number at the location of label, and store it in the destination register RZ after zero extension to 32 bits. Use register plus immediate bits and unsigned extended to 32 bits. The LRS.W instruction can address +1024KB address space. Note that the offset DISP is the binary operand Offset. No effect
<small>Affects flag bit exception</small>	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction	
	to load a word from memory into a register, unsigned extension
operation	RZ ý zero_extend(MEM[R28 + zero_extend(offset << 2)]) Syntax Description lrs32.w rz, [label]
	Load the word number at the location of label, and store it in the destination register RZ after zero extension to 32 bits. Use register plus immediate offset addressing mode. The effective address of the memory is obtained by adding the base register R28 plus the 18-bit relative offset shifted left 2 bits and unsigned extended to 32 bits. The LRS.W instruction can address +1024KB address space. Note: The offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Affects	flag bit exception

32-bit instruction format:

Figure 14.122: LRS.W

14.79 LRW - Memory Read Instruction

Unified instruction	
syntax	lrw rz, label lrw rz, imm32 operation loads word
from memory to register	RZ \leftarrow zero_extend(MEM[(PC + zero_extend(offset << 2)) & 0xffffc]) is compiled into the corresponding 16-bit or 32-bit
Compilation result	instruction according to the load range if(offset<512B), then lrw16 label; lrw16 imm32; else lrw32 label; lrw32 imm32;
Description	Load the word at the location of label, or a 32-bit immediate value (IMM32) to the destination register RZ. The memory address is obtained by adding the relative offset of the left shift of two bits to the PC, unsignedly extending it to 32 bits, and then forcing the lowest two bits to zero. The loading range of the LRW instruction is the entire 4GB address space. No
Affects	effect
flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

16-bit instruction	
	Load word from memory into register RZ \leftarrow zero_extend(MEM[(PC + zero_extend(offset << 2)) & 0xffffc]) lrw16 rz, label lrw16 rz, imm32 Load the word at label, or
	a 32-bit immediate value (IMM32) into
operation	destination register RZ. The memory address is obtained by adding the 10-bit relative offset shifted left by two bits to the PC, unsigned extended to 32 bits, and lowest two bits to zero. The load range of the LRW instruction is the entire 4GB address space. Note that the relative offset Offset is equal to the binary encoding {IMM2, IMM5}. No effect
Affects	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
flag bit exception	

16-bit instruction format:

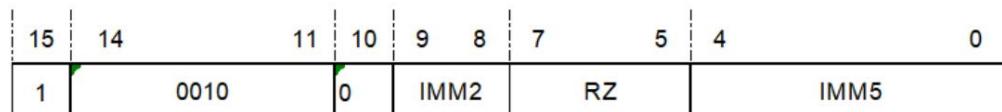


Figure 14.123: LRW-1

32-bit	
instruction operation	loads a word from memory into a register RZ = zero_extend(MEM[(PC + zero_extend(offset << 2)) & 0xffffc])
Syntax	lw32 rz, label lw32 rz, imm32
Description	Load the word at label, or 32-bit immediate value (IMM32) to destination register RZ. The memory address is obtained by adding the 16-bit relative offset shifted left by two bits to PC, unsigned extended to 32 bits, and then forcing the lowest two bits to zero. The load range of the LRW instruction is the entire 4GB address space. No
Affects	effect
flag	
bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

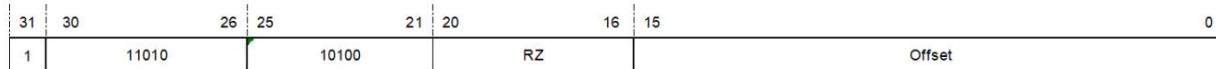
32-bit instruction format:

Figure 14.124: LRW-2

14.80 LSL -- Logical Shift Left Instruction

Unified	
instruction syntax	<code>lsl</code>
rz, rx Operation	$RZ \leftarrow RZ << RX[5:0]$
Compiled	into corresponding 16-bit or 32-bit result instruction according to the register range. if ($x<16$) and ($z<16$), then <code>lsl16 rz, rx;</code> <code>else</code> <code>lsl32 rz, rx;</code>
Description	For <code>lsl rz, rx</code> , the value of RZ is logically shifted left (the original value is shifted left, and 0 is shifted to the right), and the result is stored in RZ . The number of left shifts is determined by the value of the lower 6 bits of RX ($RX[5:0]$); if the value of $RX[5:0]$ is greater than 31, then RZ will be cleared; For <code>lsl rz, rx, ry</code> , the value of RX is logically shifted left (the original value is shifted left, and 0 is shifted to the right), and the result is stored in RZ . The number of left shifts is determined by the value of the lower 6 bits of RY ($RY[5:0]$); if the value of $RY[5:0]$ is greater than 31, then RZ will be cleared.
Affects	cleared. No effect
flag bit	
abnormality	

16-bit	
instruction operation	$RZ \leftarrow RZ << RX[5:0]$
Syntax	<code>lsl16 rz, rx</code>
Description	Perform a logical left shift on the value of RZ (shift the original value to the left and shift 0 to the right), and store the result in RZ . The number of left shifts is determined by the value of the lower 6 bits of RX ($RX[5:0]$); if the value of $RX[5:0]$ is greater than 31, RZ will be cleared.
The range of the	will be cleared. No effect
affected flag limit	register is r0- r15 .

16-bit instruction format:

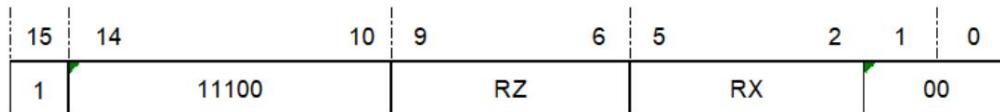


Figure 14.125: LSL-1

32-bit	
instruction	operation RZ \leftarrow RX << RY[5:0]
Syntax	lsl32 rz, rx, ry
Description	Perform a logical left shift on the value of RX (shift the original value to the left and shift 0 to the right), and store the result in RZ. The number of left shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 31, RZ
Affects	will be cleared. No effect
flag bit	
abnormality	

32-bit instruction format:



Figure 14.126: LSL-2

14.81 LSLC - Shift Immediate Logical Left to C Instruction

Unified	
instruction syntax	lslc rz, rx, oimm5
Operation	RZ \leftarrow RX << OIMM5, C \leftarrow RX [32 – OIMM5]
Compilation result	Only 32-bit instructions exist. lslc32 rz, rx, oimm5
Description	Shift the value of RX logically to the left (shift the original value to the left and shift 0 to the right), store the last bit shifted out in the conditional bit C, and store the shift result in RZ. The number of left shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then the conditional bit C is the lowest bit of RX and RZ is cleared.
The flag bit	C \leftarrow RX[32 – OIMM5]
affects	the immediate value range to 1-32.

32-bit instruction	
set	RZ \leftarrow RX << OIMM5, C \leftarrow RX[32 – OIMM5] for syntax description
	lslc32 rz, rx, oimm5
	Shift the value of RX to the left logically (shift the original value to the left and shift 0 to the right), store the last bit shifted out in the condition bit C, and store the shift result in RZ. The number of left shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then the condition bit C is the lowest bit of RX and RZ is cleared. Note: The binary operand IMM5 is equal to OIMM5 - 1.
	C \leftarrow RX[32 – OIMM5]
	The range of immediate value is 1-32.
	none
Affects flag bit restriction exception	

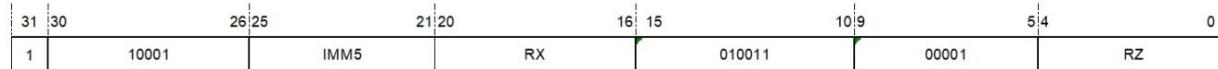
32-bit instruction format:

Figure 14.127: LSLC

IMM5 domain:

Specifies the value of an immediate value without an offset.

Note: The shift value OIMM5 must be offset by 1 compared to the binary operand IMM5.

00000:

Shift 1 bit

00001:

Shift 2 places

...

11111:

Shift 32 bits

14.82 LSLI – Logical Shift Left Immediate Instruction

Unified	
instruction	syntax lsli rz, rx, imm5
operation	RZ \leftarrow RX << IMM5 compilation
result	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 8$) and ($z < 8$), then lsli16 rz, rx, imm5 else lsli32 rz, rx, imm5
Description:	Perform a logical left shift on the value of RX (the original value is shifted left, and 0 is shifted to the right), and the result is stored in RZ. The number of left shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is equal to 0, the value of RZ will remain unchanged.
The flag bit	No effect
affects the	immediate value range to 0-31.

16-bit	
instruction	operation RZ \leftarrow RX << IMM5
Syntax	lsli16 rz, rx, imm5 Description
	Perform a logical left shift on the value of RX (shift the original value to the left and shift 0 to the right), and store the result in RZ. The number of left shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is 0, the value of RZ
The range of	will remain unchanged. No effect
registers that affect flag limits	r0-r7; the range of immediate values is 0-31.
Abnormal	

16-bit instruction format:

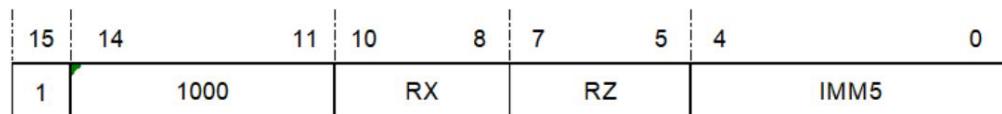


Figure 14.128: LSLI-1

32-bit	
instruction	operation RZ \leftarrow RX << IMM5
Syntax	lsli32 rz, rx, imm5 Description
	Perform a logical left shift on the value of RX (shift the original value to the left and shift 0 to the right), and store the result in RZ. The number of bits shifted to the left is increased by 5 bits. If the value of IMM5 is equal to 0, the value of RZ will be the same as RX.
The flag bit	
affects	the immediate value range to 0-31.

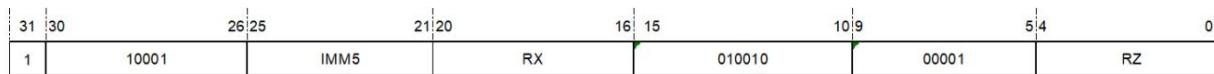
32-bit instruction format:

Figure 14.129: LSLI-2

14.83 LSR -- Logical Shift Right Instruction

Unified instruction		
syntax	<code>lsr rz, rx</code> method	<code>lsr rz, rx, ry</code>
	<code>RZ ѕ RZ >> RX[5:0]</code>	<code>RZ ѕ RX >> RY[5:0]</code>
operation	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($z<16$) and ($x<16$), then <code>lsr16 rz,</code> else <code>lsr32 rz, rz, rx;</code> For <code>lsr</code>	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x==z$) and ($z<16$) and ($y<16$), then <code>lsr16 rz, ry;</code> else <code>lsr32 rz, rx, ry;</code>
illustrate	<code>rz, rx</code> , perform a logical right shift on the value of <code>RZ</code> (shift the original value to the right and shift 0 to the left), and store the result in <code>RZ</code> . The number of right shifts is determined by the value of the lower 6 bits of <code>RX</code> (<code>RX[5:0]</code>); if the value of <code>RX[5:0]</code> is greater than 31, <code>RZ</code> will be cleared. For <code>lsr rz, rx, ry</code> , perform a logical right shift on the value of <code>RX</code> (shift the original value to the right and shift 0 to the left), and store the result in <code>RZ</code> . The number of right shifts is determined by the value of the lower 6 bits of <code>RY</code> (<code>RY[5:0]</code>); if the value of <code>RY[5:0]</code> is greater than 31, <code>RZ</code> will be cleared. No effect	
	none	
Affects flag bit exception		

16-bit	
instruction	<code>operation RZ ѕ RZ >> RX[5:0]</code>
Syntax	<code>lsr16 rz, rx</code> Description
	Perform a logical right shift on the value of <code>RZ</code> (the original value is shifted right and 0 is shifted into the left side). The result is stored in <code>RZ</code> . The number of right shifts is determined by the value of the lower 6 bits of <code>RX</code> (<code>RX[5:0]</code>); if the value of <code>RX[5:0]</code> is greater than 31, <code>RZ</code> will be cleared. No effect
The range of the	31, <code>RZ</code> will be cleared. No effect
affected flag limit	register is r0- r15 .

16-bit instruction format:

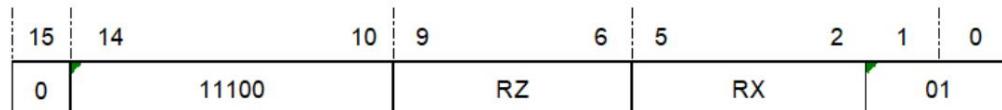


Figure 14.130: LSR-1

32-bit	
instruction	operation RZ $\ddot{\vee}$ RX >> RY[5:0]
Syntax	lsr32 rz, rx, ry
Description	Perform a logical right shift on the value of RX (shift the original value to the right and shift 0 to the left), and store the result in RZ. The number of right shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 31,
Affects	RZ will be cleared. No effect
flag bit	
abnormality	

32-bit instruction format:



Figure 14.131: LSR-2

14.84 LSRC - Immediate Logical Shift Right to C Instruction

Unified	
instruction syntax	lsrc rz, rx, oimm5
Operation	RZ $\ddot{\vee}$ RX >> OIMM5, C $\ddot{\vee}$ RX [OIMM5 - 1] Only 32-bit
	instructions exist. lsrc32 rz, rx, oimm5
Compilation result description:	Perform logical right shift on the value of RX (shift the original value to the right and shift 0 to the left), store the last bit shifted out in condition bit C, store the shift result in RZ, and the number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then condition bit C is the highest bit of RX and RZ is cleared.
The flag bit	C $\ddot{\vee}$ RX[OIMM5 - 1]
affects	the immediate value range to 1-32.

32-bit instruction	
set	RZ \leftarrow RX >> OIMM5, C \leftarrow RX[OIMM5 - 1] for syntax description
	lsrc32 rz, rx, oimm5
	Shift the value of RX right logically (shift the original value right and shift 0 in the left side), store the last bit shifted out in the condition bit C, and store the shift result in RZ. The number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then the condition bit C is the highest bit of RX and RZ is cleared. Note: The binary operand IMM5 is equal to OIMM5 - 1.
	C \leftarrow RX[OIMM5 - 1]
	The range of immediate value is 1-32.
	none
Affects flag bit restriction exception	

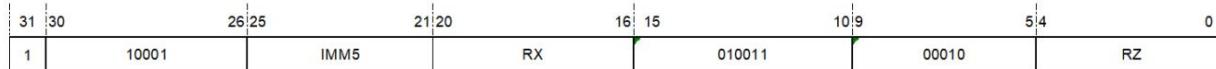
32-bit instruction format:

Figure 14.132: LSRC

IMM5 domain:

Specifies the value of an immediate value without an offset.

Note: The shift value OIMM5 must be offset by 1 compared to the binary operand IMM5.

00000:

Shift 1 bit

00001:

Shift 2 places

...

11111:

Shift 32 bits

14.85 LSRI -- Logical Shift Right Immediate Instruction

Unified	
instruction	syntax lsri rz, rx, imm5
operation	RZ \downarrow RX >> IMM5 compilation
result	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 8$) and ($z < 8$), then lsri16 rz, rx, imm5 else lsri32 rz, rx, imm5
Description:	Perform a logical right shift on the value of RX (the original value is shifted right, and 0 is shifted into the left side), and the result is stored in RZ. The number of right shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is equal to 0, the value of RZ remains unchanged or will be the same
The flag bit	as RX. No effect
affects the	immediate value range to 0-31.

16-bit	
instruction	operation RZ \downarrow RX >> IMM5 Syntax
lsri16 rz, rx, imm5	Description Perform a logical right shift on the value of RX (the original value is shifted right, and 0 is shifted to the left), and the result is stored in RZ. The number of right shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is equal to 0, the value of RZ will remain unchanged. No effect
The range of the	affected flag limit register is r0-r7; the range of the immediate value is 0-31 .

16-bit instruction format:

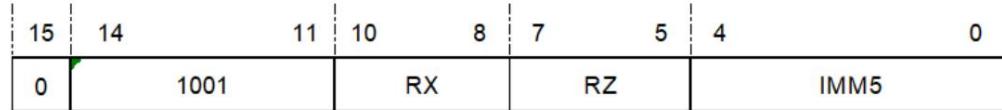


Figure 14.133: LSRI-1

32-bit instructions	
Operation	RZ \leftarrow RX >> IMM5
Syntax	lsri32 rz, rx, imm5
Description:	Shift the value of RX to the right logically (shift the original value to the right and shift 0 to the left), store the result in RZ, and shift the number of bits to the right from the 5-bit immediate value If the value of IMM5 is equal to 0, the value of RZ will be the same as RX.
Influence	No impact
Logo	
Bit	
Limits	the immediate value to the range 0-31.
Abnormal	

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	IMM5	RX	010010	00010	RZ

Figure 14.134: LSRI-2

14.86 MFCR——Control Register Read Transfer Instruction

Unified instructions	
Syntax	mfcr rz, cr<x, sel>
Operations	Transfer the contents of the control register to the general register RZ \leftarrow CR<x, sel>
The compiled result	contains only 32-bit instructions. mfcr32 rz, cr<x, sel>
Attribute:	Privileged instruction
	Transfer the contents of the control register CR<x, sel> to the general register RZ.
Description:	The impact flag has no impact
Exception	Privilege Violation Exception

32-bit instructions	
operate	Transfer the contents of the control register to the general register RZ \leftarrow CR<x, sel>
Syntax	mfcr32 rz, cr<x, sel>
attributes:	Privileged instructions
	Transfer the contents of the control register CR<x, sel> to the general register RZ.
Description:	The impact flag has no impact
Exception	Privilege Violation Exception

32-bit instruction format:

31	30	25	20	16	15	10	9	5	4	0
1	10000	sel	CRX		011000	00001		RZ		

Figure 14.135: MFCR

14.87 MOV—Data Transfer Instruction

Unified instructions	
Syntax	mov rz, rx
	RZ ÿ RX
Operations Compiled	results are always compiled into 16-bit instructions.
	mov16 rz, rx
	Copies the value in RX to the destination register RZ.
Description: The impact flag	has no impact
abnormal	none

16-bit instructions	
Operation	RZ ÿ RX
Syntax	mov16 rz, rx
Description	Copies the value in RX to the destination register RZ. Note that the instruction register index range is r0-r31.
Affects flag bit	No effect
Abnormal	

16 Mosquito command format:

15	14	10	9	6	5	2	1	0
1	11011		RZ		RX		11	

Figure 14.136: MOV-1

32-bit instructions	
Operation	RZ ÿ RX
Syntax	mov32 rz, rx
Description	Copies the value in RX to the destination register RZ. Note that this instruction is a pseudo-instruction of lsl32 rz, rx, 0x0.
Affects flag bit	No effect
Abnormal	

32-bit instruction format:

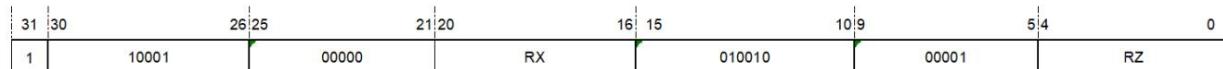


Figure 14.137: MOV-2

14.88 MOVF——C is 0 data transfer instruction

Unified	
instruction syntax	movf rz,
rx operation	if C==0, then RZ \leftarrow RX; else RZ \leftarrow RZ; Only
32-bit instructions exist in the compilation	
result . movf32 rz, rx means if C is	
	0, copy the value of RX to the destination register RZ; otherwise, the value of RZ remains unchanged. Note that this instruction is a pseudo-instruction of incf32 rz, rx, 0x0. No effect
Affects flag	
bit abnormality	

32-bit	
instruction operation	if C==0, then RZ \leftarrow RX; else RZ \leftarrow RZ;
Syntax	movf32 rz, rx
Description	If C is 0, copy the value of RX to the destination register RZ; otherwise, the value of RZ remains unchanged. Note that this instruction is a pseudo-instruction of incf32 rz, rx,
Affects flag	0x0. No effect
bit abnormality	

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RZ	RX	000011	00001	00000

Figure 14.138: MOVF

14.89 MOVI—Immediate Data Transfer Instruction

Unified	
instruction syntax:	movi rz, imm
operation:	RZ \leftarrow zero_extend(imm); Compilation result: Compile to the corresponding 16-bit or 32-bit instruction according to the range of the immediate value and register. if (imm16<256) and (z<7), then movi16 rz, imm8; else movi32 rz, imm16;
Description:	Zero-extend the 16-bit immediate value to 32 bits and then transfer it to the destination
RZ. Impact:	None Impact:
Limit	
the immediate	value to the range of 0x0-0xFFFF. Exception :
None	

16-bit	
	RZ \leftarrow zero_extend(imm8); movi16 rz, imm8
	Zero-extend the 8-bit
	immediate value to 32 bits and then transfer it to the destination RZ.
instruction operation syntax	
	The range of registers is r0-r7; the range of immediate values is 0-255.
description affected flags	no impact limit exception

16-bit instruction format:

15	14	11	10	8	7	0
0	0110	RZ		IMM8		

Figure 14.139: MOVI-1

32-bit instruction	
	RZ \leftarrow zero_extend(imm16); movi32 rz, imm16
	Zero-extend the 16-bit
	immediate value to 32 bits and then transfer it to the destination RZ.
operation syntax description	
Affects flags	No effect
Restrictions	The range of immediate values is
0x0-0xFFFF. Exceptions	none

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		10000		RZ		IMM16	

Figure 14.140: MOVI-2

14.90 MOVIH—Immediate high-order data transfer instruction

Unified instruction	
syntax	movih rz, imm16 operation RZ \leftarrow
	zero_extend(imm16) << 16 Compiled result only has 32-bit instructions.
Result	movih32 rz, imm16 Description Extend
	the 16-bit immediate value to 32 bits, then logically
	shift left 16 bits and transfer the result to the destination RZ.
	This instruction can be used with ori rz, rz, imm16 instructions to generate any 32-bit immediate
The flag bit	affects value. No effect
	the immediate value range to 0x0-0xFFFF.

32-bit instruction	
operation	RZ \leftarrow zero_extend(imm16) << 16 Syntax movih32 rz, imm16
Description	Zero-extend the 16-bit immediate
	value to 32 bits, then logically shift left 16 bits and transfer the result to the destination RZ.
	This instruction can be used with ori32 rz, rz, imm16 instructions to generate any 32-bit immediate
The flag bit	affects value. No effect
	the immediate value range to 0x0-0xFFFF.

32-bit instruction format:

31	30	26	25	21	20	16	15	0
1	11010		10001	RZ			IMM16	

Figure 14.141: MOVIH

14.91 MOVT——C is 1 data transfer instruction

Unified command	
syntax	movt rz, rx
operation	if C==1, then RZ \leftarrow RX; else RZ \leftarrow RZ; The
compiled result	contains only 32-bit instructions. movt32 rz, rx
Impact flag	No impact
Abnormal	No

32-bit	
instruction operation	if C==1, then RZ \leftarrow RX; else RZ \leftarrow RZ;
Syntax	movt32 rz, rx
Description	If C is 1, copy the value of RX to the destination register RZ; otherwise, the value of RZ remains unchanged. Note that this instruction is a pseudo-instruction of inct32 rz, rx,
Affects flag	0x0. No effect
bit abnormality	

32-bit instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10001		RZ		RX		000011		00010		00000	

Figure 14.142: MOVT

14.92 MTCR——Transfer Control Register Write Instruction

Unified instructions	
Syntax	mtc r x, cr<z, sel>
Operations	Transfer the contents of the general register to the control register CR<Z, sel> \leftarrow RX
The compiled result	contains only 32-bit instructions. mtc r32 rx, cr<z, sel>
Attribute: Privileged	instruction
	Transfer the contents of general register RX to control register CR<z, sel>.
Description Affects	flag bits If the target control register is not a PSR, this instruction does not affect the flag bits.
abnormal	Privilege Violation Exception

32-bit instructions	
operate	Transfer the contents of the general register to the control register CR<Z, sel> \leftarrow RX
Syntax	mtc r32 rx, cr<z, sel>
attributes: Privileged	instructions
	Transfer the contents of general register RX to control register CR<z, sel>.
Description Affects	flag bits If the target control register is not a PSR, this instruction does not affect the flag bits.
Exception	Privilege Violation Exception

32-bit instruction format:



Figure 14.143: MTCR

14.93 MULA.32.I - 32-bit signed multiply-accumulate instruction with lower 32 bits

Unified instructions	
Syntax	mula.32.I rz, rx, ry
	Rz[31:0] \leftarrow Rz[31:0] + {Rx[31:0] X Ry[31:0]}[31:0]
Operation Compiled results	contain only 32-bit instructions. mula.32.I rz, rx, ry

	Rx is multiplied by Ry to obtain a 64-bit result, which is added to Rz, and the lower 32 bits of the addition result are intercepted and stored in Rz.
Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

32-bit instructions	
	Rz[31:0] <- Rz[31:0] + {Rx[31:0] X Ry[31:0]}[31:0]
	mula.32.I rz, rx, ry
	Rx is multiplied by Ry to obtain a 64-bit result, which is added to Rz, and the lower 32 bits of the addition result are intercepted and stored in Rz.
Operation: Syntax: Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1	11110	RY	RX	000010	0 0 1 0 0	Rz

Figure 14.144: MULA.32.I

14.94 MULALL.S16.S – 16-bit signed low halfword multiply-accumulate with saturation

make

Unified instructions	
Syntax	mulall.s16.s rz, rx, ry
	Rz[31:0] = Saturate(Rz[31:0] + Rx[15:0] X Ry[15:0])
Operation Compiled	results contain only 32-bit instructions. mulall.s16.s rz, rx, ry

Illustrate :	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the multiplication result is added to Rz. The addition result is saturated and stored in Rz. The saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No
Affected flags:	effect
	none
	none
Restrictions: Exceptions:	

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[15:0] X Ry[15:0])
	mulall.s16.s rz, rx, ry
instruction operation: Source Description:	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the multiplication result is added to Rz. The addition result is saturated and stored in Rz. The saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No
Affected flags:	effect
	none
	none
Restrictions: Exceptions:	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	11110	RY	RX	1 0 0 0 0 0	0 1 1 0 1	Rz

Figure 14.145: MULALL.S16.S

14.95 MULA.(U/S)32——32-bit (no/yes) signed multiply-accumulate instruction

Unified instructions	
grammar	mula.u32 rz, rx, ry mula.s32 rz, rx, ry
Operation {Rz+1[31:0], Rz[31:0]} = {Rz+1[31:0], Rz[31:0]} + Rx[31:0] X Ry[31:0]	
The compiled result	contains only 32-bit instructions. mula.u32 rz, rx, ry mula.s32 rz, rx, ry

Illustrate:	Rx is multiplied by Ry to obtain a 64-bit result, which is then added to {Rz+1, Rz}. The high 32 bits of the addition result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

32-bit instructions	
	{Rz+1[31:0], Rz[31:0]} = {Rz+1[31:0], Rz[31:0]} + Rx[31:0] X Ry[31:0]
Operation: Syntax:	mula.u32 rz, rx, ry mula.s32 rz, rx, ry
Illustrate:	Rx is multiplied by Ry to obtain a 64-bit result, which is then added to {Rz+1, Rz}. The high 32 bits of the addition result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1	11110	RY	RX	1 0 0 0 0 0	S 0 1 0 0	Rz

Figure 14.146: MULA

14.96 MULT——Multiplication Instruction

System one		
	mult rz, rx	mult rz, rx, ry
Optimize instruction syntax	Multiply two numbers and put the lower 32 bits of the result into the general register RZ	Multiply two numbers and put the lower 32 bits of the result into a general register RZ \leftarrow RX \times RY;
Compilation results $y \leq 16$ and $(z < 16)$, then mult16 rz, rx; else mult32 rz, rz, rx;	$y \leq 16$ and $(z < 16)$, then mult16 rz, rx; else mult32 rz, rz, rx;	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if $(y == z)$ and $(x < 16)$ and $(z < 16)$, then mult16 rz, rx; else mult32 rz, rx, ry;

Description:	The lower 32 bits of the result of multiplying the contents of two source registers are stored in the destination register, and the upper 32 bits of the result are discarded. Whether the source operands are considered signed or unsigned, the result is
Affected flag:	the same. No effect
Abnormal:	None

16-bit	
instruction operation:	multiply two numbers and put the lower 32 bits of the result into a general register RZ \leftarrow RX \times RZ; Syntax:
mult16 rz, rx	Description: Multiply the contents of general registers RX and RZ and store the lower 32 bits of the result in general register RZ. The upper 32 bits are discarded. The result is the same whether the source operand is considered signed or unsigned.
Affected flags :	No effect
Restrictions:	The register range is r0-r15.
Exceptions:	None

Instruction format:

15	14	10	9	6	5	2	1	0
0	11111	RZ	RX	00				

Figure 14.147: MULT

32-bit	
instruction operation:	multiply two numbers and put the lower 32 bits of the result into a general register RZ $\ddot{\wedge}$ RX \times RY; Syntax:
mult32 rz, rx, ry	Description: Store the lower 32 bits of the result of multiplying the contents of general registers RX and RY in general register RZ, and the higher 32 bits of the result in Bits are discarded. The result is the same whether the source operand is considered signed or unsigned. No
Affected flag:	effect
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 0 0 0 1	RY	RX	1 0 0 0 0 1	0 0 0 0 1	Rz	

Figure 14.148: MULT2

14.97 MUL.(U/S)32——32-bit (no/yes) signed multiplication instruction

Unified command	
syntax	mul.u32 rz, rx, ry mul.s32 rz, rx, ry Operation
{Rz+1[31:0],Rz[31:0]}	= Rx[31:0] X Ry[31:0] The compiled result only has 32-bit instructions.
	mul.u32 rz, rx, ry mul.s32 rz, rx, ry

Description:	Multiply Rx and Ry to get a 64-bit result, of which the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz. Affected flags:
None Affected restrictions:	None
Exceptions:	None

32-bit instructions	
	{Rz+1[31:0],Rz[31:0]} = Rx[31:0] X Ry[31:0]
Operation: Syntax:	mul.u32 rz, rx, ry mul.s32 rz, rx, ry
	Rx is multiplied by Ry to obtain a 64-bit result, of which the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	RY	RX	1 0 0 0 0 0	S 0 0 0 0 0	Rz	

Figure 14.149: MUL

14.98 MVC—C-bit transfer instruction

Unified instructions	
Syntax	mvc
	RZ $\ddot{\wedge}$ C
Operation Compiled results	contain only 32-bit instructions. mvc32 rz;
	The condition bit C is transferred to the lowest bit of RZ, and the other bits of RZ are cleared.
Description: The impact flag	has no impact
Abnormal	

32-bit instructions	
	RZ $\ddot{\wedge}$ C
	.mvc32 rz
	The condition bit C is transferred to the lowest bit of RZ, and the other bits of RZ are cleared.
Operation Syntax	Description Impact Flags No Impact
Abnormal	

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10001	00000	00000	00000	000001	01000	RZ

Figure 14.150: MVC

14.99 MVCV——C-bit negation transfer

Unified instructions	
Syntax	mvcv
Operation RZ ý (!C)	<p>The compilation result is compiled into corresponding 16-bit or 32-bit instructions according to the range of registers.</p> <pre> if (z<16), then mvcv16 rz; else mvcv32 rz; </pre>
	The condition bit C is inverted and transferred to the lowest bit of RZ, and the other bits of RZ are cleared.
Description Impact Flags Impact	
Bit	
abnormal	none

16-bit instructions	
	RZ ý (!C)
	mvcv16 rz
	The condition bit C is inverted and transferred to the lowest bit of RZ, and the other bits of RZ are cleared.
Operation Syntax Description Impact Flags No Impact	
No	The register range is r0-r15.
restrictions	

16-bit instruction format:

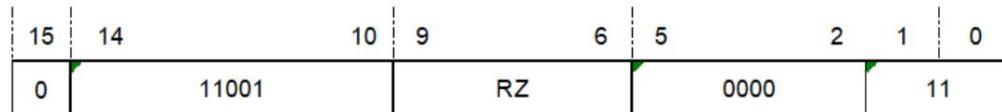


Figure 14.151: MVCV-1

32-bit instructions	
	RZ ý (!C)
	mvcv32 rz
	The condition bit C is inverted and transferred to the lowest bit of RZ, and the other bits of RZ are cleared.
Operation Syntax Description Impact Flags No Impact	
abnormal	none

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	00000	00000	000001	10000	RZ

Figure 14.152: MVCV-2

14.100 NIE—Interrupt Nesting Enable Instruction

The unified instruction	<p>syntax nie operation stores the interrupt control register context {EPSR, EPC} into the stack memory and then updates the stack pointer register to the stack Top of memory, open PSR.IE and PSR.EE; MEM[SP-4] ≠ EPC; MEM[SP-8] ≠ EPSR; SP ≠ SP-8; PSR({EE,IE}) ≠ 1; Only 16-bit instructions exist</p>
Compilation results	

Attribute:	Privileged
instruction	Description: Save the interrupt control register scene {EPSR, EPC} to the stack memory, then update the stack pointer register to the top of the stack pointer memory, turn on the interrupt and exception enable bits PSR.IE and PSR.EE. Use the stack pointer register direct addressing mode. No impact
Affected flags :	
Exceptions:	Access error exception, misalignment exception, privilege violation exception

16 Bit	
instruction operation:	Store the interrupt control register context {EPSR, EPC} into the stack memory, and then update the stack pointer register to the stack The top of the memory, open PSR.IE and PSR.EE; MEM[SP-4] ÿEPC; MEM[SP-8] ÿEPSR; SPÿSP-8; PSR({EE,IE}) ÿ 1; Syntax: nie16 Attribute: Privileged instruction
Description:	Save the interrupt control
	register scene {EPSR, EPC } to the stack memory, then update the stack pointer register to the top of the stack pointer memory, open the interrupt and exception enable bits PSR.IE and PSR.EE. Use the stack pointer register direct addressing mode. No effect
Affected flags :	
Exceptions:	Unaligned access exception, Unaligned access exception, Access error exception

Instruction format:

15	14	10	9	8	7	5	4	0
0	00101	0	0		011		00000	

Figure 14.153: NIE

14.101 NIR——Nested Interrupt Return Instruction

The unified instruction	
syntax nir operation	loads the interrupt control register context from the stack memory into {EPSR, EPC} and then updates the stack pointer register to the stack top of the stack memory; and interrupt return EPSR \leftarrow MEM[SP]; EPC \leftarrow MEM[SP+4]; SP \leftarrow SP+8; PSR \leftarrow EPSR; PC \leftarrow EPC; only
Compilation results	16-bit instructions exist

	Privileged instructions
Attributes: Description	Load the interrupted scene from the stack memory to {EPSR, EPC}, and then update the stack pointer register to the top of the stack memory; the PC value is restored to the value in the control register EPC, the PSR value is restored to the value of EPSR, and the instruction execution starts from the new PC address. Use the stack pointer register direct addressing
	mode. No impact
Affected flag:	Abnormal: Access error exception, misalignment exception, privilege violation exception

16	
Bit	
	<p>Load the interrupt control register context from the stack memory to {EPSR, EPC}, then update the stack pointer register to the top of the stack instruction operation: and interrupt return EPSR\leftarrowMEM[SP]; EPC\leftarrowMEM[SP+4]; SP\leftarrowSP+8; PSR\leftarrowEPSR; PC\leftarrowEPC;</p>
language Law:	nir16
	Privileged instructions
bright:	<p>Attribute: Say Load the interrupted scene from the stack memory to {EPSR, EPC}, and then update the stack pointer register to the top of the stack memory; the PC value is restored to the value in the control register EPC, the PSR value is restored to the value of EPSR, and the instruction execution starts from the new PC address. Use the stack pointer register direct addressing</p>
	mode. No impact
Affected flag: Abnormal:	Unaligned Access Exception, Unaligned Access Exception, Access Error Exception

Instruction format:

15	14	10	9	8	7	5	4	0
0	00101	0	0	011		00001		

Figure 14.154: NIR

14.102 NOR——Bitwise OR Instruction

Unified instruction		
	nor rz, rx	or rz, rx, ry
	RZ $\ddot{\vee}$!(RZ RX)	RZ $\ddot{\vee}$!(RX RY)
syntax operation not compilation results	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 16$) and ($z < 16$), then else nor32 rz, rz, rx; Perform	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if ($y == z$) and ($x < 16$) and ($z < 16$), then nor16 rz, rx else nor32 rz, rx, ry
	bitwise OR of the values of RX and RY/RZ, then perform bitwise NOT and store the result in RZ.	
	No impact	
	none	
Description:	Affects the flag bit abnormality	

16-bit	
	RZ $\ddot{\vee}$!(RZ RX) nor16 rz,
	rx Perform bitwise
	OR of the values of RZ and RX, then perform bitwise NOT and store the result in RZ.
instruction operation syntax	
	The register range is r0-r15.
description affected flags no impact limit exception	

16-bit instruction format:

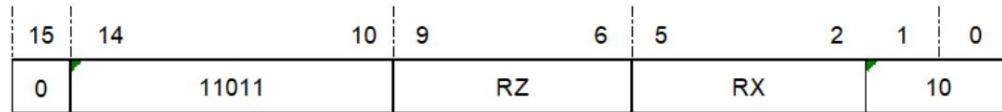


Figure 14.155: NOR-1

32-bit instructions	
	RZ \bar{y} !(RX RY)
	nor32 rz, rx, ry
	Perform a bitwise OR operation on the values of RX and RY, then perform a bitwise NOT operation on the values, and store the result in RZ.
Operation Syntax Description Impact Flags	No Impact
abnormal	none

32-bit instruction format:



Figure 14.156: NOR-2

14.103 NOT—Bitwise NOT Instruction

Unified instructions		
Syntax	not rz	
Operation	RZ \bar{y} !(RZ) Compile	
result	Compile to the corresponding 16-bit or 32-bit register range Bit instruction. if (z<16), then not16 rz; else not32 rz, rz;	Compile into corresponding 16-bit or 32-bit instructions according to the range of registers make. if (x==z) and (z<16), then not16 rz; else not32 rz, rx;
Indicates	that the value of RZ/RX is inverted bit by bit and the result is stored in RZ. Note that this instruction is a pseudo-instruction of nor rz, rz and nor rz, rx, rx.	
Influence	No impact	
Logo		
Bit		
Abnormal		

16-bit instructions	
Operation	RZ \bar{y} !(RZ)
Syntax	not16 rz
Description	Invert the value of RZ bit by bit and store the result in RZ. Note that this instruction is a pseudo-instruction of nor16 rz, rz.
Affects flag bit	No effect
Abnormal	

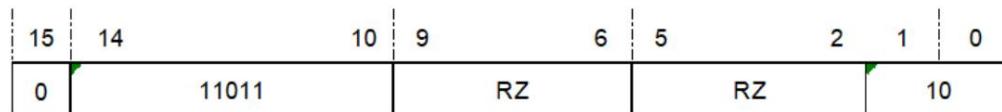
16-bit instruction format:

Figure 14.157: NOT-1

32-bit instruction	
	RZ $\ddot{\vee}$!(RX) not32
	rz, rx Invert the
operation syntax description	value of RX bit by bit and store the result in RZ. Note that this instruction is a pseudo-instruction of nor32 rz, rx, rx.
Impact flag	No impact
Abnormal	No

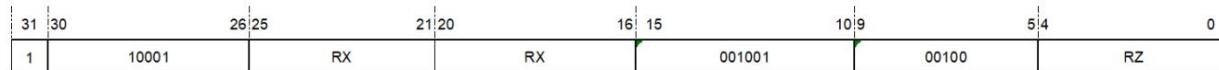
32-bit instruction format:

Figure 14.158: NOT-2

14.104 OR—Bitwise OR Instruction

Unified	
instruction syntax or	or rz, rx, ry
rz, rx operation	RZ $\ddot{\vee}$ RX RY is
compilation compatibility	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (z<16), then or16 rz, rx ; else or32 rz, rz, rx;
Description	Perform bitwise OR of the values of RX and RY/RZ, and store the result in RZ.
RZ. Impact	None
Impact	
Flag	Abnormal None

16-bit instructions	
	RZ $\ddot{\vee}$ RZ RX
	or16 rz, rx
	Perform a bitwise OR of the values in RZ and RX and store the result in RZ.
Operation Syntax Description Impact Flags No Impact	
The limit register range is r0-r15.	
abnormal	none

16-bit instruction format:

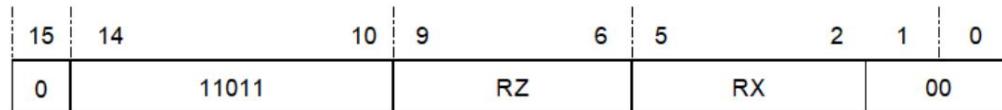


Figure 14.159: OR-1

32-bit instructions	
	RZ $\ddot{\vee}$ RX RY
	OR rZ, rX, rY
	Perform a bitwise OR operation on the values of RX and RY and store the result in RZ.
Operation Syntax Description Impact Flags No Impact	
Abnormal	

32-bit instruction format:

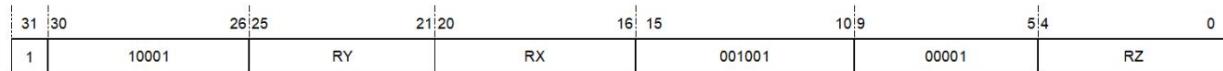


Figure 14.160: OR-2

14.105 ORI -- Bitwise OR Immediate Instruction

Unified instructions	
Syntax	ori rz, rx, imm16
	RZ $\ddot{\vee}$ RX zero_extend(IMM16)
Operation Compiled	results contain only 32-bit instructions. ori32 rz, rx, imm16
	Zero-extend the 16-bit immediate value to 32 bits, then bitwise-OR it with the value in RX and store the result in RZ.
Description: The impact flag	has no impact
Limit	The range of immediate value is 0x0-0xFFFF.
exception	none

32-bit instructions	
	RZ $\ddot{\vee}$ RX zero_extend(Imm16)
Operation affects flag bit	No effect
Limit	the range of immediate values to 0x0-0xFFFF.
Syntax	ori32 rz, rx, imm16
Description	Zero-extend the 16-bit immediate value to 32 bits, then bitwise-OR it with the value in RX and store the result in RZ.
Exception	none

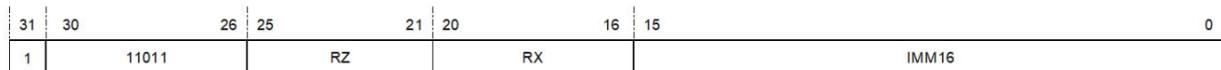
32-bit instruction format:

Figure 14.161: ORI

14.106 POP——Pop instruction

Unified command	
	pop reglist
syntax operation	Load multiple consecutive words from the stack memory into a consecutive register stack, then update the stack register to the top of the stack memory, and the routine returns; dst ý {reglist}; addr ý SP; foreach (reglist){ Rdst ý MEM[addr]; dst ý next {reglist}; addr ý addr + 4; } sp ý addr; PC ý R15 & 0xffffe; Compile to the
Compilation results	corresponding 16-bit or 32-bit instruction according to the register range if ({reglist} <16), then pop16 reglist; else pop32 reglist; Load
illustrate	multiple consecutive words from the stack memory into a continuous register stack, update the stack pointer register, and then implement the subroutine return function, that is, the program jumps to the location specified by the link register R15, and the lowest bit of the link register is ignored. Use stack register direct addressing mode. No effect
Affects flag bit exception	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

16-bit	
	instruction loads multiple consecutive words from the stack memory into a consecutive register file, then updates the stack register to the top of the stack memory, and the subroutine returns. <pre>dst ý {reglist}; addr ý SP; foreach (reglist) { Rdst ý MEM[addr]; dst ý next {reglist}; addr ý addr + 4; } sp ý addr; PC ý R15 & 0xffffe; pop16 reglist</pre>
bright	Grammar Load multiple consecutive words from the stack memory into a continuous register stack, update the stack pointer register, and then implement the subroutine return function, that is, the program jumps to the location specified by the link register R15, and the lowest bit of the link register is ignored. Use the stack pointer register direct addressing
	mode. No impact
	The register range is r4 – r11, r15.
	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception Affects flag bit restriction exception

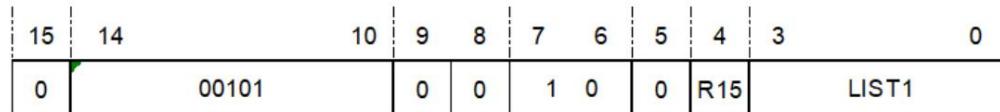
16-bit instruction format:

Figure 14.162: POP-1

LIST1 field:

Specifies whether registers r4-r11 are in the register list.

0000:

r4-r11 is not in the register list

0001:

r4 in the register list

0010:

r4-r5 in the register list

0011:

r4-r6 in the register list

...

1000:

r4-r11 in the register list

R15 Domain:

Specifies whether register r15 is in the register list.

0:

r15 is not in the register list

1:

r15 in the register list

32-bit	
instruction operation	loads multiple consecutive words from the stack memory into a consecutive register file dst {reglist}; addr ý SP; foreach (reglist){ Rdst ý MEM[addr]; dst ý next {reglist}; addr ý addr + 4; } sp ý addr; PC ý R15 & 0xffffe;
Syntax pop32 reglist Description	Load multiple consecutive words from the stack memory into a continuous register stack, update the stack pointer register, and then implement the subroutine return function, that is, the program jumps to the location specified by the link register R15, and the lowest bit of the link register is ignored. Use the stack pointer register direct addressing
The range of	mode. No effect
the affected flag limit	registers is r4 – r11, r15, r16 - r17, r29. Exceptions
Unaligned Access Exception, Access Error Exception, TLB Unrecoverable Exception, TLB Mismatch Exception, TLB Write Invalid Exception	

32-bit instruction format:**LIST1 field:**

31	30	28	25	21	20	16	15	12	11	10	9	8	7	6	5	4	3	0
1		11010		11110		00000		0000	0	0	0	R28		LIST2	R15		LIST1	

Figure 14.163: POP-2

Specifies whether registers r4-r11 are in the register list.

0000:

r4-r11 is not in the register list

0001:

r4 in the register list

0010:

r4-r5 in the register list

0011:

r4-r6 in the register list

...

1000:

r4-r11 in the register list

R15 Domain:

Specifies whether register r15 is in the register list.

0:

r15 is not in the register list

1:

r15 in the register list

LIST2 field:

Specifies whether registers r16-r17 are in the register list.

000:

r16-r19 is not in the register list

001:

r16 in the register list

010:

r16-r17 in the register list

R28 Domain:

Specifies whether register r28 is in the register list.

0:

r28 is not in the register list

1:

r28 in the register list

14.107 PSRCLR——PSR Bit Clear Instruction

The unified	
instruction syntax	is psrclr ee, ie, fe, af or the operands can also be any combination of ee, ie, fe, af.
The operation	clears one or more bits of the status register. PSR({EE, IE, FE, AF}) → 0 Only 32-bit
	instructions exist. psrclr32 ee, ie, fe, af
	Privileged instructions
Compilation result attributes:	Indicates that the selected PSR bit is cleared (1 means selected). The five-bit immediate value IMM5 is used to encode the control bit to be cleared. The corresponding relationship is as follows: Next: Immediate value IMM5 Each bit: Corresponding PSR control bit Imm5[0]: AF Imm5[1]: FE Imm5[2]: IE Imm5[3]: EE Imm5[4]: Reserved,
Affects flag bit	no effect
exception	privilege violation exception

32-bit	
instruction operation	clears one or more bits of the status register PSR({EE, IE, FE, AF}) \Rightarrow 0
Syntax	psrclr32 ee, ie, fe, af Or the operand can be any combination of ee, ie, fe, af. Privileged instruction
Attribute:	Indicates the selected PSR bit is cleared (1 means selected). The five-bit immediate value IMM5 is used to encode the control bit to be cleared. The corresponding relationship is as follows: Next: Immediate value IMM5 Each bit: Corresponding PSR control bit Imm5[0]: AF Imm5[1]: FE Imm5[2]: IE Imm5[3]: EE Imm5[4]: Reserved,
Affects flag bit	no effect
exception	privilege violation exception

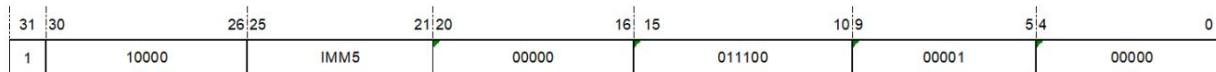
32-bit instruction format:

Figure 14.164: PSRCLR

14.108 PSRSET – PSR Set Bit Instruction

Unified	
instruction syntax	psrset ee, ie, fe, af Or the operand can also be any combination of ee, ie, fe, af. The
operation	sets some bits of the status register PSR({EE, IE, FE, AF}) ÿ 1 Only 32-bit
	instructions exist. psrset32 ee, ie, fe, af Privileged
	instruction
Compilation result attributes:	Indicates that the selected PSR bit is set (1 means selected). The five-bit immediate value IMM5 is used to encode the control bit to be cleared. The corresponding relationship is as follows: Next: Immediate value IMM5 Each bit: Corresponding PSR control bit Imm5[0]ÿAF Imm5[1]ÿFE Imm5[2]ÿIE Imm5[3]ÿEE Imm5[4]: Reserved,
Affects flag bit	no effect
exception	privilege violation exception

32-bit	
instruction	operation sets certain bits of the status register PSR ({EE, IE, FE, AF}) 1
Syntax	psrset32 ee, ie, fe, af Or the operand can also be any combination of ee, ie, fe, af. Privileged instruction
Attribute:	Indicates the selected PSR bit is set (1 means selected). The five-bit immediate value IMM5 is used to encode the control bit to be cleared. The corresponding relationship is as follows: Next: The PSR control bit corresponding to each bit of the immediate value IMM5 Imm5[0] AF Imm5[1] FE Imm5[2] IE Imm5[3] EE Imm5[4]: Reserved,
Affects	no effect
flag bit	
exception	privilege violation exception

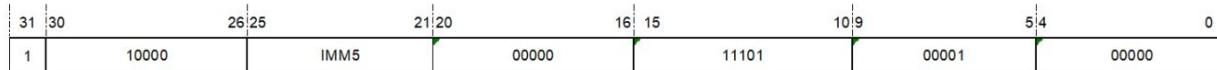
32-bit instruction format:

Figure 14.165: PSRSET

14.109 PUSH——Push instruction

Unified	
instruction syntax	push
reglist operation	stores the words in the register list into the stack memory, and then updates the stack register to the top of the stack memory; src ý {reglist}; addr ý SP; foreach (reglist) MEM[addr] ý Rsrc; src ý next {reglist}; addr ý addr - 4; } sp ý addr; Compile to the corresponding
Compilation	16-bit or 32-bit instruction according to the register range if ({reglist}<16), then push16 reglist; else push32 reglist;
Description	Store the words in the register list into the stack memory, and then update the stack register to the top of the stack memory. Register direct addressing mode.
Affects	No effect
flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

16-bit	
instruction operation	Store the word in the register list into the stack memory src \ddagger {reglist}; addr \ddagger SP; foreach (reglist) \ddagger MEM[addr] \ddagger Rsrc; src \ddagger next {reglist}; addr \ddagger addr - 4; } sp \ddagger addr
Syntax	push16 reglist
Description	Store the word in the register list into the stack memory, and then update the stack register to the top of the stack memory. Use stack Stack register direct addressing
The range of	mode. No effect
the affected flag limit	registers is r4 – r11, r15.
Exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

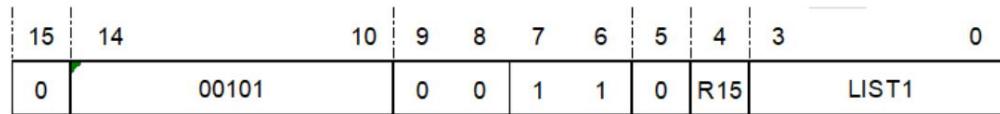
16-bit instruction format:

Figure 14.166: PUSH-1

LIST1 field:

Specifies whether registers r4-r11 are in the register list.

0000:

r4-r11 is not in the register list

0001:

r4 in the register list

0010:

r4-r5 in the register list

0011:

r4-r6 in the register list

...

1000:

r4-r11 in the register list

R15 Domain:

Specifies whether register r15 is in the register list.

1:

r15 in the register list

32-bit	
instruction operation	loads multiple consecutive words from the stack memory into a consecutive register file src {reglist}; addr < SP; foreach (reglist) { MEM[addr] < Rsrc; src < next {reglist}; addr < addr - 4; } sp < addr
	Syntax push32
reglist Description	Store the words in the register list into the stack memory, and then update the stack register to the top of the stack memory. Using the stack Register direct addressing mode.
The range of	No effect
the affected flag limit	registers is r4 – r11, r15, r16 - r17, r29. Exceptions
Unaligned	Access Exception, Access Error Exception, TLB Unrecoverable Exception, TLB Mismatch Exception, TLB Write Invalid Exception

32-bit instruction format:



Figure 14.167: PUSH-2

LIST1 field:

Specifies whether registers r4-r11 are in the register list.

0000:

r4-r11 is not in the register list

0001:

r4 in the register list

0010:

r4-r5 in the register list

0011:

r4-r6 in the register list

...

1000:

r4-r11 in the register list

R15 Domain:

Specifies whether register r15 is in the register list.

0:

r15 is not in the register list

1:

r15 in the register list

LIST2 field:

Specifies whether registers r16-r17 are in the register list.

000:

r16-r19 is not in the register list

001:

r16 in the register list

010:

r16-r17 in the register list

R29 domain:

Specifies whether register r29 is in the register list.

0:

r29 is not in the register list

1:

r29 in the register list

14.110 REVB——Byte Reverse Instruction

Unified	
instruction syntax	revb rz, rx
operation	RZ[31:24] \downarrow RX[7:0]; RZ[23:16] \downarrow RX[15:8]; RZ[15:8] \downarrow RX[23:16]; RZ[7:0] \downarrow RX[31:24];
The compilation result	The compilation result is compiled into the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 16$) and ($z < 16$), then revb16 rz, rx; else revb32 rz, rx;
Description:	Reverse the byte order of the RX value, keep the bit order of each byte unchanged, and store the result in RZ .
	No impact

16-bit instruction	
operation	RZ[31:24] \downarrow RX[7:0]; RZ[23:16] \downarrow RX[15:8]; RZ[15:8] \downarrow RX[23:16]; RZ[7:0] \downarrow RX[31:24]; revb16 rz, rx Reverse the byte order of the
Syntax	RX value, keep the
	bit order of each byte unchanged, and store the result in RZ. No effect
Description	
Impact	The register range is r0-r15.
Flag Limit Exception	

16-bit instruction format:

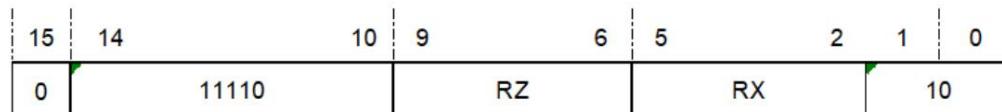


Figure 14.168: REVB-1

32-bit instruction	
operation	RZ[31:24] $\ddot{\vee}$ RX[7:0]; RZ[23:16] $\ddot{\vee}$ RX[15:8]; RZ[15:8] $\ddot{\vee}$ RX[23:16]; RZ[7:0] $\ddot{\vee}$ RX[31:24]; revb32 rz, rx Reverse the byte order of the RX
Syntax	value, keep the bit
	order of each byte unchanged, and store the result in RZ. No effect
description	
affects flag exception	none

32-bit instruction format:

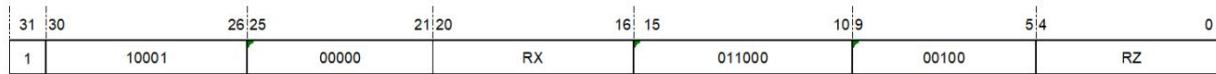


Figure 14.169: REVB-2

14.111 REVH—Reverse Halfword Instruction

Unified	
instruction syntax	revh rz, rx
operation	RZ[31:24] $\ddot{\vee}$ RX[23:16]; RZ[23:16] $\ddot{\vee}$ RX[31:24]; RZ[15:8] $\ddot{\vee}$ RX[7:0]; RZ[7:0] $\ddot{\vee}$ RX[15:8]; Compile to the corresponding 16-bit or 32-
Compilation results	bit instruction according to the register range. if (x<16) and (z<16), then revh16 rz, rx; else revh32 rz, rx;
Description:	Reverse the byte order of the value of RX in the half-word, that is, swap the two bytes in the upper half-word and the two bytes in the lower half-word respectively, and keep the order between the two half-words and the bit order in each byte unchanged, and store the result
Affects flag	in RZ. No effect
bit abnormality	

16-bit	
	instruction operation RZ[31:24] $\ddot{\vee}$ RX[23:16]; RZ[23:16] $\ddot{\vee}$ RX[31:24]; RZ[15:8] $\ddot{\vee}$ RX[7:0]; RZ[7:0] $\ddot{\vee}$ RX[15:8];
Syntax revh16 rz, rx	Description Reverse the byte order of the value of RX in the halfword, that is, swap the two bytes in the upper halfword and the two bytes in the lower halfword respectively, and keep the order between the two halfwords and the bit order in each byte unchanged, and store the result in RZ. No effect
Affected flag	result of the affected flag limit register is r0-
r15 .	

16-bit instruction format:

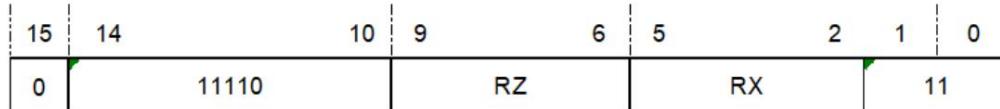


Figure 14.170: REVH-1

32-bit	
	instruction operation RZ[31:24] $\ddot{\vee}$ RX[23:16]; RZ[23:16] $\ddot{\vee}$ RX[31:24]; RZ[15:8] $\ddot{\vee}$ RX[7:0]; RZ[7:0] $\ddot{\vee}$ RX[15:8];
Syntax revh32 rz, rx	Description Reverse the byte order of the value of RX in the halfword, that is, swap the two bytes in the upper halfword and the two bytes in the lower halfword respectively, and keep the order between the two halfwords and the bit order in each byte unchanged, and store the result in RZ. No effect
Affects flag	bit abnormality

32-bit instruction format:

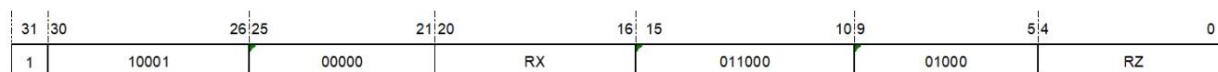


Figure 14.171: REVH-2

14.112 ROTL——Rotate Left Instruction

Unified instruction		
	rotl rz, rx	rotl rz, rx, ry
	RZ ѕ RZ <<< RX[5:0]	RZ ѕ RX <<< RY[5:0]
syntax operation	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (z<16), then rotl16 else rotl32 rz, rz, rx; rotl32 rz, rx, ry For rotl rz, rx, the value of RZ is circularly shifted left (the original value is	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (x==z) and (y<16) and (z<16), then rotl16 rz, ry else
illustrate	shifted left, and the right side is shifted into the bit shifted out from the left side), and the result is stored in RZ. The number of left shifts is determined by the value of the lower 6 bits of RX (RX[5:0]); if the value of RX[5:0] is greater than 31, RZ will be cleared. For rotl rz, rx, ry, the value of RX is circularly shifted left (the original value is shifted left, and the right side is shifted into the bit shifted out from the left side), and the result is stored in RZ. The number of left shifts is determined by the value of the lower 6 bits of RY (RY[5:0]); if the value of RY[5:0] is greater than 31, RZ will be cleared. No effect	
	none	
Affects flag bit exception		

16-bit instruction	
operation	RZ ѕ RZ <<< RX[5:0] Syntax rotl16 rz, rx
Description	Circularly shift the value
of RZ to the left (the original value is shifted left, and the right side is shifted into the bit shifted out from the left side), and the result is stored in RZ. The number of left shifts is determined by RX	The value of the lower 6 bits (RX[5:0]) determines the value; if the value of RX[5:0] is greater than 31, then RZ will be cleared.
The range of the	No effect
affected flag limit	register is r0- r15 .

16-bit instruction format:

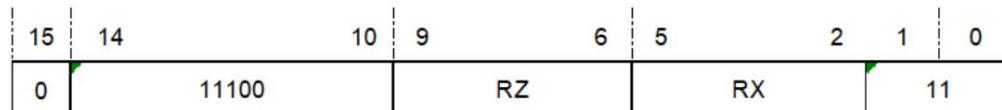


Figure 14.172: ROTL-1

32-bit instruction	
operation	RZ ← RX <<< RY[5:0]
Description	Circularly shift the value of RX
to the left (the	original value is shifted to the left, and the right side is shifted into the bit shifted out from the left side), and the result is stored in RZ. The number of left shifts is determined by RY
	The value of the lower 6 bits (RY[5:0]) determines the value; if the value of RY[5:0] is greater than 31, RZ will be cleared.
Affects	No effect
flag bit	
abnormality	

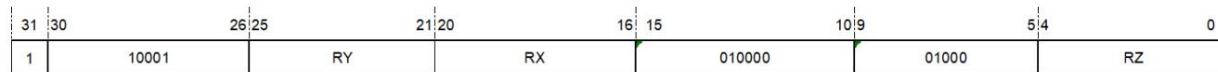
32-bit instruction format:

Figure 14.173: ROTL-2

14.113 ROTLI—Rotate Left Immediate Instruction

Unified instruction	
syntax	rotli r _z , r _x , imm5 operation RZ ←
RX <<< IMM5	Compilation result Description
Circularly	rotli32 r _z , r _x , imm5;
shift	
the value of	RX to the left (the original value is shifted to the left, and the right side is shifted into the bit shifted out on the left side), and the result is stored in RZ. The
	number of left shifts is determined by the value of the 5-bit immediate value (IMM5); if the value of IMM5 is equal to 0, then the value of RZ
The flag	will be the same as RX. No effect
bit	
affects	the immediate value range to 0-31.

32-bit instructions	
Operation	RZ \leftarrow RX <<< IMM5
Syntax	rotli32 rz, rx, imm5
Description	Shift the value of RX to the left in a circular manner (shift the original value to the left, and shift the right side into the bit shifted out from the left side), and store the result in RZ. The number of left shifts is increased by 5. The value of the immediate bit (IMM5) determines the value of RZ; if the value of IMM5 is equal to 0, the value of RZ will be the same as RX.
Influence	No impact
Logo	
Bit	
Limits	the immediate value to the range 0-31.
Abnormal	

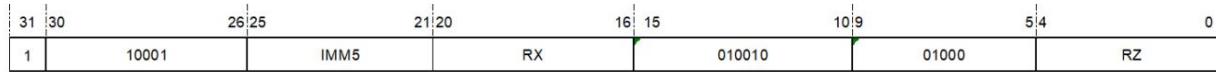
32-bit instruction format:

Figure 14.174: ROTLI

14.114 RSUB -- Reverse Subtraction Instruction

Unified instructions	
Syntax	rsub rz, rx, ry
	RZ \leftarrow RY - RX
Operation Compiled results	contain only 32-bit instructions. rsub32 rz, rx, ry
Illustrate	Subtract the RX value from the RY value and store the result in RZ. Note that this instruction is a pseudo-instruction of subu rz, ry, rx.
Affects flag bit	No effect
Abnormal	

32-bit instructions	
Operation	RZ \leftarrow RY - RX
Syntax	rsub32 rz, rx, ry
Description	Subtract the RX value from the RY value and store the result in RZ. Note that this instruction is a pseudo-instruction of subu32 rz, ry, rx.
Impact Signs	No impact
Bit	
abnormal	none

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10001 RX RY 000000 00100 RZ						

Figure 14.175: RSUB

14.115 RTS—Subroutine return instruction

Unified	
instruction	
syntax	rts operation program jumps to the location specified by the link register PC \ddagger R15 & 0xffffe
	Always compile to 16-bit instructions. rts16
Compilation result shows	that the program jumps to the location specified by link register R15, and the lowest bit of the link register is ignored. The jump range of RTS16 instruction is the entire 4GB address space. This instruction is used to implement the subroutine return function. Note that this instruction is a
Affects	pseudo instruction of jmp r15. No effect
flag bit	
abnormality	

16-bit	
instruction	to jump to the location specified by the link register PC \ddagger R15 & 0xffffe
Syntax	rts16
Description	The program jumps to the location specified by the link register R15. The lowest bit of the link register is ignored. The jump range of the RTS16 instruction is the entire 4GB address space. This instruction is used to implement the subroutine return function. Note that this instruction is a pseudo-
Affects	instruction of jmp16 r15. No effect
flag bit	
abnormality	

16-bit instruction format:

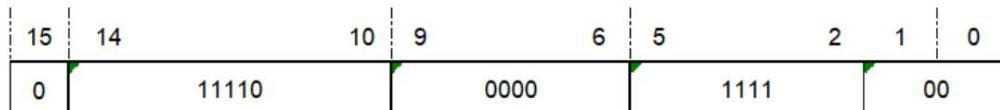


Figure 14.176: RTS-1

32-bit	
instruction	to jump the program to the location specified by the link register PC \leftarrow R15 & 0xffffe
The syntax	rts32
indicates	that the program jumps to the location specified by the link register R15. The lowest bit of the link register is ignored. The jump range of the RTS instruction is The entire 4GB address space. This instruction is used to implement the subroutine return function. Note that this instruction is a pseudo-instruction of jmp32
Affects	r15. No effect
flag bit	
abnormality	

32-bit instruction format:

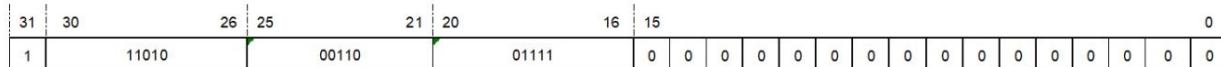


Figure 14.177: RTS-2

14.116 RTE - Return from exception and normal interrupt

Unified	
instruction syntax	
RTE operation	exception and normal interrupt return PC \leftarrow EPC, PSR \leftarrow EPSR
The compiled result	contains only 32-bit instructions. rte32
Attribute:	Privileged instruction
description	The PC value is restored to the value stored in the control register EPC, the PSR value is restored to the value stored in EPSR, and the instruction is executed Start from the new PC address. No
Affects flag	effect
bit	
exception	privilege violation exception

32-bit instruction	
operation	exceptions and normal interrupt returns PC → EPC, PSR → EPSR Syntax rte32
Attributes:	Privileged
Instruction Description	The PC
value is restored	to the value stored in the control register EPC, and the PSR value is restored to the value stored in EPSR. The instruction is executed
	Start from the new PC address. No
Affects flag bit	effect
exception	privilege violation exception

32-bit instruction format:

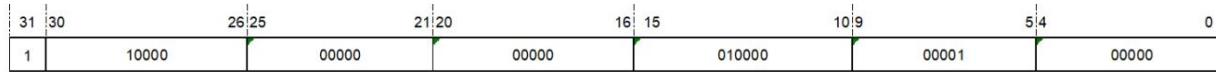


Figure 14.178: RTE

14.117 SCE - Conditional Execution Set Instruction

The unified	
	sce cond
	Set the conditional execution bits for the next 4 instructions
instruction syntax	operation compilation result only has 32-bit instructions
	sce32 cond

illustrate	The sce instruction is used to set the conditional execution bit of the following 4 instructions. The operand COND is a 4-bit binary immediate value. The lowest bit represents the conditional bit of the first instruction after the sce instruction, and the second lowest bit represents the conditional bit of the second instruction after the sce instruction... and so on. The conditional bit is 1, which means that C is 1 and the execution is normal. The conditional bit is 0, which means that C is 0 and the execution is normal. If the C bit does not meet the conditional bit, the conditional execution instruction has no effect. The value of the C bit used for judgment is based on the value when the sce instruction is executed.
Affected	If the following 4 instructions generate an exception or interrupt, the conditional execution bit will be saved in EPSR or FPSR.
Affected	flags:
limit system:	The instructions following the sce instruction can only be arithmetic operation instructions, multiplication and division instructions, byte, half-word, word load and store instructions in the immediate addressing mode, and these instructions cannot affect the condition bit C. The operand is a 4-bit binary immediate number.
	none
Exception: Notes	<p>For example, the command sequence is:</p> <pre>100101 mov r1, r0 mov r3, r2 mov r5, r4 mov r7, r6 The conditional bits of the sce</pre> <p>instruction are 0101. If the C bit is 0 when the sce instruction is executed, the second and fourth mov instructions meet the execution conditions and write the results to registers 3 and 7; the first and third mov instructions do not meet the execution conditions and do not write the results to the destination register.</p>

32 Bit	
	Set the conditional execution bit of the next 4 instructions set_condition_execution(COND);
	sce32 cond
	The sce instruction is used to set the conditional execution bit of the following 4 instructions. The operand COND is a 4-bit binary immediate value. The lowest bit represents the instruction operation symbol; the second lowest bit represents the first instruction after the sce instruction, and the second lowest bit represents the conditional bit of the second instruction after the sce instruction... and so on. The conditional bit is 1, which means that C is 1 and the execution is normal. The conditional bit is 0, which means that C is 0 and the execution is normal. If the C bit does not meet the conditional bit, the conditional execution instruction has no effect. The value of the C bit used for judgment is based on the value when the sce instruction is executed. If the following 4 instructions generate an exception or interrupt, the conditional execution bit will be saved in EPSR or FPSR.
Impact Mark	
Zhi	
Bit:	
Limit:	The instructions following the sce instruction can only be arithmetic operation instructions, multiplication and division instructions, byte, half-word, word load and store instructions in the immediate addressing mode, and these instructions cannot affect the condition bit C. The operand is a 4-bit binary immediate
	number. None
Exception Notes:	For example, the command sequence is: sce32,0101 mov32 r1, r0 mov32 r3, r2 mov32 r5, r4 mov32 r7, r6 The conditional bits of the sce instruction are 0101. If the C bit is 0 when the sce instruction is executed, the second and fourth mov instructions meet the execution conditions and write the results to registers 3 and 7; the first and third mov instructions do not meet the execution conditions and do not write the results to the destination register.

Instruction format:

31	30	26	25	24	21	20	16	15	10	9	5	4	0
1	10000	0	COND		00000		000110		00001		00000		

Figure 14.179: SCE

14.118 SEXT -- BIT EXTRACT AND SIGN EXTEND INSTRUCTION

Unified instruction	
language	sext rz, rx, msb, lsb method
	RZ = sign_extend(RX[MSB:LSB])
	Only 32-bit instructions exist. sext32 rz, rx, msb, lsb
operation	Extract a continuous bit range (RX[MSB:LSB]) of RX specified by two 5-bit immediate values (MSB, LSB), sign-extend to 32 bits, and store the result is RZ if MSB is equal to 31 and LSB is equal to 0, the value of RZ is the same as RX. If MSB is equal to LSB, the value of RZ is the result of one-bit sign extension of RX[MSB] (ie, RX[LSB]). If MSB is less than LSB, the behavior of this instruction is unpredictable. No effect
	The range of MSB is 0-31, the range of LSB is 0-31, and MSB should be greater than or equal to LSB.
	none
Affects flag bit restriction exception	

32-bit instruction	
	RZ ȳ sign_extend(RX[MSB:LSB])
	sext32 rz, rx, msb, lsb
operation	Extract a continuous bit range (RX[MSB:LSB]) of RX specified by two 5-bit immediate values (MSB, LSB), sign-extend to 32 bits, and store the result in RZ. If MSB is equal to 31 and LSB is equal to 0, the value of RZ is the same as RX. If MSB is equal to LSB, the value of RZ is the result of one-bit sign extension of RX[MSB] (ie, RX[LSB]). If MSB is less than LSB, the behavior of this instruction is unpredictable. No effect
	The range of MSB is 0-31, the range of LSB is 0-31, and MSB should be greater than or equal to LSB.
	none
Affects flag bit restriction exception	

32-bit instruction format:

31	30	25	20	15	9	4	0
1	10001	LSB	RX	010110	MSB	RZ	

Figure 14.180: SEXT

MSB field

Specifies the bit to be extracted starting from.

LSB field

Specifies the end bit to be extracted.

00000

0

00000

0 position

00001

1

00001

1 bit

.....
11111

31

11111

31 bits

14.119 SEXTB -- Extract Byte and Sign-Extend Instruction

Unified	
instruction syntax	sextb rz, rx
operation	RZ := sign_extend(RX[7:0]); The compilation
result	is compiled into the corresponding 16-bit or 32-bit instruction according to the register range. if (z<16) and (x<16), then sextb16 rz, rx; else sextb32 rz, rx;
Description:	The low byte of RX (RX[7:0]) is sign-extended to 32 bits, and the result is stored in RZ. Affected
flags	None Affected bits
Exception	None

16-bit	
instruction operation	RZ := sign_extend(RX[7:0]); Syntax
sextb16 rz, rx	The low byte of RX
flag	(RX[7:0]) is sign-extended to 32 bits, and the result is stored in RZ. Description Affects the
bit	No impact Restriction
The register range	is r0-r15. Exception
	none

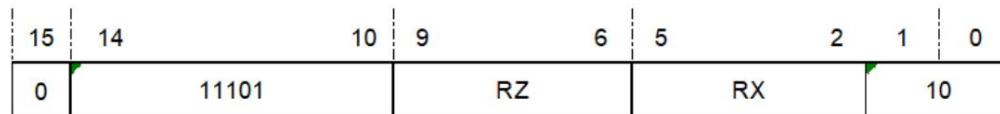
16-bit instruction format:

Figure 14.181: SEXTB-1

32-bit	
instruction operation	RZ \leftarrow sign_extend(RX[7:0]); Syntax
Description	Sign-extend
the low byte of RX (RX[7:0]) to 32 bits, and store the result in RZ.	Note that this instruction is a pseudo instruction of sext32 rz, rx, 0x7, 0x0.
Affects flag bit	No effect
abnormality	

32-bit instruction format:

Figure 14.182: SEXTB-2

14.120 SEXTH -- Extract Halfword and Sign-Extend Instruction

Unified	
instruction syntax	sexth rz, rx
operation	RZ \leftarrow sign_extend(RX[15:0]); The compilation result
is compiled into	the corresponding 16-bit or 32-bit instruction according to the register range. if (z<16) and (x<16), then sexth16 rz, rx; else sexth32 rz, rx;
Description:	Sign-extend the lower halfword of RX (RX[15:0]) to 32 bits, and store the result in RZ. Affected flags None
Affected bits Exceptions	
None	

16-bit	
	RZ \leftarrow sign_extend(RX[15:0]); sexth16 rz, rx
	Sign-extend the
	lower halfword of RX (RX[15:0]) to 32 bits and store the result in RZ.
instruction operation syntax	
	The register range is r0-r15.
description affected flags	no impact limit exception

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11101		RZ		RX		11	

Figure 14.183: SEXTH-1

32-bit	
instruction operation	RZ ý sign_extend(RX[15:0]); Syntax
sexth32 rz, rx	Description Sign-extend the lower halfword of RX (RX[15:0]) to 32 bits and store the result in RZ. Note that this instruction is a pseudo instruction of sext32 rz, rx, 0x15, 0x0.
Affects flag	No effect
bit	
abnormality	

32-bit instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10001		00000		RX		010110		01111		RZ	

Figure 14.184: SEXTH-2

14.121 SRS.B -- Byte Signed Store Instruction

Unified instruction	
syntax	srs.b rz, [label] method
	Store the least significant byte of register to memory MEM[R28 + zero_extend(offset)] ÿ RZ[7:0] Only 32-bit instructions exist. srs32.b rz,
	[label]
operation	Store the lowest byte sign in register RZ to the location of label. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX to the 18-bit relative offset unsigned extended to 32 bits. The SRS.B instruction can address +256KB of address space. Note that the offset DISP is the binary operand Ofset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Affects	flag bit exception

32-bit instruction	
operation	Store the lowest byte sign in the register to the memory MEM [R28 + zero_extend(offset)] ÿ RZ[7:0] Syntax srs32.b rz, [label]
	Store the lowest byte sign in register RZ to the location of label. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX to the value of the 18-bit relative offset unsigned extended to 32 bits. The SRS.B instruction can address +256KB address space. Note that the offset DISP is the binary operand Ofset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Affects	flag bit exception

32-bit instruction format:



Figure 14.185: SRS.B

14.122 SRS.H -- Store Half-Word Instruction

Unified instruction	
	srs.h rz, [label]
	Store the lowest half word number from register to memory MEM[R28 + zero_extend(offset << 1)] ÿ RZ[7:0] Only 32-bit instructions exist. srs32.h rz, [label]
	Store the lowest half-word in register RZ to the location of label. Use register plus unsigned immediate offset addressing. The effective address of the memory is <small>syntax operand1:register, operand2:register, operand3:register</small> register RX to the 18-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The SRS.H instruction can address +512KB address space. Note: The offset DISP is the binary operand Offset. No effect
Affects	flag bit exception Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction	
	to store the lowest half word in a register to memory Operation MEM[R28 + zero_extend(offset << 1)] & RZ[7:0] Syntax Description srs32.h rz, [label]
	Store the lowest half word in register RZ to the location of label. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX to the 18-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The SRS.H instruction can address +512KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Affects	flag bit exception

32-bit instruction format:

Figure 14.186: SRS.H

14.123 SRS.W——Word Store Instruction

Unified instruction	
syntax	srs.w rz, [label] operation
	Store the lowest word in register to memory MEM[R28 + zero_extend(offset << 2)] \ddagger RZ[7:0] Only 32-bit instructions exist. srs32.w rz, [label]
compilation results description	Store the lowest character number in register RZ to the location of label. Use register plus unsigned immediate offset addressing. The effective address of the memory obtained by adding the base register RX to the 18-bit relative offset shifted left by 2 bits and unsigned extended to 32 bits. The SRS.W instruction can address $+1024KB$ address space. Note that the offset DISP is the binary operand Offset. No effect
Affects	flag bit exception Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction	
operation	Store the lowest word in the register to memory MEM [R28 + zero_extend(offset << 2)] \ddagger RZ[7:0] Syntax Description srs32.w rz, [label]
	Store the lowest character number in register RZ to the location of label. Use register plus unsigned immediate offset addressing. The effective address of the memory obtained by adding the base register RX to the 18-bit relative offset shifted left by 2 bits and unsigned extended to 32 bits. The SRS.W instruction can address $+1024KB$ address space. Note that the offset DISP is the binary operand Offset. No effect
Affects	flag bit exception Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

32-bit instruction format:

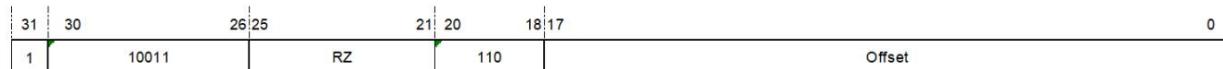


Figure 14.187: SRS.W

14.124 ST.B - Byte Store Instructions

Unified instruction	
syntax	st.b rz, (rx, disp) method
	Store the lowest byte of the register to memory $\text{MEM}[\text{RX} + \text{zero_extend}(\text{offset})] \leftarrow \text{RZ}[7:0]$ Compile to the corresponding 16-bit or 32-bit instruction according to the offset and register range. if ($\text{disp} < 32$) and ($\text{x} < 7$) and ($\text{z} < 7$), then st16.b rz, (rx, disp) ;
operation	compilation result st32.b rz, (rx, disp); Store the lowest
explain	byte in register RZ to memory. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX to the value of the 12-bit relative offset unsignedly extended to 32 bits. The ST.B instruction can address +4KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception
Affects flag	bit exception

16-bit	
	instruction to store the lowest byte of a register to memory
Operation	MEM[RX + zero_extend(offset)] < RZ[7:0] Syntax st16.b rz, (rx, disp)
	Store the lowest byte in register RZ to memory. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX plus the 5-bit relative offset unsignedly extended to 32 bits. The ST16.B instruction can address +32B space. Note that the offset DISP is the binary operand Offset. No effect
	The register range is r0-r7.
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception Affects flag bit restriction exception

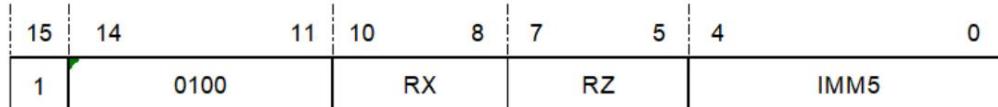
16-bit instruction format:

Figure 14.188: ST.B-1

32-bit instruction	
MEM[RX + zero_extend(offset)]	to store the lowest byte of a register to memory Operation Syntax st32.b rz, (rx, disp) Description
	Store the lowest byte in register RZ to memory. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX to the value of the 12-bit relative offset unsignedly extended to 32 bits. The ST32.B instruction can address +4KB address space. Note that the offset DISP is the binary operand Offset. No effect
	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception Affects flag bit exception

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	10111		RZ		RX	0	0	0	0	Offset

Figure 14.189: ST.B-2

14.125 ST.H——Halfword Store Instruction

Unified instruction	
syntax	st16.h rz, (rx, disp) operation
	stores the lowest byte in a register into memory MEM[RX + zero_extend(offset<< 1)] & RZ[15:0] is compiled into the corresponding
Compilation result	16-bit or 32-bit instruction according to the offset and register range. if (disp<64)and(x<7)and(z<7), st16.h rz, (rx, disp) ; else st32.h rz, (rx, disp);
Description	Store the lower halfword in register RZ into the memory. The addressing mode is register plus unsigned immediate offset. The effective address of the memory is obtained by adding the base register RX plus the 12-bit relative offset shifted left by 1 bit and unsignedly extended to 32 bits. The ST.H instruction can address +8KB address space. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

16-bit	
	Store the lower halfword of register to memory MEM[RX + zero_extend(offset << 1)] & RZ[15:0] st16.h rz, (rx, disp)
instruction	The lower halfword in register RZ is stored in memory. The addressing mode is register plus unsigned immediate offset. The effective address of the memory is obtained by adding the base register RX plus the 5-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits. The ST16.H instruction can address the +64B space. Note that the offset DISP is obtained by shifting the binary operand Ofset left by 1 bit. No effect
	The register range is r0-r7.
Affects flag bit restriction exception	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception

16-bit instruction format:

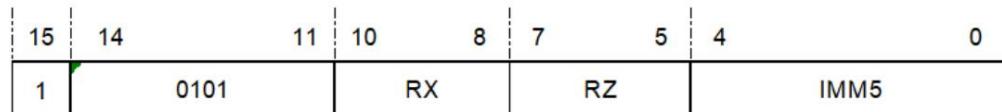


Figure 14.190: ST.H-1

32-bit instruction	
	Store the lower halfword of register to memory MEM[RX + zero_extend(offset << 1)] ; RZ[15:0] st32.h rz, (rx, disp)
bright	<p>operation Store lower halfword in register RZ is stored in memory. The addressing mode is register plus unsigned immediate offset. The effective address of the memory is obtained by adding the base register RX plus the 12-bit relative offset shifted left by 1 bit and unsigned extended to 32 bits.</p> <p>The ST32.H instruction can address +8KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by 1 bit. No effect</p>
	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception
Affects	flag bit exception

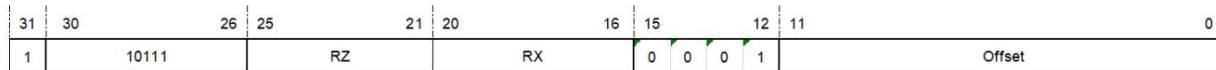
32-bit instruction format:

Figure 14.191: ST.H-2

14.126 ST.W——Word Store Instruction

Unified	
instruction syntax	st.w rz, (rx, disp)
operation	<p>stores the word in register to memory</p> <p>MEM[RX + zero_extend(offset<< 2)] \leftarrow RZ[31:0] is compiled into the</p>
Compilation result	<p>corresponding 16-bit or 32-bit instruction according to the offset and register range. if (x=sp)</p> <pre>(z<7) and (disp < 1024), st16.w rz, (sp, disp) ; else if (disp<128) and (x<7) and (z<7), st16.w rz, (rx, disp) ; else st32.w rz, (rx, disp);</pre>
Description	<p>Store the word in register RZ into the memory. The addressing mode is register plus unsigned immediate offset. The effective address of the memory is obtained by adding the base register RX to the 12-bit relative offset shifted left by two bits and unsigned extended to 32 bits.</p> <p>The ST.W instruction can address +16KB address space.</p>
Affects	No effect
flag	
bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

16-bit	
	instruction to store word in register to memory
Operation	MEM[RX + zero_extend(offset << 2)] & RZ[31:0] Syntax st16.w rz, (rx, disp)
Method	st16.w rz, (sp, disp) Store word in register RZ to memory. Register plus
the	unsigned immediate offset addressing mode is adopted. When rx=sp , the effective address of memory is obtained by adding the base register RX to 8-bit relative offset shifted left by two bits and unsignedly extended to 32 bits. When rx is another register, the effective address of memory is obtained by adding the base register RX to the 5-bit relative offset shifted left by two bits and unsignedly extended to 32 bits. The ST16.W instruction can address +1KB space. Note that the offset DISP is obtained by shifting the binary operand IMM5 left by two bits. When the base register RX is SP, the offset DISP is obtained by shifting the binary operand {IMM3, IMM5} left by two bits. No impact
Impact flag	
limit	The register range is r0-r7.
	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB invalid read exception
Control Abnormality	

16-bit instruction format:

st16.w rz, (rx, disp)

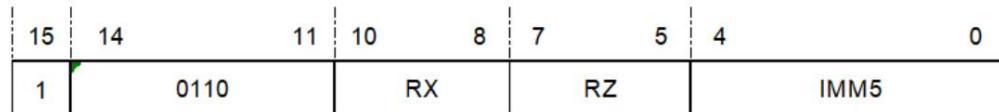


Figure 14.192: ST.W-1

st16.w rz, (sp, disp)

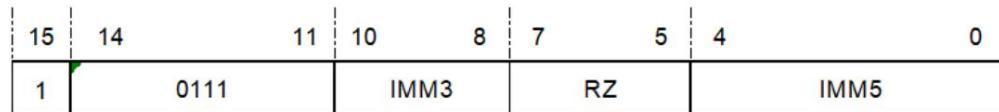


Figure 14.193: ST.W-2

32-bit instruction	
	to store word from register to memory Operation MEM[RX + zero_extend(offset << 2)] ÿ RZ[31:0] Syntax st32.w rz, (rx, disp) Description
	Store the word in register RZ into memory. Use register plus unsigned immediate offset addressing. The effective address of the memory is obtained by adding the base register RX to the 12-bit relative offset shifted left by two bits and unsigned extended to 32 bits. The ST32.W instruction can address +16KB address space. Note that the offset DISP is obtained by shifting the binary operand Offset left by two bits. No effect
Affects	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception
Flags	flag bit exception

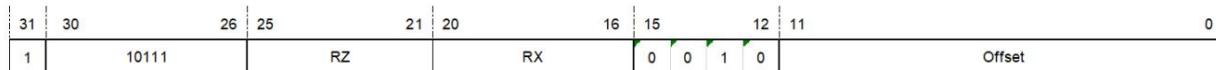
32-bit instruction format:

Figure 14.194: ST.W-3

14.127 STM -- Continuous Multi-Word Store Instruction

Unified instruction	
syntax	stm ry-rz, (rx) method
operation	Store the contents of a continuous register file to a continuous memory address. src \leftarrow Y; addr \leftarrow RX; for ($n = 0$; $n <= (ZY)$; $n++$) MEM[addr] \leftarrow Rsrc; src \leftarrow src + 1; addr \leftarrow addr + 4; } Only 32-bit instructions exist. stm32 ry-rz, (rx)
Compilation Result	Specified address of the memory, store the contents of register RY+1 to the address of the second word starting from the specified address of the memory, and so on, store the contents of register RZ to the address of the last word starting from the specified address of the memory. The effective address of the memory is determined by the contents of the base register RX. No effect
Impact flag	
limit	RZ should be greater than or equal to RY.
Control Abnormality	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit	
Operation	instruction stores the contents of a continuous register file to a continuous memory address. src \leftarrow Y; addr \leftarrow RX; for (n = 0; n <= IMM5; n++) { MEM[addr] \leftarrow Rsrc; src \leftarrow src + 1; addr \leftarrow addr + 4; } stm32
ry-rz, (rx)	
Syntax	Store the contents of a continuous register stack starting from RY to a continuous memory address in sequence, that is, store the contents of register RY to the address of the first word starting from the specified address of the memory, store the contents of register RY+1 to the address of the second word starting from the specified address of the memory, and so on, store the contents of register RZ to the address of the last word starting from the specified address of the memory. The effective address of the memory is determined by the contents of the base register
	RX. No effect
	RZ should be greater than or equal to RY.
Affects flag bit restriction exception	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit instruction format:

31	30	25	25	21	20	16	15	10	9	5	4	0
1	10101	RY	RX	000111	00001							IMM5

Figure 14.195: STM

IMM5 domain:

Specify the number of target registers, IMM5 = Z - Y.

00000:

1 destination register

00001:

2 destination registers

...

11111:

32 destination registers

14.128 STOP - Enter low power stop mode instruction

Unified	
instruction syntax	stop
operation	enters low power pause mode
Compilation result	Only 32-bit instructions stop32
	exist. This instruction causes the processor to enter low power mode and wait for an interrupt to exit this mode. At this time, the CPU clock stops and most peripherals are also stopped. No impact
Affects flag bit	
exception	privilege violation exception

32-bit	
instruction operation	to enter low-power stop
mode Syntax	stop32
Attributes:	Privileged instruction
Description	This instruction puts the processor into low-power mode and waits for an interrupt to exit this mode. At this time, the CPU clock stops, Most peripherals are also stopped. No
Affects flag bit	impact
exception	privilege violation exception

32-bit instruction format:



Figure 14.196: STOP

14.129 STQ——Continuous Four-Word Store Instruction

Unified instruction	
syntax	stq r4-r7, (rx) method
operation	<p>Store the words in registers R4-R7 in a continuous memory address. src \ddagger 4; addr \ddagger RX; for (n = 0; n \leq 3;</p> <p>n++) { MEM[addr] \ddagger Rsrc;</p> <p>src \ddagger src + 1; addr \ddagger</p> <p>addr + 4; } Only 32-bit instructions</p>
	exist. stq32 r4-r7, (rx);
Compilation address of the first word starting from the specified address of the memory, store the content of register R5 at the specified address of the memory, store the content of register R6 at the address of the third word starting from the content of register R7 at the address of the fourth word starting from the specified address of the memory. The effective address of the memory is determined by the content of the base register RX. Note that this instruction is a pseudo-instruction	
	of stm r4-r7, (rx). No effect
Affects	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception flag bit exception

The 32-bit	
Ý 4; addr { src addr + 4; }	instruction stores the words in registers R4-R7 in a continuous memory address. Operation src RX; for (n = 0; n <= 3; n++) { MEM[addr] Ý Rsrc; src Ý src + 1; addr Ý stq32 r4-r7, (rx)
Syntax	Store the words in the register stack [R4, R7] (including the boundary) in a continuous memory address in sequence, that is, store the content of register R4 at the address of the first word starting from the specified address of the memory, store the content of register R5 at the address of the second word starting from the specified address of the memory, store the content of register R6 at the address of the third word starting from the specified address of the memory, and store the content of register R7 at the address of the fourth word starting from the specified address of the memory. The effective address of the memory is determined by the content of the base register RX. Note that this instruction is a pseudo-instruction
	of stm r4-r7, (rx). No effect
Affects	Unaligned access exception, access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception flag bit exception

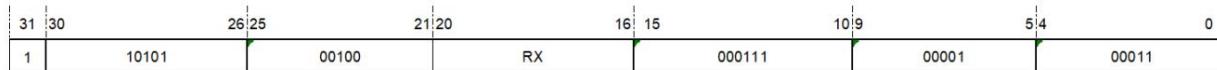
32-bit instruction format:

Figure 14.197: STQ

14.130 STR.B – Register Shift Addressed Byte Store Instruction

Unified	
Instruction Syntax	str.b rz, (rx, ry << 0) str.b rz, (rx, ry << 1) str.b rz, (rx, ry << 2) str.b rz, (rx, ry << 3) Operation Store the lowest byte in register
to memory	MEM[RX + RY << IMM2] → RZ[7:0] Only 32-bit instructions exist. str32.b rz, (rx, ry << 0) str32.b rz, (rx, ry << 1) str32.b rz, (rx, ry << 2) str32.b rz,
Compilation result	ry << 3) Description Store the lowest byte in register RZ to memory. Register plus register shift addressing mode is used. The effective address
of the memory	is obtained by adding the base register RX to the value of the offset register RY shifted left by 2 bits by the immediate value IMM2. The default value for IMM2 is 0. No effect
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit	
instruction operation	Store the lowest byte in the register to the memory MEM[RX + RY << IMM2] → RZ[7:0] Syntax str32.b rz, (rx, ry << 0) str32.b rz, (rx, ry << 1) str32.b rz, (rx, ry << 2) str32.b rz, (rx, ry << 3) Description Store the lowest byte in register RZ to the memory. Register
plus register shift addressing mode	is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit instruction format:

str32.b rz, (rx, ry << 0)

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10101	RY	RX	000000	00001	RZ	

Figure 14.198: STR.B-1

str32.b rz, (rx, ry << 1)

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10101	RY	RX	000000	00010	RZ	

Figure 14.199: STR.B-2

str32.b rz, (rx, ry << 2)

31 30	26 25	21 20	16 15	10 9	5 4	0
1 10101	RY	RX	000000	00100	RZ	

Figure 14.200: STR.B-3

str32.b rz, (rx, ry << 3)

14.131 STR.H—Register Shift Addressing Half-Word Store Instruction

Unified	
command syntax	str.h rz, (rx, ry << 0) str.h rz, (rx, ry << 1) str.h rz, (rx, ry << 2) str.h rz, (rx, ry << 3)
The operation	stores the lower halfword of a register into memory. MEM[RX + RY << IMM2] → RZ[15:0] Only 32-bit instructions
Compilation result	exist. str32.h rz, (rx, ry << (rx, ry << 1) str32.h rz, (rx, ry << 2) str32.h rz, (rx, ry << 3) Description Store the lower halfword in register RZ to the
memory	Register plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000000	01000	RZ

Figure 14.201: STR.B-4

32-bit	
instruction operation	stores the lower halfword of a register into memory MEM[$RX + RY \ll IMM2$] $\leftarrow RZ[15:0]$ Syntax str32.h rz,
(rx, ry << 0) str32.h rz, (rx, ry << 1) str32.h rz, (rx, ry << 2) str32.h rz, (rx, ry << 3) Description Store the lower halfword in register RZ to the	
memory.	Register plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0. No effect
Affects flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit instruction format:

str32.h rz, (rx, ry << 0)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000001	00001	RZ

Figure 14.202: STR.H-1

str32.h rz, (rx, ry << 1)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000001	00010	RZ

Figure 14.203: STR.H-2

str32.h rz, (rx, ry << 2)

str32.h rz, (rx, ry << 3)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000001	00100	RZ

Figure 14.204: STR.H-3

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000001	01000	RZ

Figure 14.205: STR.H-4

14.132 STR.W—Register Shift Addressed Word Store Instruction

Unified	
instruction syntax	str.w rz, (rx, ry << 0) str.w rz, (rx, ry << 1) str.w rz, (rx, ry << 2) str.w rz, (rx, ry << 3) Operation Store the word in the register to
the memory	MEM[RX + RY << IMM2] & RZ[31:0] Only 32-bit instructions
Compilation result	exist. str32.w rz, (rx, ry << str32.w rz, (rx, ry << 1) str32.w rz, (rx, ry << 2) str32.w rz, (rx, ry << 3) Description Store the word in register RZ into memory. Register
	plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits to the immediate value IMM2. The default value of IMM2 is 0.
Affects flag bit	No impact
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit	
instruction operation	stores a word from a register into memory MEM[RX + RY << IMM2] → RZ[31:0] Syntax str32.w
rz, (rx, ry << 0) str32.w rz, (rx, ry << 1) str32.w	 rz, (rx, ry << 2) str32.w rz, (rx, ry << 3) Description Store the word in register RZ into the memory.
Register	plus register shift addressing mode is used. The effective address of the memory is obtained by adding the base register RX to the offset register RY shifted left by 2 bits of the immediate value IMM2. The default value of IMM2 is 0.
Affects	No impact
flag bit	
exceptions	Unaligned access exception, Access error exception, TLB unrecoverable exception, TLB mismatch exception, TLB write invalid exception

32-bit instruction format:

str32.w rz, (rx, ry << 0)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000010	00001	RZ	

Figure 14.206: STR.W-1

str32.w rz, (rx, ry << 1)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000010	00010	RZ	

Figure 14.207: STR.W-2

str32.w rz, (rx, ry << 2)

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000010	00100	RZ	

Figure 14.208: STR.W-3

str32.w rz, (rx, ry << 3)

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10101	RY	RX	000010	01000	RZ

Figure 14.209: STR.W-4

14.133 SUBC – Unsigned Subtract with Borrow Instruction

Unified instruction		
	subc rz, rx	subc rz, rx, ry
	RZ ѕ RZ - RX - (IC), C ѕ borrow is compiled into	RZ ѕ RX - RY - (IC), C ѕ borrow is compiled into
	the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (z<16), then subc16 rz, rx; syntax operation compilation results else subc32 rz, rz, rx; subc32 rz, rx, ry; For subc rz, rx, subtract the value of register RX and the negation	the corresponding 16-bit or 32-bit instruction according to the range of the register. if (x==z) and (y<16) and (z<16), then subc16 rz, ry; else
illustrate	of C from the value of RZ; for subcrz, rx, ry, subtract the value of register RY and the negation of C from the value of RX. Store the result in RZ, and the borrow in C. For this subtraction instruction, if a borrow occurs, the C bit will be cleared, otherwise it will be set.	
	C ѕ Borrow	
Affects flag bit exception	none	

16-bit	
instruction	operation RZ ѕ RZ - RX - (IC), C ѕ borrow Syntax
subc16 rz, rx	Description Subtract
	the value of register RX and the negation of C from the value of RZ, and store the result in RZ. The borrow is stored in C. For this subtraction instruction, if a borrow occurs, C is cleared, otherwise it is set.
The range of the	C ѕ Borrow
affected flag	limit register is r0- r15 .

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11000		RZ		RX		11	

Figure 14.210: SUBC-1

32-bit	
instruction operation	RZ - RX - (IC), C borrow Syntax subc32
rz, rx, ry	Description Subtract the value
	of register RY and the negation of the C bit from the value of RX, and store the result in RZ, with the borrow in C. For this subtraction instruction, if a borrow occurs, the C bit will be cleared, otherwise it will be set.
Affects	C Borrow
flag	
bit abnormality	

32-bit instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10001		RY		RX		00000		01000		RZ	

Figure 14.211: SUBC-2

14.134 SUBI – Subtract Immediate Unsigned Instruction

Unified command		
syntax	ubi rz, oimm12	subi rz, rx, oimm12
S	RZ $\ddot{\vee}$ RZ - zero_extend(OIMM12) RZ $\ddot{\vee}$ RX - zero_extend(OIMM12)	
Operation	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (oimm12<8) and (z<8) and (x<8), then subi16 rz, rx, oimm3; elseif (x==z) and (z<8) and (oimm12<257), then subi16 rz, oimm8; else subi32 rz, rx, oimm12;	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (oimm12<8) and (z<8) and (x<8), then subi16 rz, rx, oimm3; elseif (x==z) and (z<8) and (oimm12<257), then subi16 rz, oimm8; else subi32 rz, rx, oimm12;
	Zero-extend the 12-bit immediate value (OIMM12) with offset 1 to 32 bits, then subtract the 32-bit value from the value of RZ/RX and store the result in RZ. Note: Affects the flag bit limit exception	
	effect	
	The range of immediate values is 0x1-0x1000.	
	none	

16-bit instruction	
1 Operation	RZ $\ddot{\vee}$ RZ - zero_extend(OIMM8) Syntax subi16
rz, oimm8	Description Zero-extend the 8-bit immediate value (OIMM8) with offset 1 to 32 bits, then subtract the 32-bit value from the value of RZ. The result is stored in RZ. Note: Binary operand IMM8 is equal to OIMM8 - 1.
The range of the	No effect
affected flag limit	register is r0-r7; the range of the immediate value is 1-256 .

16-bit instruction format1:

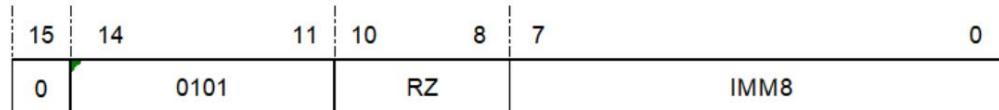


Figure 14.212: SUBI-1

IMM8 domain:

Specifies the value of an immediate value without an offset.

Note: The value to be subtracted from register OIMM8 must be offset by 1 compared to the binary operand IMM8.

00000000:

Minus 1

00000001:

Minus 2

...

11111111:

Subtract 256

16-bit instruction	
2 Operation	RZ - RX - zero_extend(OIMM3) Syntax subi16 rz,
rx, oimm3 Description	Zero-extend the 3-bit immediate value (OIMM3) with offset 1 to 32 bits, then subtract the 32-bit value from the value in RX.
	The result is stored in RZ. Note: Binary operand IMM3 is equal to OIMM3 - 1.
The range of the affected flag limit	No effect
register	r0-r7; the range of the immediate value is 1-8 .

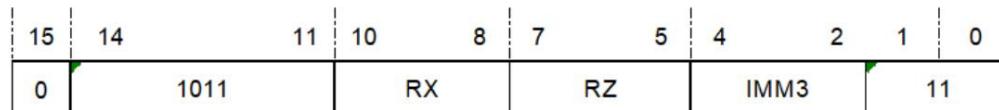
16-bit instruction format 2:

Figure 14.213: SUBI-2

IMM3 domain

Specifies the value of an immediate value without an offset.

Note: The value to be subtracted from register OIMM3 must be offset by 1 compared to the binary operand IMM3.

000

Minus 1

001

Minus 2

...

111

Minus 8

32-bit	
instruction operation	RZ - zero_extend(OIMM12)
Syntax	subi32 rz, rx, oimm12 Description
	Zero-extend the 12-bit immediate value (OIMM12) with offset 1 to 32 bits, then subtract the 32-bit value from the value of RX.
	The result is stored in RZ. Note: Binary operand IMM12 is equal to OIMM12
The flag bit	- 1. No effect
affects	the immediate value range to 0x1-0x1000.

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	11001		RZ		RX	0	0	0	1	IMM 12

IMM12 domain

Specifies the value of an immediate value without an offset.

Note: The value to be subtracted from register OIMM12 must be offset by 1 compared to the binary operand IMM12.

000000000000

Subtract 0x1

000000000001

Subtract 0x2

...

111111111111

Subtract 0x1000

14.135 SUBI(SP) – Subtract Immediate Unsigned (Stack Pointer) Instruction

Unified instruction	
syntax	subi sp, sp, imm operation SP - SP-
zero_extend(IMM)	The compilation result only has 16-bit instructions.
	subi sp, sp, imm Zero-
illustrate	extends the immediate value (IMM) to 32 bits, shifts it left 2 bits, and subtracts it from the stack pointer (SP), placing the result in SP.
Affects flag bit	No impact
	The range of immediate value is 0x0-0x1fc.
restriction exception	

16-bit	
instruction	operation SP - SP - zero_extend(IMM) Syntax subi
sp, sp, imm	Description Zero-extend the
	immediate value (IMM) to 32 bits and shift it left by 2 bits, then subtract it from the value of the stack pointer (SP) and store the result in the stack
	Stack
	pointer. Note: The immediate value (IMM) is equal to the binary operand {IMM2, IMM5} << 2. No effect
The source and	
destination	registers of the affected flag bit are both the stack instruction register (R14); the range of the immediate value is (0x0-0x7f) << 2.

16-bit instruction format:

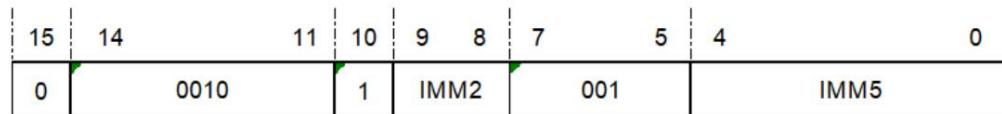


Figure 14.214: SUBI (SP)

IMM domain:

Specifies the value of an immediate value without shifting.

Note: The value added to register IMM is shifted left 2 bits compared to the binary operand {IMM2, IMM5}.

{00, 00000}

Subtract 0x0

{00, 00001}

Subtract 0x4

...

{11, 11111}

Subtract 0x1fc

14.136 SUBU -- UNSIGNED SUBTRACT INSTRUCTION

Unified instruction		
	subu rz, rx sub rz, rx	subu rz, rx, ry
	RZ $\ddot{\vee}$ RZ - RX	RZ $\ddot{\vee}$ RX - RY
syntax operation combination results	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (z<8) and (x<8) and (y<8), then subu16 rz, rx, ry; elsif (x==z) and (z<16) and (y<16), then subu16 rz, ry; else subu32 rz, rx, ry;	Compile to the corresponding 16-bit or 32-bit instruction according to the register range. if (z<8) and (x<8) and (y<8), then subu16 rz, rx, ry; elsif (x==z) and (z<16) and (y<16), then subu16 rz, ry; else subu32 rz, rx, ry;
	For subu rz, rx, subtract the RX value from the RZ value and store the result in RZ. For subu rz, rx, ry, subtract the RY value from the RX value and store the result in RZ. No effect	
	none	
Description: Affects the flag bit abnormality		

16-bit instruction 1	
	RZ $\ddot{\vee}$ RZ - RX
Operation syntax	subu16 rz, rx sub16 rz, rx
	Subtract the value of RX from the value of RZ and store the result in RZ.
Description: The impact flag has	
no	The register range is r0-r15.
impact limit exception	

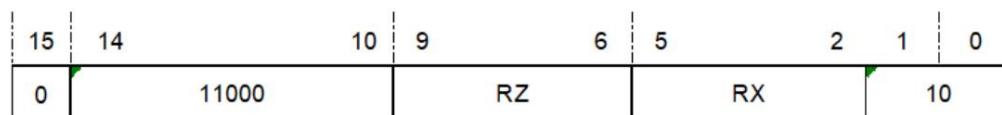
16-bit instruction format1:

Figure 14.215: SUBU-1

16-bit instructions2	
	RZ - RX-RY
Operation Syntax	subu16 rz, rx, ry sub16 rz, rx, ry
	Subtract the RY value from the RX value and store the result in RZ.
Description:	The impact flag has no impact
Limit	The register range is r0-r7.
exception	none

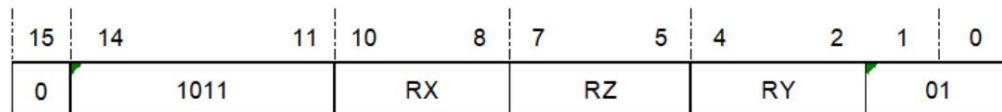
16-bit instruction format 2:

Figure 14.216: SUBU-2

32-bit instructions	
	RZ - RX - RY
	subu32 rz, rx, ry
	Subtract the RY value from the RX value and store the result in RZ.
Operation Syntax	Description Impact Flags No Impact
Abnormal	

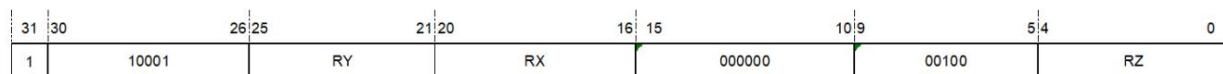
32-bit instruction format:

Figure 14.217: SUBU-3

14.137 SYNC——CPU Synchronization Instruction

Unified command	
syntax	sync.is sync.i sync.s sync
Operate	this instruction to ensure that all previous instructions retire earlier than this instruction, and all subsequent instructions retire later than this instruction.
The compilation result	only contains 32-bit instructions. sync32.is sync32.i sync32.s sync32
indicates	whether S represents broadcasting outside the CPU core, and "1" is valid. I Indicates whether to synchronously fetch values when the instruction is retired and execute pipeline clearing operation. "1" is
Affects flag	valid. No effect
bit	
abnormality	

The 32-bit	
instruction operation	synchronizes
the CPU. Syntax:	sync32
Description:	S represents whether to broadcast to the outside of the CPU core. "1" is valid. I Indicates whether to synchronously fetch values when the instruction is retired and execute pipeline clearing operation. "1" is
Affects flag	valid. No effect
bit	
abnormality	

32-bit instruction format:

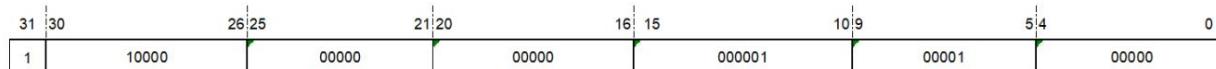


Figure 14.218: SYNC

14.138 TRAP — Operating System Trap Instruction

Unified instructions	
grammar	trap 0, trap 1 trap 2, trap 3
Operation	Causes a trap exception to occur
compilation results	Only 32-bit instructions exist.
illustrate	trap32 0, trap32 1 trap32 2, trap32 3 When the processor encounters a trap instruction, a trap exception operation occurs.
Affects flag bit	No effect
Exception	Trap Exception

32-bit instructions	
	Causes a trap exception to occur
Operation Syntax	trap32 0, trap32 1, trap32 2, trap32 3
	When the processor encounters a trap instruction, a trap exception operation occurs.
Description:	The impact flag has no impact
Exception	Trap Exception

32-bit instruction format:

trap32 0

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10000		00000		00000		001000		00001		00000	

Figure 14.219: TRAP-1

trap32 1

31	30	26	25	21	20	16	15	10	9	5	4	0
1	10000		00000		00000		001001		00001		00000	

Figure 14.220: TRAP-2

trap32 2

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10000	00000	00000	001010	00001	00000

Figure 14.221: TRAP-3

trap32 3

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10000	00000	00000	001011	00001	00000

Figure 14.222: TRAP-4

14.139 TST——Zero Test Instruction

Unified	
instruction syntax	tst rx,
ry operation	If (RX & RY) != 0, then C = 1; else C = 0; The
compilation result	is compiled into the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (y<16), then tst16 rx, ry; else tst32 rx, ry;
Description	Test the bitwise AND result of the values of RX and RY. If the result is not equal to 0, set the condition bit C; otherwise, clear the condition bit C. Set
Affects	the condition bit C according to the bitwise AND result
flag bit abnormality	

16-bit instructions	
OperationIf (RX & RY) != 0, then	C = 1; else C = 0;
Syntax	tst16 rx, ry
	Test the bitwise AND result of the values of RX and RY. If the result is not equal to 0, condition bit C is set; otherwise, condition bit C is cleared.
Impact Signs	Set condition bit C according to the bitwise AND result
Bit	
Limit	The register range is r0-r15.
exception	none

16-bit instruction format:

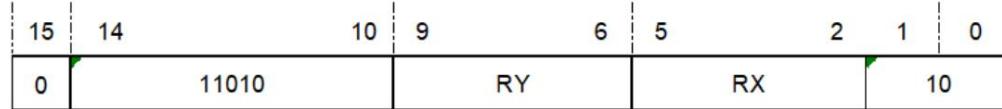


Figure 14.223: TST-1

32-bit instructions	
OperationIf (RX & RY) != 0, then	C = 1; else C = 0;
Syntax	tst32 rx, ry
	Test the bitwise AND result of the values of RX and RY. If the result is not equal to 0, condition bit C is set; otherwise, condition bit C is cleared.
Impact Signs	Set condition bit C according to the bitwise AND result
Bit	
abnormal	none

32-bit instruction format:

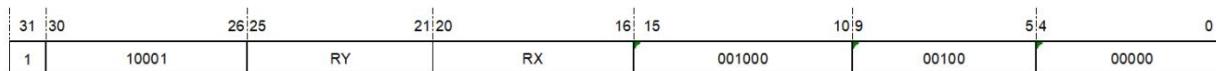


Figure 14.224: TST-2

14.140 TSTNBZ——No Byte Equals Zero Register Test Instruction

Unified	
instruction syntax	tstnbz16
rx operation	If ((RX[31:24] != 0) &(RX[23:16] != 0) &(RX[15: 8] != 0) &(RX[7 : 0] != 0)), then C = 1; else C = 0; The
compilation result	is compiled into the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16), then tstnbz16 rx; else tstnbz32 rx;
Description	Tests whether no byte in RX is equal to zero. If no byte in RX is equal to zero, condition bit C is set; otherwise, clear Condition bit C.
Affects	Set condition bit C according to the bitwise AND result
flag bit abnormality	

16-bit	
instruction operation	If ((RX[31:24] != 0) &(RX[23:16] != 0) &(RX[15: 8] != 0) &(RX[7 : 0] != 0)), then C = 1; else C = 0;
Syntax	tstnbz16 rx
Description	Tests whether no byte in RX is equal to zero. If no byte in RX is equal to zero, set condition bit C; otherwise, clear Condition bit C.
The range of	Set condition bit C according to the bitwise AND result
the affected flag limit	register is r0- r15 .

16-bit instruction format:

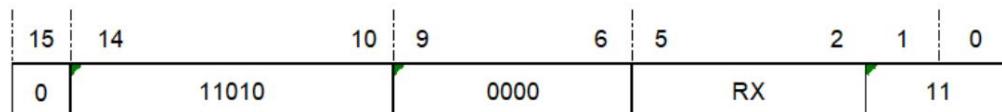


Figure 14.225: TSTNBZ-1

32-bit	
instruction operation	If ((RX[31:24] != 0) & (RX[23:16] != 0) & (RX[15:8] != 0) & (RX[7:0] != 0)), then C = 1; else C = 0;
Syntax	tstnbz32 rx
Description	Test if no byte in RX is equal to zero. If no byte in RX is equal to zero, set condition bit C; otherwise, clear Condition bit C.
Affects	Set condition bit C according to the bitwise AND result
flag bit abnormality	

32-bit instruction format:

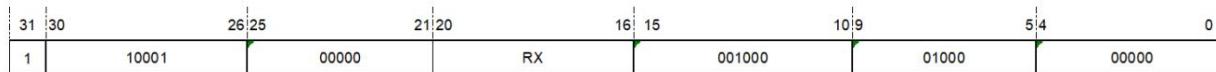


Figure 14.226: TSTNBZ-2

14.141 WAIT——Enter low power wait mode instruction

Unified	
instruction syntax	
Wait operation	Enter low power wait mode
Compilation	Only 32-bit instructions exist. wait32
result Attribute:	Privileged
instruction Description	This instruction stops the current instruction execution and waits for an interrupt. At this time, the CPU clock stops. All peripheral devices continue to operate. Running, and may generate an interrupt causing the CPU to exit from wait mode. No effect
Affects flag	
bit exception	privilege violation instruction

32-bit	
instruction operation	to enter low-power wait
mode Syntax	wait32
Attribute:	Privileged
instruction Description	This instruction stops the current instruction execution and waits for an interrupt, at which time the CPU clock stops. All peripheral devices continue to run and may generate interrupts to cause the CPU to exit from wait mode. No
Affects flag	effect
bit exception	privilege violation instruction

32-bit instruction format:

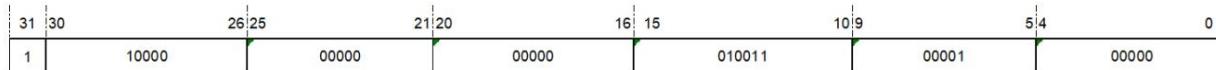


Figure 14.227: WAIT

14.142 XOR——Bitwise Exclusive OR Instruction

Unified instructions		
language Law	XOR RZ, RX	xor rz, rx, ry
Hold do	RZ $\ddot{\wedge}$ RZ $\ddot{\wedge}$ RX	RZ $\ddot{\wedge}$ RX $\ddot{\wedge}$ RY
Edit Translation Knot fruit	Compile to the corresponding 16-bit or 32-bit register range Bit instruction. if (x<16) and (z<16), then xor16 rz, rx; else xor32 rz, rz, rx;	Compile into corresponding 16-bit or 32-bit instructions according to the range of registers make. if (y==z) and (z<16) and (x<16), then xor16 rz, rx; else xor32 rz, rx, ry;
explain bright	Perform bitwise XOR of the values of RX and RZ/RY and store the result in RZ.	
film ring Standard Zhi Bit	No impact	
different often	none	

16-bit instructions		
	RZ $\ddot{\wedge}$ RZ $\ddot{\wedge}$ RX	
	xor16 rz, rx	
	Perform bitwise XOR of the values in RZ and RX and store the result in RZ.	
Operation Syntax	Description	Impact Flags No Impact
Limit	The register range is r0-r15.	
exception	none	

16-bit instruction format:

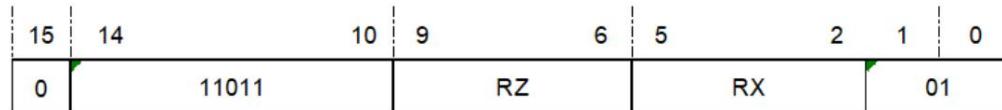


Figure 14.228: XOR-1

32-bit instructions	
	RZ $\ddot{\wedge}$ RX ^ RY
	xor32 rz, rx, ry
	Perform bitwise XOR of the values of RX and RY and store the result in RZ.
Operation Syntax	Description Impact Flags No Impact
abnormal	none

32-bit instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1	10001	RY	RX	001001	00010	RZ

Figure 14.229: XOR-2

14.143 XORI—Immediate Bitwise XOR Instruction

Unified instructions	
Syntax	xori rz, rx, imm16
	RZ $\ddot{\wedge}$ RX ^ zero_extend(IMM12)
Operation Compiled results	contain only 32-bit instructions. xori32 rz, rx, imm12
	Zero-extend the 12-bit immediate value to 32 bits, then perform a bitwise XOR operation with the value in RX and store the result in RZ.
Description:	The impact flag has no impact
Limit	the range of immediate values to 0x0-0xFFFF.
Abnormal	

32-bit instructions	
	RZ $\ddot{\wedge}$ RX ^ zero_extend(IMM12)
	xori32 rz, rx, imm12
	Zero-extend the 12-bit immediate value to 32 bits, then perform a bitwise XOR operation with the value in RX and store the result in RZ.
Operation Syntax	Description Impact Flags No Impact
Limit	the range of immediate values to 0x0-0xFFFF.
Abnormal	

32-bit instruction format:

31	30	26	25	21	20	16	15	12	11	0
1	11001		RZ		RX		0100			IMM 12

Figure 14.230: XORI

14.144 XSR—Extended Shift Right Instruction

Unified instruction	
syntax	xsr rz, rx, oimm5 method operation
	{RZ,C} ý(RX,C) >>> OIMM5
	Only 32-bit instructions exist. xsr32 rz, rx, oimm5
compilation bright	destination description: RX with conditional bit C ((RX,C)) right circularly (shift the original value right, shift the left bit into the bit shifted out on the right), store the lowest bit ([0]) of the shift result into conditional bit C, and the high bit ([32:1]) into RZ, the number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then conditional bit C is the highest bit of RX.
	C ý RX[OIMM5 - 1]
	The range of immediate value is 1-32.
Affects flag bit restriction exception	none

32-bit instruction	
	{RZ,C} $\ddot{\vee}$ {RX,C} >>> OIMM5
	xsr32 rz, rx, oimm5
operation system level	Shift the value of RX with conditional bit C ({RX,C}) right circularly (shift the original value right, shift the left bit into the bit shifted out on the right), store the lowest bit ([0]) of the shift result into conditional bit C, and the high bit ([32:1]) into RZ. The number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. If the value of OIMM5 is equal to 32, then conditional bit C is the highest bit of RX. Note: The binary operand IMM5 is equal to OIMM5 - 1.
	C $\ddot{\vee}$ RX[OIMM5 - 1]
	The range of immediate value is 1-32.
	none
Affects flag bit restriction exception	

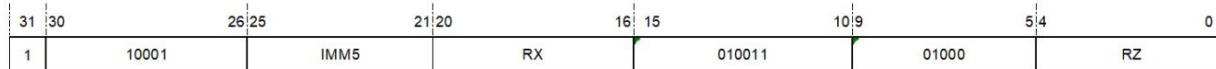
32-bit instruction format:

Figure 14.231: XSR

IMM5 domain

Specifies the value of an immediate value without an offset.

Note: The shift value OIMM5 must be offset by 1 compared to the binary operand IMM5.

00000

Shift 1 bit

00001

Shift 2 places

...

11111

Shift 32 bits

14.145 XTRB0 - Extract Byte 0 without Sign Extension Instruction

Unified	
instruction syntax	xtrb0 rz, rx
operation	RZ \leftarrow zero_extend(RX[31:24]); if (RX[31:24] == 0), then C \leftarrow 0; else C \leftarrow 1;
Only 32-bit instructions exist in the compilation result . Result	
xtrb0.32 rz, rx	Description Extract byte 0 (RX[31:24]) of RX to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C
Affects flag	bit. If the result is equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit	
instruction operation	RZ \leftarrow zero_extend(RX[31:24]); if (RX[31:24] == 0), then C \leftarrow 0; else C \leftarrow 1;
Syntax	xtrb0.32 rz, rx
Description	Extract byte 0 (RX[31:24]) of RX to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C bit. If the result is
Affects flag	equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit instruction format:

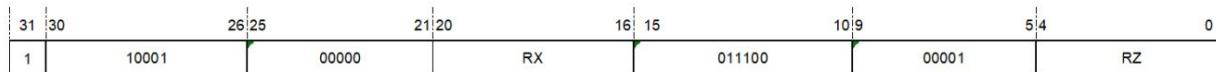


Figure 14.232: XTRB0

14.146 XTRB1 - Extract Byte 1 without Sign Extension Instruction

Unified	
instruction syntax	xtrb1 rz, rx
operation	RZ \leftarrow zero_extend(RX[23:16]); if (RX[23:16] == 0), then C \leftarrow 0; else C \leftarrow 1;
Only 32-bit instructions exist in the compilation result . Result	
xtrb1.32 rz, rx	Description Extract byte 1 of RX (RX[23:16]) to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C
Affects flag	bit. If the result is equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit	
instruction operation	RZ \leftarrow zero_extend(RX[23:16]); if (RX[23:16] == 0), then C \leftarrow 0; else C \leftarrow 1;
Syntax	xtrb1.32 rz, rx
Description	Extract byte 1 of RX (RX[23:16]) to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C bit. If the result is
Affects flag	equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit instruction format:

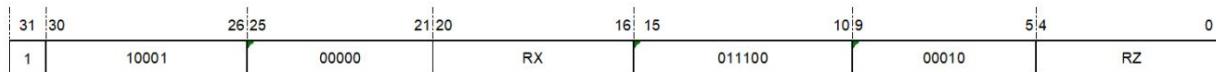


Figure 14.233: XTRB1

14.147 XTRB2 - Extract Byte 2 without Sign Extension Instruction

Unified	
instruction syntax	xtrb2 rz,
rx operation	RZ \leftarrow zero_extend(RX[15:8]); if (RX[15:8] == 0), then C \leftarrow 0; else C \leftarrow 1;
Compiled result:	Only 32-bit instructions exist. Result xtrb2.32 rz, rx
Description	Extract byte 2 (RX[15:8]) of RX to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C bit.
Affects flag	the result is equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit	
instruction operation	RZ \leftarrow zero_extend(RX[15:8]); if (RX[15:8] == 0), then C \leftarrow 0; else C \leftarrow 1;
Syntax	xtrb2.32 rz, rx Description
	Extract byte 2 (RX[15:8]) of RX to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, the C bit is cleared, otherwise the C bit is set. If the result
Affects flag	is equal to 0, the C bit is cleared, otherwise the C bit is set.
bit abnormality	

32-bit instruction format:

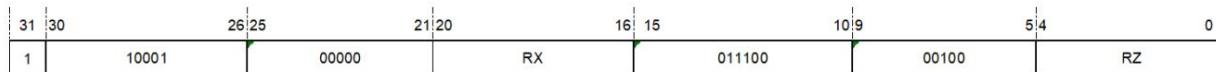


Figure 14.234: XTRB2

14.148 XTRB3 - Extract Byte 3 without Sign Extension Instruction

Unified	
instruction syntax	xtrb3 rz,
rx operation	RZ \leftarrow zero_extend(RX[7:0]); if (RX[7:0] == 0), then C \leftarrow 0; else C \leftarrow 1;
Compiled result:	Only 32-bit instructions
exist. Result	xtrb3.32 rz, rx
Description	Extract byte 3 (RX[7:0]) of RX to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C bit. If the
Affects flag	result is equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit	
instruction operation	RZ \leftarrow zero_extend(RX[7:0]); if (RX[7:0] == 0), then C \leftarrow 0; else C \leftarrow 1;
Syntax	xtrb3.32 rz, rx
Description	Extract byte 3 (RX[7:0]) of RX to the lower bit of RZ (RZ[7:0]) and perform zero extension. If the result is equal to 0, clear the C bit, otherwise set the C bit. If the result
Affects flag	is equal to 0, clear the C bit, otherwise set the C bit.
bit abnormality	

32-bit instruction format:

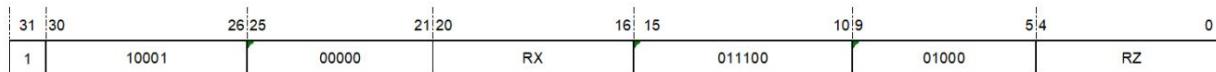


Figure 14.235: XTRB3

14.149 ZEXT -- Bit Extract and No Sign Extension Instruction

Unified command	
	zext rz, rx, msb,
syntax operation	RZ \leftarrow zero_extend(RX[MSB:LSB])
lsb	Only 32-bit instructions exist. zext32 rz, rx, msb, lsb
Compilation Result	Extract a continuous range of bits (RX[MSB:LSB]) of RX specified by two 5-bit immediate values (MSB, LSB), zero-extend to 32 bits, and store the result in RZ. If MSB is equal to 31 and LSB is equal to 0, the value of RZ is the same as RX. If MSB is equal to LSB, the value of RZ is the result of zero-extending RX[MSB] (ie, RX[LSB]) by one bit. If MSB is less than LSB, the behavior of this instruction is unpredictable. No effect
Impact Mark	
Zhi	
Bit	
	The range of MSB is 0-31, the range of LSB is 0-31, and MSB should be greater than or equal to LSB.
Limit Exception	none

32-bit instruction	
	RZ \leftarrow zero_extend(RX[MSB:LSB])
	zext32 rz, rx, msb, lsb
	<p>Extract a continuous range of bits (RX[MSB:LSB]) of RX specified by two 5-bit immediate values (MSB, LSB), zero-extend to 32 bits, and store the result in RZ. If MSB is equal to 31 and LSB is equal to 0, the value of RZ is the same as RX. If MSB is equal to LSB, the value of RZ is the result of zero-extending RX[MSB] (ie, RX[LSB]) by one bit. If MSB is less than LSB, the behavior of this instruction is unpredictable.</p> <p>No effect</p>
	The range of MSB is 0-31, the range of LSB is 0-31, and MSB should be greater than or equal to LSB.
	none
	Affects flag bit restriction exception

32-bit instruction format:

Figure 14.236: ZEXT

MSB field

Specifies the bit to be extracted starting from.

LSB field

Specifies the end bit to be extracted.

00000

0

00000

0 position

00001

1

00001

1 bit

.....
11111

31

11111

31 bits

14.150 ZEXTB -- Extract Byte Without Sign Extension Instruction

Unified	
instruction syntax	<code>zextb rz, rx</code>
operation	<code>RZ ← zero_extend(RX[7:0]); The compilation</code>
result	is compiled into the corresponding 16-bit or 32-bit instruction according to the register range. if ($x < 16$) and ($z < 16$), then <code>zextb16 rz, rx;</code> else <code>zextb32 rz, rx</code>
Description	Zero-extend the low byte of RX (RX[7:0]) to 32 bits, and store the result in RZ. Affects flags
None Affects bits	
Exception None	

16-bit	
	<code>RZ ← zero_extend(RX[7:0]); zextb16 rz,</code>
	rx Zero-extend the
	low byte of RX (RX[7:0]) to 32 bits and store the result in RZ.
instruction operation syntax	
	The register range is r0-r15.
description affected flags no impact limit exception	

16-bit instruction format:

15	14	10	9	6	5	2	1	0
0	11101		RZ		RX		00	

Figure 14.237: ZEXTB-1

32-bit	
instruction operation	RZ ý zero_extend(RX[7:0]); Syntax
zextb32 rz, rx	Description Zero-extend the low byte of RX (RX[7:0]) to 32 bits, and store the result in RZ. Note that this instruction is a pseudo-instruction of zext32 rz, rx, 0x7, 0x0. No effect
Affects flag	
bit	
abnormality	

32-bit instruction format:

31	30	26 25	21 20	16 15	10 9	5 4	0
1	10001	00000	RX	010101	00111	RZ	

Figure 14.238: ZEXTB-2

14.151 ZEXTH -- Extract Halfword Without Sign-Extend Instruction

Unified	
instruction syntax	zexth rz, rx
operation	RZ ý zero_extend(RX[15:0]); The compilation result
is compiled into	the corresponding 16-bit or 32-bit instruction according to the register range. if (x<16) and (z<16), then zexth16 rz, rx; else zexth32 rz, rx
Description	Zero-extend the lower halfword of RX (RX[15:0]) to 32 bits and store the result in RZ. Affects flags
None Affects bits Exceptions	
None	

16-bit	
	RZ ý zero_extend(RX[15:0]); zexth16 rz, rx
	Zero-extend the
	lower halfword of RX (RX[15:0]) to 32 bits and store the result in RZ.
instruction operation	syntax
	The register range is r0-r15.
description affected	flags no impact limit exception

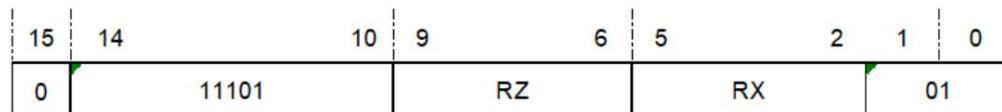
16-bit instruction format:

Figure 14.239: ZEXTH-1

32-bit	
instruction operation	RZ = zero_extend(RX[15:0]); Syntax
zext32 rz, rx	Description Zero-extend
	the lower halfword of RX (RX[15:0]) to 32 bits, and store the result in RZ. Note that this instruction is a pseudo-instruction of zext32 rz, rx, 0x15, 0x0. No effect
Affects flag	
bit	
abnormality	

32-bit instruction format:

Figure 14.240: ZEXTH-2

Chapter 15 Appendix B DSP Instruction Set Glossary

The following describes the DSP instruction subsets in alphabetical order.

The mnemonic of each DSP instruction contains three parts, each part is separated by . The first part identifies the basic operation of the instruction, the second part identifies the data type of the operand, and the third part identifies the processing of the operation result. Take "16-bit parallel signed addition instruction with saturation operation-PADD.S16.S" as an example, PADD indicates that this is a parallel addition instruction, S16 indicates that the operand is a signed half word, and S indicates that the operation result is saturated.

Common letters in instruction mnemonics and their meanings are as follows:

Mnemonic	English meaning	
	P (Parallel)	
Part 1 Part 1	Parallel computing, indicating single instruction multiple data operations.	
Part 1 Part 1	R(Raction) Fractional operation.	
Part 2 Part 3	X(Unalign) The operand width is not aligned, or the operand position is not aligned.	
Part 3 Part 3	H(Half)	Take the average.
	(S/U)/(8/16/32)(signed/unsigned)/(byte/halfword/word)	If there is no sign bit indication, it means that the operation of this instruction does not care about the sign bit.
	S (Saturate)	Saturation operation.
	R (Round)	rounding operation.
	E (Extension)	extension operation, that is, the operands are extended and then participate in the calculation to produce the result.

15.1 PADD.8——8-bit parallel addition instruction

Unified	
instruction syntax	padd.8 rz, rx, ry
operation	$Rz[31:24] = Rx[31:24] + Ry[31:24]$ $Rz[23:16] = Rx[23:16] + Ry[23:16]$ $Rz[15:8] = Rx[15:8] + Ry[15:8]$ $Rz[7:0] = Rx[7:0] + Ry[7:0]$ Only 32-bit
Compilation results	instructions exist. padd.8 rz, rx, ry

Description:	In byte units, add the 4 bytes of Rx to the 4 bytes of Ry, and store the addition result in Rz. Affected flags: Exception:
No 32-bit	No impact
instruction	
operation:	
$Rz[31:24] = Rx[31:24] + Ry[31:24]$	
$Rz[23:16] = Rx[23:16] + Ry[23:16]$	
$Rz[15:8] = Rx[15:8] + Ry[15:8]$	
$Rz[7:0] = Rx[7:0] + Ry[7:0]$	
Syntax:	padd.8 rz, rx, ry Description:
	In byte units, add the 4 bytes of Rx to the 4 bytes of Ry, and store the addition result in Rz. Affected flags: Exception: None
	No impact

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	1	1	1	1	0	RY	RX	1	1	0	0	X

Figure 15.1: PADD.8

15.2 PADD.16——16-bit parallel addition instruction

Unified instructions	
Syntax	padd.16 rz, rx,ry
Operations	Rz[31:16] = Rx[31:16] + Ry[31:16] Rz[15:0] = Rx[15:0] + Ry[15:0]
The compiled result	contains only 32-bit instructions. padd.16 rz, rx,ry

	Add the upper and lower halfwords of Rx to the upper and lower halfwords of Ry respectively, and store the result in Rz.
Description: Impact flag:	No impact
	none
Exceptions: 32-bit instructions	
operate:	Rz[31:16] = Rx[31:16] + Ry[31:16] Rz[15:0] = Rx[15:0] + Ry[15:0]
Syntax:	padd.16 rz, rx,ry
	Add the upper and lower halfwords of Rx to the upper and lower halfwords of Ry respectively, and store the result in Rz.
Description: Impact flag:	No impact
Abnormal:	None

Instruction format:

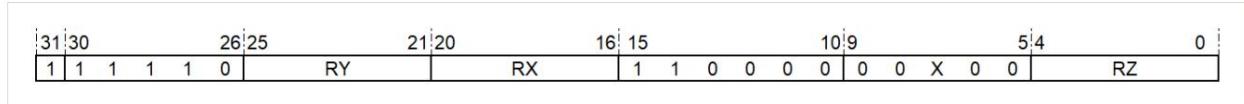


Figure 15.2: PADD.16

15.3 PADD.(U/S)8.S -- 8-bit Parallel (Un/Signed) Addition with Saturation

instruction

Unified	
instruction	syntax padd.u8.s rz, rx,ry padd.s8.s rz, rx,ry
Operation	Rz[31:24] = Saturate(Rx[31:24] + Ry[31:24]) Rz[23:16] = Saturate(Rx[23:16] + Ry[23:16]) Rz[15:8] = Saturate(Rx[15:8] + Ry[15:8]) Rz[7:0] = Saturate(Rx[7:0] + Ry[7:0]) Only 32-bit
Compilation results	instructions exist. padd.u8.s rz, rx,ry padd.s8.s rz, rx,ry

	In bytes, add the 4 bytes of Rx to the 4 bytes of Ry respectively, and store the addition result in Rz after saturation processing. For unsigned numbers, the saturation processing process is as follows: if the 8-bit addition result is greater than the saturation value 0xFF, the result is 0xFF, otherwise the addition result itself is used. For signed numbers, the saturation processing process is as follows: if the 8-bit addition result is greater than the upper saturation value 0x7F, the result is 0x7F, if the 8-bit addition result is less than the lower saturation value 0x80, the result is 0x80, otherwise the addition result itself
	is used. No impact
	none
	Affected flag: Abnormal:
32 Bit	
	Rz[31:24] = Saturate(Rx[31:24] + Ry[31:24]) Rz[23:16] = Saturate(Rx[23:16] + Ry[23:16]) Rz[15:8] = Saturate(Rx[15:8] + Ry[15:8]) Rz[7:0] = Saturate(Rx[7:0] + Ry[7:0]) padd.u8.s rz, rx, ry
	padd.s8.s rz, rx, ry In bytes, add the 4 bytes of Rx to the
Grammar bright:	Say bytes of Ry respectively, and store the addition result in Rz after saturation processing. For unsigned numbers, the saturation processing process is as follows: if the 8-bit addition result is greater than the saturation value 0xFF, the result is 0xFF, otherwise the addition result itself is taken. For signed numbers, the saturation processing process is as follows: if the 8-bit addition result is greater than the upper saturation value 0x7F, the result is 0x7F, if the 8-bit addition result is less than the lower saturation value 0x80, the result is 0x80, otherwise the addition result itself is taken. No effect
	none
	Affected flag: Abnormal:

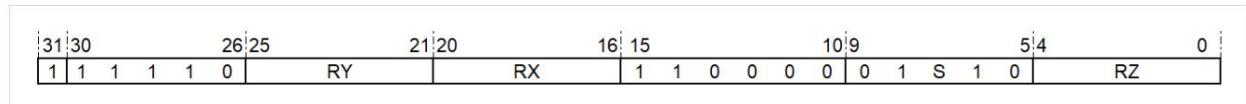
Instruction format:

Figure 15.3: PADD.(US)8.S

15.4 PADD.(U/S)16.S – 16-bit Parallel (Un/With) Signed Add with Saturation**Legal Directive**

Unified instructions	
grammar	padd.u16.s rz, rx,ry padd.s16.s rz, rx,ry
operate	Rz[31:16] = Saturate(Rx[31:16] + Ry[31:16]) Rz[15:0] = Saturate(Rx[15:0] + Ry[15:0])
The compiled result	contains only 32-bit instructions. padd.u16.s rz, rx,ry padd.s16.s rz, rx,ry

	<p>illustrate : Add the upper and lower half words of Rx and Ry respectively, and store the addition result in Rz after saturation processing. For unsigned numbers, the saturation processing process is that if the 16-bit addition result is greater than the saturation value 0xFFFF, the result is 0xFFFF, otherwise the addition result itself is taken. For signed numbers, the saturation processing process is that if the 16-bit addition result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the 16-bit addition result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the addition result itself is taken. No impact</p>
	<p>none</p> <p>Affected flag: Abnormal:</p>
32 Bit	
	<p>Rz[31:16] = Saturate(Rx[31:16] + Ry[31:16]) Rz[15:0] = Saturate(Rx[15:0] + Ry[15:0]) padd.u16.s rz, rx, ry padd.s16.s</p> <p>instruction operation: the upper and lower Law: halfwords of Rx and Ry respectively,</p> <p>illustrate : add the addition result in Rz after saturation processing. For unsigned numbers, the saturation processing process is as follows: if the 16-bit addition result is greater than the saturation value 0xFFFF, the result is 0xFFFF, otherwise the addition result itself is taken. For signed numbers, the saturation processing process is as follows: if the 16-bit addition result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the 16-bit addition result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the addition result itself is taken. No effect</p>
	<p>none</p>
	<p>Affected flags: Exception: Instruction format :</p>

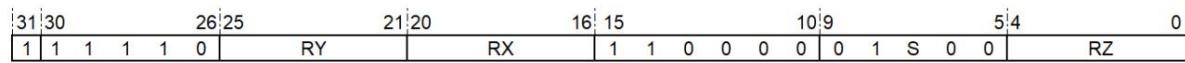


Figure 15.4: PADD.(US)16.S

15.5 ADD.(U/S)32.S -- 32-bit (unsigned/signed) addition with saturation

Unified instructions		
	operate	Compilation results
Syntax add.u32.s rz, rx,ry add.s32.s rz, rx,ry	Rz[31:0] = Saturate(Rx[31:0] + Ry[31:0])	Only 32-bit instructions exist. add.u32.s rz, rx,ry add.s32.s rz, rx,ry

	Rx is added to Ry, and the result of the addition is saturated and stored in Rz. For Illustrate : unsigned numbers, the saturation process is as follows: if the 32-bit addition result is greater than the upper saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, otherwise the addition result itself is used. For signed numbers, the saturation process is as follows: if the 32-bit addition result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the 32-bit addition result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the addition result itself is used. No effect
	none
	Affected flag: Abnormal:
32	
Bit	
	Rz[31:0] = Saturate(Rx[31:0] + Ry[31:0])
	add.u32.s rz, rx, ry add.s32.s rz, rx, ry Add Rx and Ry, and store
	the result of addition in Rz after saturation processing. For unsigned numbers, the Instruction operation steps description is as follows: if the 32-bit addition result is greater than the upper saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, otherwise the addition result itself is used. For signed numbers, the saturation processing process is as follows: if the 32-bit addition result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the 32-bit addition result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the addition result itself is used. No effect
	none
	Affected flag: Abnormal:

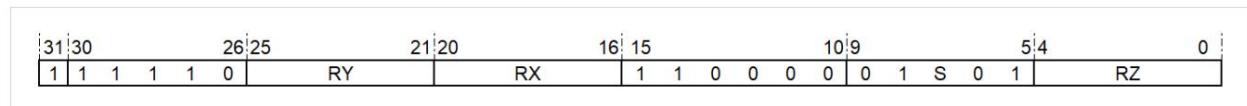
Instruction format:

Figure 15.5: ADD.(US)32.S

15.6 PSUB.8 - 8-bit Parallel Subtraction Instruction

Unified	
instruction syntax	psub.8 rz, rx, ry
operation	<p>Rz[31:24] = Rx[31:24] - Ry[31:24]</p> <p>Rz[23:16] = Rx[23:16] - Ry[23:16]</p> <p>Rz[15:8] = Rx[15:8] - Ry[15:8]</p> <p>Rz[7:0] = Rx[7:0] - Ry[7:0] Only exists</p>
Compilation results	for 32-bit instructions. psub.8 rz, rx, ry

Description:	In byte units, subtract the 4 bytes of Rx from the 4 bytes of Ry, and store the subtraction result in Rz. Affected flags:
Exception:	No impact
No 32-bit instruction	
operation:	
	Rz[31:24] = Rx[31:24] - Ry[31:24]
	Rz[23:16] = Rx[23:16] - Ry[23:16]
	Rz[15:8] = Rx[15:8] - Ry[15:8]
	Rz[7:0] = Rx[7:0] - Ry[7:0]
Syntax:	psub.8 rz, rx, ry Description:
	In byte units, subtract the 4 bytes of Rx from the 4 bytes of Ry, and store the subtraction result in Rz. Affected flags: Exception: None
	No impact

Instruction format:

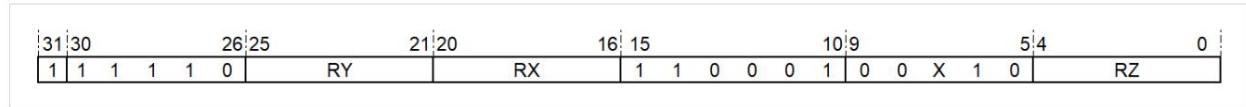


Figure 15.6: PSUB.8

15.7 PSUB.16 - 16-bit Parallel Subtraction Instruction

Unified instructions	
Syntax	psub.16 rz, rx,ry
Operations	Rz[31:16] = Rx[31:16] - Ry[31:16] Rz[15:0] = Rx[15:0] - Ry[15:0]
The compiled result	contains only 32-bit instructions. psub.16 rz, rx,ry

	Subtract the high and low halfwords of Rx from the high and low halfwords of Ry respectively, and store the result of the subtraction in Rz.
Description: Impact flag:	No impact
Exception:	none
32-bit instructions	
operate:	Rz[31:16] = Rx[31:16] - Ry[31:16] Rz[15:0] = Rx[15:0] - Ry[15:0]
Syntax:	psub.16 rz, rx,ry
	Subtract the high and low halfwords of Rx from the high and low halfwords of Ry respectively, and store the result of the subtraction in Rz.
Description: Impact flag:	No impact
Abnormal:	None

Instruction format:

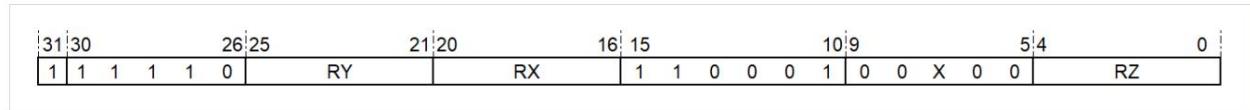


Figure 15.7: PSUB.16

15.8 PSUB.(U/S)8.S – 8-bit Parallel (Un/Signed) Subtraction with Saturation

make

Unified	
instruction	syntax psub.u8.s rz, rx,ry psub.s8.s rz, rx,ry
Operation	Rz[31:24] = Saturate(Rx[31:24] - Ry[31:24]) Rz[23:16] = Saturate(Rx[23:16] - Ry[23:16]) Rz[15:8] = Saturate(Rx[15:8] - Ry[15:8]) Rz[7:0] = Saturate(Rx[7:0] - Ry[7:0]) Only 32-bit
Compilation results	instructions exist. psub.u8.s rz, rx,ry psub.s8.s rz, rx,ry

	<p>In bytes, the 4 bytes of Rx are subtracted from the 4 bytes of Ry, and the subtraction result is saturated and stored in Rz. For unsigned numbers, the saturation process is: if the 8-bit subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the subtraction result itself is used.</p> <p>For signed numbers, the saturation process is: if the 8-bit subtraction result is greater than the upper saturation value 0x7F, the result is 0x7F, if the 8-bit subtraction result is less than the lower saturation value 0x80, the result is 0x80, otherwise the subtraction result itself</p>
	is used. No impact
	none
	Affected flag: Abnormal:
32 Bit	
	<p>Rz[31:24] = Saturate(Rx[31:24] - Ry[31:24])</p> <p>instruction op Rz[23:16] = Saturate(Rx[23:16] - Ry[23:16])</p> <p>Rz[15:8] = Saturate(Rx[15:8] - Ry[15:8])</p> <p>Rz[7:0] = Saturate(Rx[7:0] - Ry[7:0]) psub.u8.s rz, rx,ry</p>
	<p>psub.s8.s rz, rx,ry In byte</p> <p>units, the 4 bytes of Rx are</p>
Grammar bright:	<p>Subtracted from the 4 bytes of Ry, and the subtraction result is saturated and stored in Rz. For unsigned numbers, the saturation process is as follows: if the 8-bit subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the subtraction result itself is used. For signed numbers, the saturation process is as follows: if the 8-bit subtraction result is greater than the upper saturation value 0x7F, the result is 0x7F, if the 8-bit subtraction result is less than the lower saturation value 0x80, the result is 0x80, otherwise the subtraction result itself</p>
	is used. No effect
	none
	Affected flag: Abnormal:

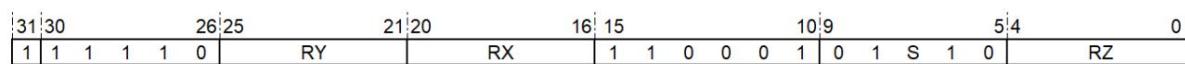
Instruction format:

Figure 15.8: PSUB.(US)8.S

15.9 PSUB.(U/S)16.S – 16-bit Parallel (Un/Signed) Subtraction with Saturation

instruction

Unified instructions	
grammar	psub.u16.s rz, rx,ry psub.s16.s rz, rx,ry
operate	Rz[31:16] = Saturate(Rx[31:16] - Ry[31:16]) Rz[15:0] = Saturate(Rx[15:0] - Ry[15:0])
The compiled result	contains only 32-bit instructions. psub.u16.s rz, rx,ry psub.s16.s rz, rx,ry

illustrate	Subtract the upper and lower halfwords of Rx and Ry respectively, and store the subtraction result in Rz after saturation processing. For unsigned numbers, the saturation processing process is as follows: if the 16-bit subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the subtraction result itself is taken. For signed numbers, the saturation processing process is as follows: if the 16-bit subtraction result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the 16-bit subtraction result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the subtraction result itself is
	taken. No effect
	none
Affected flag: Abnormal:	
32 Bit	
	Rz[31:16] = Saturate(Rx[31:16] - Ry[31:16]) Rz[15:0] = Saturate(Rx[15:0] - Ry[15:0]) psub.u16.s rz, rx, ry psub.s16.s
instruction operation	subtract the upper and lower halfwords of Rx and Ry
Law:	respectively, and store the subtraction result in Rz after saturation processing. For unsigned numbers, the saturation processing process is as follows: if the 16-bit subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the subtraction result itself is taken. For signed numbers, the saturation processing process is as follows: if the 16-bit subtraction result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the 16-bit subtraction result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the subtraction result itself is taken. No effect
	none
Affected flag: Abnormal:	

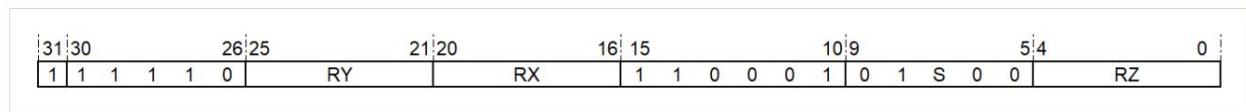
Instruction format:

Figure 15.9: PSUB.US16.S

15.10 SUB.(U/S)32.S – 32-bit (un/signed) subtraction with saturation

Unified command	
syntax	sub.u32.s rz, rx, ry sub.s32.s rz, rx, ry The operation
	Rz[31:0] = Saturate(Rx[31:0] - Ry[31:0]) is compiled to produce only 32-bit instructions.
	sub.u32.s rz, rx, ry sub.s32.s rz, rx, ry

	<p>Rx is subtracted from Ry, and the result of the subtraction is saturated and stored</p> <p>illustrate : in Rz. For unsigned numbers, the saturation process is as follows: if the 32-bit subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the subtraction result itself is used. For signed numbers, the saturation process is as follows: if the 32-bit subtraction result is greater than the upper saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, if the 32-bit subtraction result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the subtraction result itself is</p>
	used. No effect
	none

Affected flag: Abnormal:

32	Bit
	Rz[31:0] = Saturate(Rx[31:0] - Ry[31:0])
	sub.u32.s rz, rx, ry sub.s32.s rz, rx, ry Subtract Rx from Ry,
	and store the subtraction result in Rz after saturation processing. For unsigned instruction opmembers S the saturation processing process is as follows: if the 32-bit subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the subtraction result itself is used. For signed numbers, the saturation processing process is as follows: if the 32-bit subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the 32-bit subtraction result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the
	subtraction result itself is used. No effect
	none Affected flag: Abnormal:

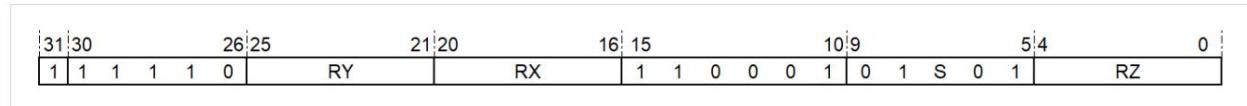
Instruction format:

Figure 15.10: SUB.(US)32.S

15.11 PADDH.(U/S)8——8-bit parallel (unsigned/signed) addition and averaging instruction

Unified	
command syntax	paddh.u8 rz, rx,ry paddh.s8 rz, rx,ry
Operation	Rz[31:24] = (Rx[31:24] + Ry[31:24])/2 Rz[23:16] = (Rx[23:16] + Ry[23:16])/2 Rz[15:8] = (Rx[15:8] + Ry[15:8])/2 Rz[7:0] = (Rx[7:0] + Ry[7:0])/2 Only 32-bit instructions exist. paddh.u8 rz, rx,ry
Compilation results	paddh.s8 rz, rx,ry

illustrate:	In byte units, add the 4 bytes of Rx to the 4 bytes of Ry, divide the result by 2 and store it in Rz.
Affected flag:	No impact
Abnormal:	none

32-bit	
instruction operation:	Rz[31:24] = (Rx[31:24] + Ry[31:24])/2 Rz[23:16] = (Rx[23:16] + Ry[23:16])/2 Rz[15:8] = (Rx[15:8] + Ry[15:8])/2 Rz[7:0] = (Rx[7:0] + Ry[7:0])/2
Syntax:	paddh.u8 rz, rx,ry paddh.s8 rz, rx,ry
Description:	In byte units, add the 4 bytes of Rx to the 4 bytes of Ry, divide the result by 2 and store it in Rz. Affected flags: Exception: None
	No impact

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	RY	RX	1 1 0 0 0 0 1 0 S 1 0	RZ		

Figure 15.11: PADDH.(US)8

15.12 PADDH.(U/S)16——16-bit parallel (unsigned/signed) addition average value

make

Unified instructions	
grammar	paddh.u16 rz, rx, ry paddh.s16 rz, rx, ry
operate	Rz[31:16] = (Rx[31:16] + Ry[31:16])/2 Rz[15:0] = (Rx[15:0] + Ry[15:0])/2
The compiled result	contains only 32-bit instructions. paddh.u16 rz, rx, ry paddh.s16 rz, rx, ry

Description:	Add the upper and lower halfwords of Rx to the upper and lower halfwords of Ry respectively, divide the result by 2 and store it in Rz.
Impact flag:	No impact
Abnormal:	None

32-bit instructions	
operate:	Rz[31:16] = (Rx[31:16] + Ry[31:16])/2 Rz[15:0] = (Rx[15:0] + Ry[15:0])/2
grammar:	paddh.u16 rz, rx, ry paddh.s16 rz, rx, ry
	Add the high and low halfwords of Rx to the high and low halfwords of Ry respectively, divide the result by 2 and store it in Rz.
Description: Impact flag:	No impact
abnormal:	none

Instruction format:

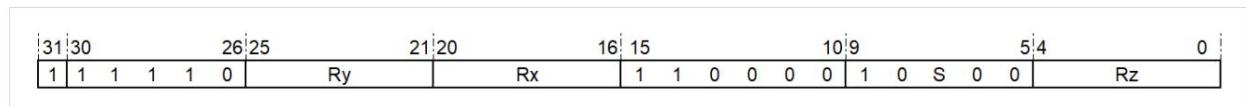


Figure 15.12: PADDH.(US)16

15.13 ADDH.(U/S)32——32-bit (unsigned/signed) addition and averaging instruction

Unified instructions	
grammar	addh.u32 rz, rx,ry addh.s32 rz, rx,ry
The	Rz[31:0] = (Rx[31:0] + Ry[31:0])/2
result of the operation compilation	is only 32-bit instructions. addh.u32 rz, rx,ry addh.s32 rz, rx,ry

	Rx is added to Ry, and the result is divided by 2 and stored in Rz.
Description: Impact flag:	No impact
abnormal:	none

32-bit instructions	
	Rz[31:0] = (Rx[31:0] + Ry[31:0])/2
Operation: Syntax:	addh.u32 rz, rx,ry addh.s32 rz, rx,ry
Description:	Add Rx and Ry, divide the result by 2 and store it in Rz.
Impact flag:	No impact
Abnormal:	None

Instruction format:

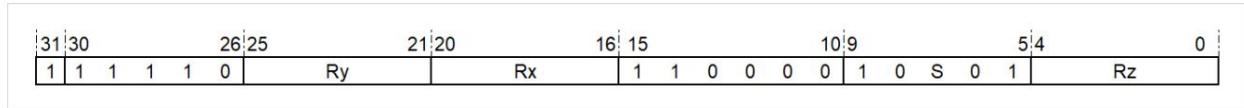


Figure 15.13: ADDH.(US)32

15.14 PSUBH.(U/S)8——8-bit parallel (unsigned/signed) subtraction average instruction

Unified	
Instruction Syntax	psubh.u8 rz, rx,ry psubh.s8 rz, rx,ry
Operations	$Rz[31:24] = (Rx[31:24] - Ry[31:24])/2$ $Rz[23:16] = (Rx[23:16] - Ry[23:16])/2$ $Rz[15:8] = (Rx[15:8] - Ry[15:8])/2$ $Rz[7:0] = (Rx[7:0] - Ry[7:0])/2$ Only 32-bit instructions exist. psubh.u8 rz, rx,ry psubh.s8 rz, rx,ry
Compilation results	

illustrate:	In byte units, subtract the 4 bytes of Rx from the 4 bytes of Ry, divide the subtraction result by 2 and store it in Rz.
Affected flag:	No impact
Abnormal:	none

32-bit	
instruction operation:	$Rz[31:24] = (Rx[31:24] - Ry[31:24])/2$ $Rz[23:16] = (Rx[23:16] - Ry[23:16])/2$ $Rz[15:8] = (Rx[15:8] - Ry[15:8])/2$ $Rz[7:0] = (Rx[7:0] - Ry[7:0])/2$
Syntax:	psubh.u8 rz, rx,ry psubh.s8 rz, rx,ry
Description:	In byte units, subtract the 4 bytes of Rx from the 4 bytes of Ry, divide the subtraction result by 2 and store it in Rz. Affected flags:
Exception:	No impact
None	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 0	1 1 0 S 1 0	Rz	

Figure 15.14: PSUBH.(US)8

15.15 PSUBH.(U/S)16——16-bit parallel (unsigned/signed) subtraction average instruction

Unified instructions	
grammar	psubh.u16 rz, rx,ry psubh.s16 rz, rx,ry
operate	$Rz[31:16] = (Rx[31:16] - Ry[31:16])/2$ $Rz[15:0] = (Rx[15:0] - Ry[15:0])/2$
The compiled result	contains only 32-bit instructions. psubh.u16 rz, rx,ry psubh.s16 rz, rx,ry

	Subtract the high and low halfwords of Rx from the high and low halfwords of Ry respectively, divide the result by 2 and store it in Rz.
Description: Impact flag:	No impact
Abnormal:	None

32-bit instructions	
operate:	$Rz[31:16] = (Rx[31:16] - Ry[31:16])/2$ $Rz[15:0] = (Rx[15:0] - Ry[15:0])/2$
grammar:	psubh.u16 rz, rx,ry psubh.s16 rz, rx,ry
Explanation:	Subtract the high and low halfwords of Rx from the high and low halfwords of Ry respectively, divide the subtraction result by 2 and store it in Rz.
Impact flag:	No impact
Abnormal:	None

Instruction format:

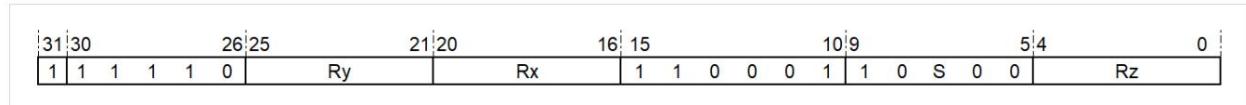


Figure 15.15: PSUBH.(US)16

15.16 SUBH.(U/S)32——32-bit (unsigned/signed) subtraction average instruction

Unified instructions	
grammar	subh.u32 rz, rx,ry subh.s32 rz, rx,ry
The	Rz[31:0] = (Rx[31:0] - Ry[31:0])/2
result of the operation compilation	is only 32-bit instructions. subh.u32 rz, rx,ry subh.s32 rz, rx,ry

	Rx is subtracted from Ry, and the result of the subtraction is divided by 2 and stored in Rz.
Description: Impact flag:	No impact
abnormal:	none

32-bit instructions	
	Rz[31:0] = (Rx[31:0] - Ry[31:0])/2
Operation: Syntax:	subh.u32 rz, rx,ry subh.s32 rz, rx,ry
Explanation:	Rx is subtracted from Ry, and the result of the subtraction is divided by 2 and stored in Rz.
Impact flag:	No impact
Abnormal:	None

Instruction format:

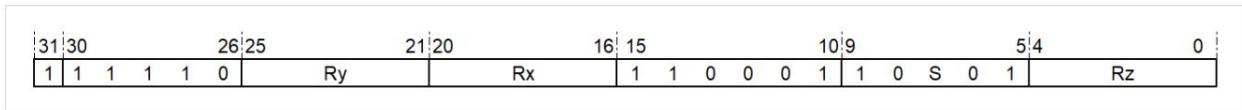


Figure 15.16: SUBH.(US)32

15.17 PASX.16——16-bit parallel cross addition and subtraction instructions

Unified instructions	
Syntax	pasx.16 rz, rx,ry
Operations	Rz[31:16] = Rx[31:16] + Ry[15:0] Rz[15:0] = Rx[15:0] - Ry[31:16]
The compiled result	contains only 32-bit instructions. pasx.16 rz, rx,ry

Description:	Add the high 16 bits of Rx to the low 16 bits of Ry, and store the result in the high 16 bits of Rz; Subtract the low 16 bits of Rx from the high 16 bits of Ry, and store the result in the low 16 bits of Rz;
Affected flag:	Rz. No effect
Abnormal:	None

32-bit	
instruction operation:	$Rz[31:16] = Rx[31:16] + Ry[15:0]$ $Rz[15:0] = Rx[15:0] - Ry[31:16]$
Syntax:	psax.16 rz, rx,ry
Description:	Add the high 16 bits of Rx to the low 16 bits of Ry, and store the result in the high 16 bits of Rz; Subtract the low 16 bits of Rx from the high 16 bits of Ry, and store the result in the low 16 bits of Rz;
Affected flag:	Rz. No effect
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 1 0 0 1 0	0 0 x 1 1	Rz	

Figure 15.17: PASX.16

15.18 PSAX.16——16-bit parallel cross add and subtract instruction

Unified command	
syntax	psax.16 rz, rx,ry
operation	$Rz[31:16] = Rx[31:16] - Ry[15:0]$ $Rz[15:0] = Rx[15:0] + Ry[31:16]$
The compiled result	contains only 32-bit instructions. psax.16 rz, rx,ry

Description:	Subtract the upper 16 bits of Rx from the lower 16 bits of Ry, and store the result in the upper 16 bits of Rz; Add the lower 16 bits of Rx to the upper 16 bits of Ry, and store the result in the lower 16 bits of Rz;
Affected flag:	Rz. No effect
Abnormal:	None

32-bit	
instruction operation: Rz[31:16] = Rx[31:16] - Ry[15:0]	Rz[15:0] = Rx[15:0] + Ry[31:16] Syntax: psax.16
rz, rx, ry Description: Subtract the upper 16 bits of Rx from the lower 16 bits of Ry, and store the result in the upper 16 bits of Rz; Add the lower 16 bits of Rx to the upper 16 bits of Ry, and store the result in the lower 16 bits of Rz. No effect	
Affected flag:	
Abnormal: None	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0 Ry Rx 1 1 0 0 1 1 0 0 x 1 1 Rz						

Figure 15.18: PSAX.16

15.19 PASX.(U/S)16.S – 16-bit parallel (without/with) signed interleaving with saturation

Fork addition and subtraction instructions

Unified command	
syntax	pasx.u16.s rz, rx, ry pasx.s16.s rz, rx, ry Rz[31:16] =
operate	saturate(Rx[31:16] + Ry[15:0]) Rz[15:0] = saturate(Rx[15:0] - Ry[31:16]) The compiled result
only has 32-bit instructions.	pasx.u16.s rz, rx, ry pasx.s16.s rz, rx, ry

	<p>Illustrate : Rx are subtracted from the upper 16 bits of Ry, and the result is saturated and stored in the lower 16 bits of Rz; the lower 16 bits of saturation process is as follows: if the 16-bit addition/subtraction result is greater than the upper saturation value 0xFFFF, the result is 0xFFFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x0, the result is 0x0; otherwise, the result itself is used. For signed numbers, the saturation process is as follows: if the 16-bit addition/subtraction result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x8000, the result is 0x8000; otherwise, the result itself is used. No impact</p>
	<p>Affected flag: Abnormal: none</p>

32 Bit	
	<p>Rz[31:16] = saturate(Rx[31:16] + Ry[15:0]) Rz[15:0] = saturate(Rx[15:0] - Ry[31:16]) pasx.u16.s rz, rx, ry pasx.s16.s rz,</p>
	<p>rx,ry Add the upper 16 bits of Rx to the lower 16 bits of Ry, and store</p>
instruction operation: Syntax	<p>upper 16 bits of Rz after saturation processing: Subtract the lower 16 bits of Rx from the upper 16 bits of Ry, and store the result in the lower 16 bits of Rz after saturation processing. For unsigned numbers, the saturation processing process is as follows: if the 16-bit addition/subtraction result is greater than the upper saturation value 0xFFFF, the result is 0xFFFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x0, the result is 0x0; otherwise, the result itself is used. For signed numbers, the saturation process is: if the 16-bit addition/subtraction result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x8000, the result is 0x8000; otherwise, the result itself is used. No effect</p>
	<p>Affected flag: Abnormal: none</p>

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 1 0	0 1 S 1 1	Rz	

Figure 15.19: PASX.(US)16.S

15.20 PSAX.(U/S)16.S – 16-bit Parallel (Un/Yes) Signed Interleave with Saturation

Cross subtraction and addition instructions

Unified command	
syntax	psax.u16.s rz, rx,ry psax.s16.s rz, rx,ry Rz[31:16] =
operate	saturate(Rx[31:16] - Ry[15:0]) Rz[15:0] = saturate(Rx[15:0] + Ry[31:16]) The compiled result only
has 32-bit instructions.	psax.u16.s rz, rx,ry psax.s16.s rz, rx,ry

explain bright:	The upper 16 bits of Rx are subtracted from the lower 16 bits of Ry, and the result is saturated and stored in the upper 16 bits of Rz; the lower 16 bits of Rx are added to the upper 16 bits of Ry, and the result is saturated and stored in the lower 16 bits of Rz. For unsigned numbers, the saturation process is as follows: if the 16-bit addition/subtraction result is greater than the upper saturation value 0xFFFF, the result is 0xFFFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x0, the result is 0x0; otherwise, the result itself is used. For signed numbers, the saturation process is as follows: if the 16-bit addition/subtraction result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x8000, the result is 0x8000; otherwise, the result itself is used. No effect
Affected flag: Abnormal:	none

32 Bit	
	Rz[31:16] = saturate(Rx[31:16] - Ry[15:0]) Rz[15:0] = saturate(Rx[15:0] + Ry[31:16])
	psax.u16.s rz, rx, ry psax.s16.s rz, rx, ry For unsigned
	numbers, the saturation process is: if the 16-bit addition/subtraction result is greater than the upper saturation value 0xFFFF, the result is 0xFFFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x0, the result is 0x0; otherwise, the result itself is used. For signed numbers, the saturation process is: if the 16-bit addition/subtraction result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF; if the 16-bit addition/subtraction result is less than the lower saturation value 0x8000, the result is 0x8000; otherwise, the result itself is used. No effect
Impact Mark Zhi Bit:	
	none
Exception:	

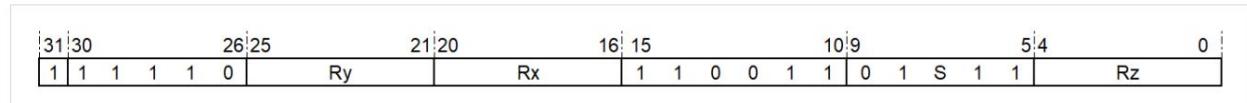
Instruction format:

Figure 15.20: PSAX.(US)16.S

15.21 PASXH.(U/S)16——16-bit parallel (without/with) sign cross addition and subtraction average value instruction

Unified command	
syntax	pasxh.u16 rz, rx, ry pasxh.s16 rz, rx, ry Rz[31:16] = (Rx[31:16]
operate	+ Ry[15:0])/2 Rz[15:0] = (Rx[15:0] - Ry[31:16])/2 The compiled result only contains 32-bit instructions. pasxh.u16 rz, rx, ry
pasxh.s16 rz, rx, ry	

Description:	Add the upper 16 bits of Rx to the lower 16 bits of Ry, divide the result by 2 and store it in the upper 16 bits of Rz; Subtract the lower 16 bits of Rx from the upper 16 bits of Ry, divide the result by 2 and store it in the lower 16 bits
Affected flag:	of Rz. No effect
Abnormal:	None

32-bit	
instruction operation:	Rz[31:16] = (Rx[31:16] + Ry[15:0])/2 Rz[15:0] = (Rx[15:0] - Ry[31:16])/2 Syntax: pasxh.u16 rz, rx, ry
pasxh.s16 rz, rx, ry Description:	Add the upper 16 bits of Rx to the lower 16 bits of Ry, divide the result by 2 and store it in the upper 16 bits of Rz; subtract the lower 16 bits of Rx from the upper 16 bits of Ry, divide the result by 2 and store it in the lower 16 bits of Rz. No effect
Affected flag:	
Abnormal:	None

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0					
1	1	1	1	1	0	Ry	Rx	1	1	0	0	1	0	S	1	1	Rz

Figure 15.21: PASXH.(US)16

15.22 PSAXH.(U/S)16——16-bit parallel (without/with) sign cross subtraction and addition average instruction

Unified command	
syntax	psaxh.u16 rz, rx, ry psaxh.s16 rz, rx, ry Rz[31:16] = (Rx[31:16]
operate	- Ry[15:0])/2 Rz[15:0] = (Rx[15:0] + Ry[31:16])/2 The compiled result only contains 32-bit instructions.
	psaxh.u16 rz, rx, ry psaxh.s16 rz, rx, ry

Explanation:	Subtract the upper 16 bits of Rx from the lower 16 bits of Ry, divide the result by 2 and store it in the upper 16 bits of Rz; Subtract the upper 16 bits of Ry from the lower 16 bits of Rx, add the result by 2 and store it in the lower 16 bits of Rz.
Affected flag:	of Rz. No effect
Abnormal:	None

32-bit	
instruction operation:	Rz[31:16] = (Rx[31:16] - Ry[15:0])/2 Rz[15:0] = (Rx[15:0] + Ry[31:16])/2
Syntax:	psaxh.u16 rz,rx,ry psaxh.s16 rz, rx,ry
Description:	Subtract the upper 16 bits of Rx from the lower 16 bits of Ry, divide the result by 2 and store it in the upper 16 bits of Rz; Add the lower 16 bits of Rx to the upper 16 bits of Ry, divide the result by 2 and store it in the lower 16 bits of Rz.
Affected flag:	Rz. No effect
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 1 0 0 1 1 0 S 1 1	Rz		

Figure 15.22: PSAXH.(US)16

15.23 ADD.64——64-bit addition instruction

The unified	
instruction syntax	add.64 rz, rx,ry
operation	{Rz+1[31:0], Rz[31:0]} = {Rx+1[31:0], Rx[31:0]} + {Ry+1[31:0], Ry[31:0]} compilation result only has 32-bit
instructions.	add.64 rz, rx,ry

Illustrate :	Rx+1 and Rx form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), Ry+1 and Ry form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), perform 64-bit addition, store the upper 32 bits of the addition result in Rz+1, and store the lower 32 bits in Rz. No impact
Affected flag: Abnormal:	none

32 Bit	
	$\{Rz+1[31:0], Rz[31:0]\} = \{Rx+1[31:0], Rx[31:0\} + \{Ry+1[31:0], Ry[31:0\}$
	add.64 rz, rx, ry
instruction operation mode: when Rx+1 is the upper 32 bits and Rx is the lower 32 bits,	Rx+1 and Rx form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), Ry+1 and Ry form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), perform 64-bit addition, store the upper 32 bits of the addition result in Rz+1, and store the lower 32 bits in Rz. No impact
Affected flag: Abnormal:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 0 0	0 0 x 1 1	Rz	

Figure 15.23: ADD.64

15.24 ADD.(U/S)64.S -- 64-bit (unsigned/signed) addition with saturation

Unified command	
syntax	add.u64.s rz, rx,ry add.s64.s rz, rx,ry {Rz+1[31:0]},
The	Rz[31:0] = Saturate((Rx+1[31:0], Rx[31:0]) + (Ry+1[31:0], Ry[31:0]))
compiled result	contains only 32-bit instructions. add.u64.s rz, rx,ry add.s64.s rz, rx,ry

32 Bit	
	{Rz+1[31:0], Rz[31:0]} = Saturate((Rx+1[31:0], Rx[31:0]) + (Ry+1[31:0], Ry[31:0]))
	add.u64.s rz, rx,ry add.s64.s rz, rx,ry Rx+1 and Rx form
instruction operation system: bright:	where Rx+1 is the upper 32 bits and Rx is the lower 32 bits, Ry+1 and Ry form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), perform 64-bit addition, and after the addition result is saturated, the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz. For unsigned numbers, the saturation process is that if the addition result is greater than the saturation value 0xFFFF FFFF FFFF FFFF, the result is 0xFFFF FFFF FFFF FFFF, otherwise the result itself is used. For signed numbers, the saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF FFFF FFFF, the result is 0x7FFF FFFF FFFF FFFF; if the addition result is less than the lower saturation value 0x8000 0000 0000 0000, the result is 0x8000 0000 0000 0000; otherwise, the result itself is used. No impact
	none
Affected flag: Abnormal:	

Instruction format:

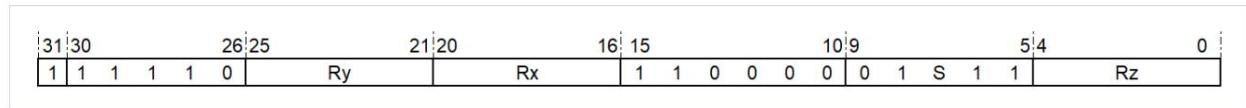


Figure 15.24: ADD.(U/S)64.S

15.25 SUB.64——64-bit subtraction instruction

The unified instruction syntax	sub.64 rz, rx, ry
operation	{Rz+1[31:0], Rz[31:0]} = {Rx+1[31:0], Rx[31:0]} - {Ry+1[31:0], Ry[31:0]} compilation result has only 32-bit instructions.
	sub.64 rz, rx, ry

Illustrate :	Rx+1 and Rx form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), Ry+1 and Ry form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), perform 64-bit subtraction, and store the upper 32 bits of the subtraction result in Rz+1 and the lower 32 bits in Rz. No impact
Affected flag:	Abnormal: none

32 Bit	
	{Rz+1[31:0], Rz[31:0]} = {Rx+1[31:0], Rx[31:0]} - {Ry+1[31:0], Ry[31:0]}
	sub.64 rz, rx, ry
Instruction operation:	Rx+1 and Rx form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), Ry+1 and Ry form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), perform 64-bit subtraction, and store the upper 32 bits of the subtraction result in Rz+1 and the lower 32 bits in Rz. No impact
Affected flag:	Abnormal: none

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	1	1	1	1	0	Ry	Rx	1	1	0	0	x

Figure 15.25: SUB.64

15.26 SUB.(U/S)64.S -- 64-bit (un/signed) subtraction with saturation

Unified command	
syntax	sub.u64.s rz, rx, ry sub.s64.s rz, rx, ry {Rz+1[31:0],
The	Rz[31:0] = Saturate({Rx+1[31:0], Rx[31:0]} - {Ry+1[31:0], Ry[31:0]})
result of the operation compilation	is only 32-bit instructions. sub.u64.s rz, rx, ry sub.s64.s rz, rx, ry

32 Bit	
	{Rz+1[31:0], Rz[31:0]} = Saturate({Rx+1[31:0], Rx[31:0]} - {Ry+1[31:0], Ry[31:0]})
	sub.u64.s rz, rx, ry sub.s64.s rz, rx, ry Rx+1 and Rx form
instruction operation system bright:	where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), Ry+1 and Ry form a 64-bit number (where Rx+1 is the upper 32 bits and Rx is the lower 32 bits), perform 64-bit subtraction, and after the subtraction result is saturated, the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz. For unsigned numbers, the saturation process is that if the subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise the result itself is used. For signed numbers, the saturation process is as follows: if the subtraction result is greater than the upper saturation value 0xFFFF FFFF FFFF FFFF, the result is 0xFFFF FFFF FFFF FFFF; if the subtraction result is less than the lower saturation value 0x8000 0000 0000 0000, the result is 0x8000 0000 0000 0000; otherwise, the result itself is used. No impact
Affected flag: Abnormal:	none

Instruction format:

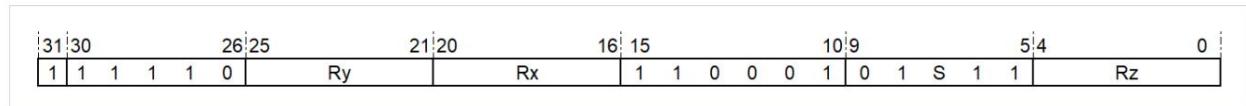


Figure 15.26: SUB.(US)64.S

15.27 ASRI.S32.R -- Immediate Arithmetic Shift Right with Rounding

The unified	
	asri.s32.r rz, rx, oimm5 Rz[31:0] <-
	Round(Rx[31:0] >>>oimm[4:0])
instruction syntax operation compilation result	only has 32-bit instructions. asri.s32.r rz, rx, oimm5

explain bright:	Perform arithmetic right shift on the value of Rx (shift the original value to the right and shift a copy of the original sign bit to the left), and perform rounding. The rounding operation is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset
Affected flags:	1. No effect
	The range of immediate value is 1-32.
	none
Restrictions: Exceptions:	

32 Bit	
	Rz[31:0] <- Round(Rx[31:0] >>>oimm[4:0])
	asri.s32.r rz, rx, oimm5
instruction syntax operation description	Perform arithmetic right shift on the value of Rx (shift the original value to the right and shift a copy of the original sign bit to the left), and perform rounding. The rounding operation is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with offset
Affected flags:	1. No effect
	The range of immediate value is 1-32
	none
Restrictions: Exceptions:	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Oimm5	Rx	1 1 0 1 0 0	0 1 1 0 1	Rz	

Figure 15.27: ASR.S32.R

15.28 ASR.S32.R — Arithmetic Right Shift with Rounding

The unified	
instruction syntax	asr.s32.r rz, rx, ry operation
	Rz[31:0] <- Round(Rx[31:0] >>> Ry[5:0]) only has 32-bit instructions in the compiled result .
	asr.s32.r rz, rx, ry

Description:	Perform arithmetic right shift on the value of Rx (the original value is shifted right, and a copy of the original sign bit is shifted in from the left side), and perform rounding operation. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by Ry[5:0]. No effect
Affected flag:	
Abnormal:	None

32-bit	
instruction operation:	Rz[31:0] <- Round(Rx[31:0] >>> Ry[5:0]) Syntax: asr.s32.r
rz, rx, ry Description:	Perform arithmetic right shift on the value of Rx (the original value is shifted right, and a copy of the original sign bit is shifted in from the left), and round the value. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by Ry[5:0]. No effect
Affected flag:	
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 0 0	0 1 1 1 1	Rz	

Figure 15.28: ASR.S32.R

15.29 LSRI.U32.R – Immediate Logical Shift Right Instruction with Rounding

The unified	
instruction syntax	lsri.u32.r rz, rx, oimm5 operation
	Rz[31:0] <- Round(Rx[31:0] >> oimm[4:0]) only has 32-bit instructions in the compiled result .
	lsri.u32.r rz, rx, oimm5

Description:	Perform a logical right shift on the value of Rx (shift the original value right and fill 0 on the left side), and perform a rounding operation. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 5-bit immediate value (OIMM5) with
Affected flags :	offset 1. No effect
Restrictions:	The range of immediate values is
1-32. Exceptions:	None

32-bit	
instruction operation:	Rz[31:0] <- Round(Rx[31:0] >>oimm[4:0]) Syntax: lsri.u32.r
rz, rx, oimm5 Description:	Perform a logical right shift on the value of Rx (shift the original value right and fill 0 on the left), and perform a rounding operation. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 5-bit immediate value (OIMM5)
Affected flags :	with offset 1. No effect
Restrictions:	The immediate value range is
1-32 Exceptions:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Oimm5	Rx	1 1 0 1 0 0	1 1 0 0 1	Rz	

Figure 15.29: LSRI.U32.R

15.30 LSR.U32.R -- Logical Right Shift with Rounding

The unified	
instruction syntax	lsr.u32.r rz, rx, ry
operation	Rz[31:0] <- Round(Rx[31:0] >> Ry[5:0]) only has 32-bit instructions
in the compiled result .	lsr.u32.r rz, rx, ry

Description:	Perform a logical right shift on the value of Rx (shift the original value right and fill in 0 on the left), and perform a rounding operation. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by Ry[5:0]. No effect
Affected flag:	Ry[5:0]. No effect
Abnormal:	None

32-bit	
instruction operation:	Rz[31:0] <- Round(Rx[31:0] >> Ry[5:0]) Syntax: lsr.u32.r
rz, rx, ry Description:	Perform a logical right shift on the value of Rx (shift the original value right and fill in 0 on the left), and perform a rounding operation. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by Ry[5:0]. No effect
Affected flag:	determined by Ry[5:0]. No effect
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 0 0	1 1 0 0 1	Rz	

Figure 15.30: LSR.U32.R

15.31 LSLI.(U/S)32.S -- Immediate (Unsigned/With) Logical Shift Left Instruction with Saturation

Unified command	
syntax	lsli.u32.s rz, rx, oimm5 lsli.s32.s rz, rx, oimm5 Rz[31:0] <- Saturate(Rx[31:0])
The	<< oimm[4:0])
result of the operation compilation	is only 32-bit instructions. lsli.u32.s rz, rx, oimm5 lsli.s32.s rz, rx, oimm5

illustrate	Shift the value of Rx logically to the left (shift the original value to the left and fill the right side with 0), and perform saturation operation. The number of left shift bits is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, otherwise the result itself is taken. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the shift result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the result itself is taken. No impact
	The range of immediate value is 1-32.
Affected flags: Restrictions: Exceptions:	none

32	
Bit	
	Rz[31:0] <- Saturate(Rx[31:0] << oimm[4:0])
	Isli.u32.s rz, rx, oimm5 Isli.s32.s rz, rx, oimm5 Perform a logical left
	shift on the value of Rx (shift the original value left and fill the right side with 0), and perform a saturation operation. The number of left shifts is determined <small>instruction op by the value of the offset immediate value</small> (OIMM5) with offset 1. For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, otherwise the result itself is used. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the shift result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the result
	itself is used. No effect
	The range of immediate value is 1-32
Affected	none
	flags: Restrictions: Exceptions:

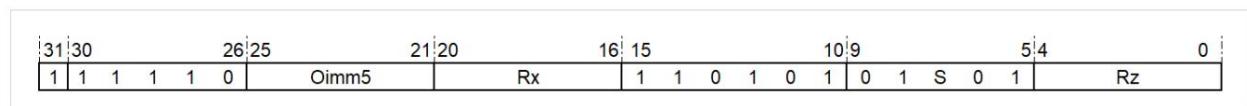
Instruction format:

Figure 15.31: LSLI.(US)32.S

15.32 LSL.(U/S)32.S -- Signed Logical Shift Left with Saturation (Unsigned/Signed)

Unified command	
syntax	lsl.u32.s rz, rx, ry lsl.s32.s rz, rx, ry The operation Rz[31:0]
<- Saturate(Rx[31:0] << Ry[5:0])	is compiled to have only 32-bit instructions.
	lsl.u32.s rz, rx, ry lsl.s32.s rz, rx, ry

illustrate : bits is determined by the value of Ry[5:0]. For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, otherwise the result itself is taken. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the shift result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the result itself is taken. No impact	Affected flag: Abnormal: none
---	---

32 Bit	
	Rz[31:0] <- Saturate(Rx[31:0] << Ry[5:0])
	lsl.u32.s rz, rx, ry lsl.s32.s rz, rx, ry Perform a logical
	left shift on the value of Rx (shift the original value to the left and add 0 to the right), and perform a saturation operation. The number of left shifts is instruction determined by the value of Ry[5:0]. For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF FFFF, the result is 0xFFFF FFFF, otherwise the result itself is used. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF FFFF, the result is 0x7FFF FFFF, if the shift result is less than the lower saturation value 0x8000 0000, the result is 0x8000 0000, otherwise the result itself is used. No effect
	none Affected flag: Abnormal:

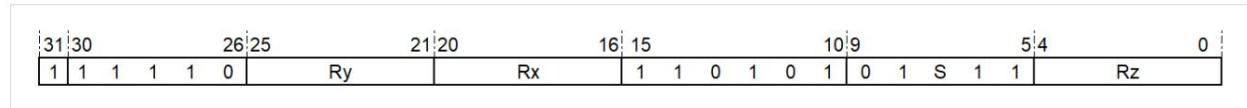
Instruction format:

Figure 15.32: LSL.(US)32.S

15.33 PASRI.S16 – 16-bit Parallel Immediate Arithmetic Shift Right Instruction

Unified command	
syntax	pasri.s16 rz, rx, oimm4 Rz[31:16] <-
operation	Rx[31:16] >>> oimm[3:0] Rz[15:0] <- Rx[15:0] >>> oimm[3:0] The compiled result
only contains 32-bit instructions.	pasri.s16 rz, rx, oimm4

Description:	The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted into the left side), and the shift results are stored in The upper and lower halfwords of Rz. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with offset
Affected flags :	1. No effect
Restrictions:	The immediate value range
is 1-16 Exceptions:	None

32-bit	
instruction operation:	Rz[31:16] <- Rx[31:16] >>> oimm[3:0] Rz[15:0] <- Rx[15:0] >>> oimm[3:0]
Syntax: pasr.s16 rz, rx, oimm4 Description: The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted into the left side), and the shift results are stored in The upper and lower halfwords of Rz. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with offset	
Affected flags :	1. No effect
Restrictions:	The immediate value range
is 1-16 Exceptions:	None

Instruction format:

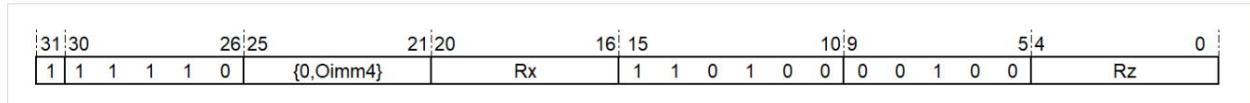


Figure 15.33: PASRI.S16

15.34 PASR.S16 — 16-bit Parallel Arithmetic Shift Right Instruction

Unified	
instruction syntax	pasr.s16 rz, rx, ry
operation	Rz[31:16] <- Rx[31:16] >>> Ry[4:0] Rz[15:0] <- Rx[15:0] >>> Ry[4:0] The compiled result only has 32-bit
instructions.	pasr.s16 rz, rx, ry

Description:	The upper and lower halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted in on the left), and the shift results are stored in the upper and lower halfwords of Rz respectively. The number of right shifts is determined by the value of the lower 5 bits of Ry
Affected flag:	(Ry[4:0]). No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:16] <- Rx[31:16] >>> Ry[4:0] Rz[15:0] <- Rx[15:0] >>> Ry[4:0]
Syntax:	pasr.s16 rz, rx,
ry Description:	The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted in on the left), and the shift results are stored in the high and low halfwords of Rz respectively. The number of right shifts is determined by the value of the lower 5 bits of Ry (Ry[4:0]). No effect
Affected flag:	
Restriction:	None
Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 1 0 1 0 0 0 0 1 1 0	Rz		

Figure 15.34: PASR.S16

15.35 PASRI.S16.R – 16-Bit Parallel Immediate Arithmetic Shift Right Instruction with Rounding

Unified	
	pasri.s16.r rz, rx, oimm4 Rz[31:16] <-
command syntax operation:	Rx[31:16] >>> oimm[3:0] Rz[15:0] <- Round(Rx[15:0] >>> oimm[3:0]) The compiled result only
contains 32-bit instructions.	pasri.s16.r rz, rx, oimm4

illustrate :	The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted in from the left), and the shift results are rounded and stored in the high and low halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with
Impact Mark	offset 1. No effect
Zhi	
Bit:	
	The immediate value range is 1-16
Limitation:	Exception: none

32 Bit	
	Rz[31:16] <- Round(Rx[31:16] >>> oimm[3:0]) Rz[15:0] <- Round(Rx[15:0] >>> oimm[3:0])
	pasri.s16.r rz, rx, oimm4
instruction operation : Syntax :	
illustrate :	The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted in from the left), and the shift results are rounded and stored in the high and low halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with
	offset 1. No effect
Affected flags:	
limit	The range of immediate value is 1-16
	none
Control: Abnormal:	

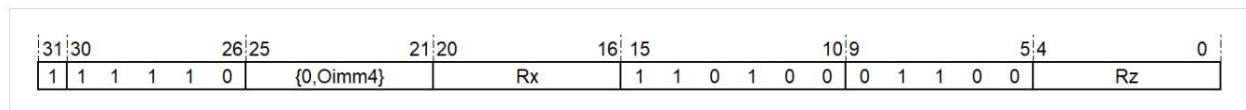
Instruction format:

Figure 15.35: PASRI.S16.R

15.36 PASR.S16.R -- 16-bit Parallel Arithmetic Shift Right with Rounding

Unified command	
syntax	<code>pasr.s16.r Rz, Rx, Ry</code> $Rz[31:16] <-$
operation	$\text{Round}(Rx[31:16] \ggg Ry[4:0])$ $Rz[15:0] <- \text{Round}(Rx[15:0] \ggg Ry[4:0])$ The compiled result only contains 32-bit instructions.
	<code>pasr.s16.r rz, rx, ry</code>

illustrate :	The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted in from the left), and the shift results are rounded and stored in the high and low halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the lower 5 bits
Affected flags:	of Ry (Ry[4:0]). No effect
	none
Restrictions: Exceptions:	none

32 Bit	
	$Rz[31:16] <- \text{Round}(Rx[31:16] \ggg Ry[4:0])$ $Rz[15:0] <- \text{Round}(Rx[15:0] \ggg Ry[4:0])$ <code>pasr.s16.r rz, rx, ry</code>
instruction operation syntax description	The high and low halfwords of Rx are arithmetic shifted right (the original value is shifted right, and a copy of the original sign bit is shifted in from the left), and the shift results are rounded and stored in the high and low halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the lower 5 bits
Affected flags:	of Ry (Ry[4:0]). No effect
	none
Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 0 0	0 1 1 1 0	Rz	

Figure 15.36: PASR.S16.R

15.37 PLSRI.U16 – 16-bit Parallel Immediate Logical Shift Right Instruction

Unified command	
syntax	plsri.u16 rz, rx, oimm4 Rz[31:16]
operation	<- Rx[31:16] >> oimm[3:0] Rz[15:0] <- Rx[15:0] >> oimm[3:0]
The compiled result	contains only 32-bit instructions. plsri.u16 rz, rx, oimm4

Description: The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left), and the shift results are stored in the upper and lower halfwords of Rz respectively. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with offset 1. No effect
Affected flags:
Restrictions: The range of immediate values is
Exceptions: None

32-bit	
instruction operation:	Rz[31:16] <- Rx[31:16] >> oimm[3:0] Rz[15:0] <- Rx[15:0] >> oimm[3:0]
Syntax:	plsri.u16 rz, rx, oimm4
Description:	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left), and the shift results are stored in the upper and lower halfwords of Rz respectively. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4)
Affected flags:	with offset 1. No effect
Restrictions:	The immediate value range is
Exceptions:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	{0,Oimm4}	Rx	1 1 0 1 0 0	1 0 0 0 0	Rz	

Figure 15.37: PLSR.U16

15.38 PLSR.U16—16-bit Parallel Logical Shift Right Instruction

Unified command	
syntax	plsr.u16 rz, rx, ry Rz[31:16]
operation	<- Rx[31:16] >> Ry[4:0] Rz[15:0] <- Rx[15:0] >> Ry[4:0]
The compiled result	contains only 32-bit instructions. plsr.u16 rz, rx, ry

Description:	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right and 0 is added on the left), and the shift results are stored in the upper and lower halfwords of Rz respectively. The number of right shifts is determined by the value of the lower 5 bits of Ry
Affected flag:	(RY[4:0]). No effect
Restriction:	None
Exception:	None

32 Bit	
	Rz[31:16] <- Round(Rx[31:16] >> Ry[4:0]) Rz[15:0] <- Round(Rx[15:0] >> Ry[4:0]) plsr.u16.rz, rx, ry
	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left side). The shift results are rounded and stored in the upper and lower halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the lower 5 bits of Ry
Affected flags:	Ry (Ry[4:0]). No effect
	none
Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 0 0	1 0 0 1 0	Rz	

Figure 15.38: PLSR.U16

15.39 PLSRI.U16.R – 16-Bit Parallel Immediate Logical Shift Right Instruction with Rounding

Unified command	
syntax	plsri.u16.r rz, rx, oimm4 Rz[31:16] <-
operation	Round(Rx[31:16] >> oimm[3:0]) Rz[15:0] <- Round(Rx[15:0] >> oimm[3:0]) The compiled result only has 32-bit instructions. plsri.u16.r rz, rx, oimm4

illustrate :	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left side). The shift results are rounded and stored in the upper and lower halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with offset 1. No effect flags:
	The range of immediate values is 1-16.
Restrictions: Exceptions:	none

32	
Bit	
	Rz[31:16] <- Round(Rx[31:16] >> oimm[3:0]) Rz[15:0] <- Round(Rx[15:0] >> oimm[3:0]) plsri.u16.r rz, rx,
	oimm4
instruction operation halves description:	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left side). The shift results are rounded and stored in the upper and lower halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with
Affected flags:	offset 1. No effect
	The range of immediate value is 1-16
Restrictions: Exceptions:	none

Instruction format:

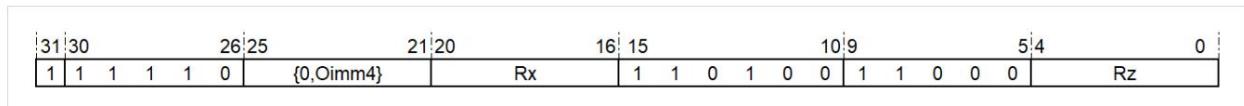


Figure 15.39: PLSRI.U16.R

15.40 PLSR.U16.R – 16-Bit Parallel Logical Shift Right Instruction with Rounding

Unified command	
syntax	plsri.u16.r rz, rx, ry Rz[31:16]
operation	<- Round(Rx[31:16] >> Ry[4:0]) Rz[15:0] <- Round(Rx[15:0] >> Ry[4:0]) The compiled
result only contains 32-bit instructions. plsri.u16.r rz, rx, ry	

Illustrate :	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left side). The shift results are rounded and stored in the upper and lower halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the lower 5 bits of Ry (Ry[4:0]). No effect
Affected flags:	Ry (Ry[4:0]). No effect
	none
	none
Restrictions: Exceptions:	

32 Bit	
	Rz[31:16] <- Round(Rx[31:16] >> Ry[4:0]) Rz[15:0] <- Round(Rx[15:0] >> Ry[4:0]) plsr.u16.r rz, rx, ry
instruction operation symbols Description:	The upper and lower halfwords of Rx are logically shifted right (the original value is shifted right, and 0 is added to the left side). The shift results are rounded and stored in the upper and lower halfwords of Rz respectively. The rounding operation process is that if the highest bit of the shifted data is 1, the shift result is increased by 1, otherwise the shift result remains unchanged. The number of right shifts is determined by the value of the lower 5 bits of Ry (Ry[4:0]). No effect
Affected flags:	Ry (Ry[4:0]). No effect
	none
	none
Restrictions: Exceptions:	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 0 0	1 1 0 1 0 0	Rz	

Figure 15.40: PLSR.U16.R

15.41 PLSLI.16 – 16-bit Parallel Immediate Logical Left Shift Instruction

Unified command	
syntax	plsl.i.16 rz, rx, oimm4 Rz[31:16]
operation	<- Rx[31:16] << oimm[3:0] Rz[15:0] <- Rx[15:0] << oimm[3:0] The compiled
result only contains 32-bit instructions. plsl.i.16 rz, rx, oimm4	

Description: The upper and lower halfwords of Rx are logically shifted left (the original value is shifted left and 0 is added on the right), and the shift results are stored in the upper and lower halfwords of Rz respectively. The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with offset
Affected flag: 1. No effect
Restriction:
None Exception: None

32-bit	
instruction operation: Rz[31:16] <- Rx[31:16] << oimm[3:0] Rz[15:0] <- Rx[15:0] << oimm[3:0]	
Syntax: plsl.i.16 rz, rx, oimm4 Explanation: The high	
and low halfwords of Rx are logically shifted left (the original value is shifted left and 0 is added on the right), and the shift results are stored in the high and low halfwords of Rz respectively.	
The number of right shifts is determined by the value of the 4-bit immediate value (OIMM4) with offset	
Affected flag: 1. No effect	
Restriction:	
None Exception: None	

Instruction format:

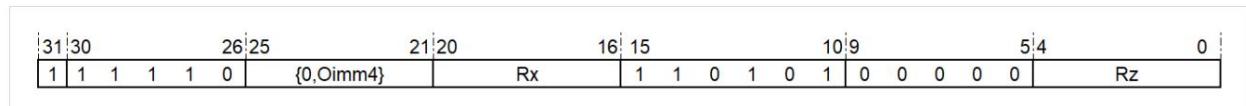


Figure 15.41: PLSLI.16

15.42 PSL.16 – 16-bit Parallel Logical Shift Left Instruction

Unified command	
syntax	<code>psl.16 rz, rx, ry Rz[31:16]</code>
operation	$\leftarrow Rx[31:16] \ll Ry[4:0]$ $Rz[15:0] \leftarrow Rx[15:0] \ll Ry[4:0]$
The compiled result	contains only 32-bit instructions. <code>psl.16 rz, rx, ry</code>

Description: The upper and lower halfwords of Rx are logically shifted left (the original value is shifted left and 0 is added on the right), and the shift results are stored in the upper and lower halfwords of Rz respectively. The number of right shifts is determined by the value of the lower 5 bits of Ry
Affected flag: (Ry[4:0]). No effect
Restriction: None
Exception: None

32-bit	
instruction operation: $Rz[31:16] \leftarrow Rx[31:16] \ll Ry[4:0]$	
	Rz[15:0] $\leftarrow Rx[15:0] \ll Ry[4:0]$ Syntax: <code>psl.16 rz, rx,</code>
ry Explanation: The upper and lower	
halfwords of Rx	are logically shifted left (the original value is shifted left and 0 is added on the right), and the shift results are stored in the upper and lower halfwords of Rz respectively.
	The number of right shifts is determined by the value of the lower 5 bits of Ry
Affected flag:	(Ry[4:0]). No effect
Restriction: None	
Exception: None	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 1 0 1 0 1 0 0 0 1 0	Rz		

Figure 15.42: PSL.16

15.43 PLSLI.(U/S)16.S – 16-Bit Parallel Immediate (No/Yes) Signed Logical Shift Left Instruction with Saturation

Unified command	
syntax	<pre>plsl.s16.s rz, rx, oimm4 plsl.u16.s rz, rx, oimm4</pre>
operate	$Rz[31:16] \leftarrow \text{Saturate}(Rx[31:16] \ll oimm[3:0])$ $Rz[15:0] \leftarrow \text{Saturate}(Rx[15:0] \ll oimm[3:0])$ The compiled result only has
32-bit instructions.	<pre>plsl.s16.s rz, rx, oimm4 plsl.u16.s rz, rx, oimm4</pre>

explain	Perform logical left shift on the upper and lower half words of Rx respectively (shift the original value to the left and fill in 0 on the right), and perform saturation operation.
bright:	<p>The number of left shift bits is determined by the value of the 5-bit immediate value (OIMM5) with offset 1. For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF, the result is 0xFFFF, otherwise the result itself is used. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the shift result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the result</p>
	itself is used. No impact
	none
	none
Affected flags:	
Restrictions:	
Exceptions:	

32	Bit
	Rz[31:16] <- Saturate(Rx[31:16] << oimm[3:0]) Rz[15:0] <- Saturate(Rx[15:0] << oimm[3:0]) plsl.u16.s rz, rx, oimm4
	plsl.u16.s rz, rx, oimm4 Perform logical left shift on the upper and lower
	halfwords of Rx respectively (shift the original value left and fill with 0 on the right), and perform saturation operation. The number of left shift bits is determined by the value of the fifth immediate value (OIMM5) with offset 1. For unsigned numbers, the saturation processing process is: if the shift result is greater than the saturation value 0xFFFF, the result is 0xFFFF, otherwise the result itself is taken. For signed numbers, the saturation processing process is: if the shift result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the shift result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the result itself is taken. No effect
	The range of immediate value is 1-16
	none
Affected	flags: Restrictions: Exceptions:

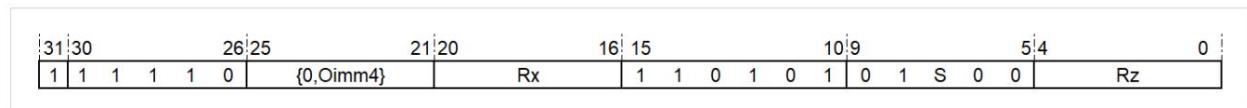
Instruction format:

Figure 15.43: PLSLI.(US)16.S

15.44 PSL.(U/S)16.S – 16-bit Parallel (Un/With) Signed Logical Shift Left Instruction with Saturation

Unified command	
syntax	plsl.s16.s rz, rx, ry plsl.u16.s rz, rx, ry Rz[31:16] <- Saturate(Rx[31:16])
operate	<< Ry[4:0]) Rz[15:0] <- Saturate(Rx[15:0] << Ry[4:0]) The compiled result only has 32-bit instructions. plsl.s16.s rz, rx, ry plsl.u16.s rz, rx, ry

explain bright:	Perform logical left shift on the upper and lower halfwords of Rx respectively (shift the original value to the left and fill in 0 on the right), and perform saturation operation. The number of left shift bits is determined by the value of the lower 5 bits of Ry (Ry[4:0]). For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF, the result is 0xFFFF, otherwise the result itself is used. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the shift result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the result itself is used. No impact
	none
Affected	none
flags:	Restrictions: Exceptions:

32	
Bit	
	Rz[31:16] <- Saturate(Rx[31:16] << Ry[4:0]) Rz[15:0] <- Saturate(Rx[15:0] << Ry[4:0]) psl.u16.s rz, rx, ry
	psl.u16.s rz, rx, ry Logically shift the upper and lower
	halfwords of Rx left (shift the original value left and fill in 0 on the right), and perform saturation operation. The number of left shifts is determined by the value instruction offset lower 8 bits of Ry(Ry[4:0]). For unsigned numbers, the saturation process is as follows: if the shift result is greater than the saturation value 0xFFFF, the result is 0xFFFF, otherwise the result itself is taken. For signed numbers, the saturation process is as follows: if the shift result is greater than the upper saturation value 0x7FFF, the result is 0x7FFF, if the shift result is less than the lower saturation value 0x8000, the result is 0x8000, otherwise the result itself is
	taken. No effect
	none
	none
Affected flags:	Restrictions:
Exceptions:	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 1 0 1 0 1 0 1 S 1 0	Rz		

Figure 15.44: PSL.(US)16.S

15.45 SEL——Bit Select Instruction

Unified command	
syntax	sel rz, rx, ry, rs
operation	for(i=0;i<31;i++) Rz[i] = Rs[i] ? Rx[i] : Ry[i]
The compiled result	contains only 32-bit instructions. sel rz, rx, ry, rs

Description:	According to the value of each bit in the bit select register Rs, select the corresponding bit in Rx and Ry and store it in Rz. Rx[i] is stored in Rz[i], otherwise Ry[i] is selected and stored in
Affected flag:	Rz[i]. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	for(i=0;i<31;i++) Rz[i] = Rs[i] ? Rx[i] : Ry[i]
Syntax: sel rz, rx, ry, rs	Description:
	According to the value of each bit in the bit select register Rs, select the corresponding bit in Rx and Ry and store it in Rz. If Rs[i]==1, select Rx[i] is stored in Rz[i], otherwise Ry[i] is selected and stored in
Affected flag:	Rz[i]. No effect
Restriction:	None
Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 0 0 1 0 0	Rs	Rz	

Figure 15.45: SEL

15.46 PCMPNE.8——8-bit parallel compare not equal instruction

Unified	
Instruction Syntax	pcmpne.8 rz, rx, ry
Operation	Rz[31:24] = (Rx[31:24] != Ry[31:24]) ? 8'hFF:8'h00 Rz[23:16] = (Rx[23:16] != Ry[23:16]) ? 8'hFF:8'h00 Rz[15:8] = (Rx[15:8] != Ry[15:8]) ? 8'hFF:8'h00 Rz[7:0] = (Rx[7:0] != Ry[7:0]) ? 8'hFF:8'h00 Only 32-bit instructions exist. pcmpne.8 rz, rx, ry
Compilation results	

Description:	In byte units, determine whether the 4 bytes of Rx are equal to the 4 bytes of Ry. If they are equal, the corresponding byte of Rz is equal to The result of the corresponding byte of Rz is 0xFF, otherwise the result of the corresponding byte of Rz is 0x0. No effect
Affected flag:	None
Restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:24] = (Rx[31:24] != Ry[31:24]) ? 8'hFF:8'h00 Rz[23:16] = (Rx[23:16] != Ry[23:16]) ? 8'hFF:8'h00 Rz[15:8] = (Rx[15:8] != Ry[15:8]) ? 8'hFF:8'h00 Rz[7:0] = (Rx[7:0] != Ry[7:0]) ? 8'hFF:8'h00
Syntax:	pcmpne.8 rz, rx, ry
Description:	In bytes, determine whether the 4 bytes of Rx are equal to the 4 bytes of Ry.
If they are equal, the corresponding bytes of Rz are equal.	The result is 0xFF, otherwise the result of the byte corresponding to Rz is 0x0.
Affected flag:	0x0. No effect
Restriction:	None
Exception:	None

Instruction format:

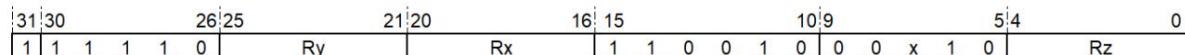


Figure 15.46: PCMPNE.8

15.47 PCMPNE.16——16-bit parallel compare not equal instruction

Unified	
instruction syntax	pcmpne.16 rz, rx, ry
Operation	Rz[31:16] = (Rx[31:16] != Ry[31:16]) ? 16'hFFFF:16'b0000 Rz[15:0] = (Rx[15:0] != Ry[15:0]) ? 16'hFFFF:16'b0000 Only 32-bit instructions exist. pcmpne.16 rz, rx, ry
Compilation results	

Description:	Determine whether the upper and lower halfwords of Rx are equal to the upper and lower halfwords of Ry. If they are equal, the result of the corresponding halfword of Rz is 0xFFFF, otherwise the result of the byte corresponding to Rz is 0x0.
Affected flag:	None
Restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:16] = (Rx[31:16] != Ry[31:16]) ? 16'hFFFF:16'b0000 Rz[15:0] = (Rx[15:0] != Ry[15:0]) ? 16'hFFFF:16'b0000
Syntax:	pcmpne.16 rz, rx, ry
Description:	Determine whether the upper and lower halfwords of Rx are equal to the upper and lower halfwords of Ry. If they are equal, the result of the corresponding halfword of Rz is 0xFFFF, otherwise the result of the byte corresponding to Rz is 0x0.
Affected flag:	None
Restriction:	None
Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 1 0 0 1 0 0 0 x 0 0	Rz		

Figure 15.47: PCMPNE.16

15.48 PCMPHS.(U/S)8——8-bit parallel (unsigned/signed) greater than or equal to comparison instruction

Unified	
instruction syntax	pcmphs.u8 rz, rx, ry pcmphs.s8 rz, rx, ry
Operation	Rz[31:24] = (Rx[31:24] >= Ry[31:24]) ? 8'hFF:8'h00 Rz[23:16] = (Rx[23:16] >= Ry[23:16]) ? 8'hFF:8'h00 Rz[15:8] = (Rx[15:8] >= Ry[15:8]) ? 8'hFF:8'h00 Rz[7:0] = (Rx[7:0] >= Ry[7:0]) ? 8'hFF:8'h00 Only 32-bit instructions exist. pcmphs.u8 rz, rx, ry pcmphs.s8 rz, rx, ry
Compilation results	

Description:	In bytes, determine whether the 4 bytes of Rx are greater than or equal to the 4 bytes of Ry. If the byte is equal to Ry, the result of the byte corresponding to Rz is 0xFF, otherwise the result of the byte corresponding to Rz is 0x0. No effect
Affected flag:	
Restriction:	
None	Exception: None

32 Bit	
instruction operation:	Rz[31:24] = (Rx[31:24] >= Ry[31:24]) ? 8'hFF:8'h00 Rz[23:16] = (Rx[23:16] >= Ry[23:16]) ? 8'hFF:8'h00 Rz[15:8] = (Rx[15:8] >= Ry[15:8]) ? 8'hFF:8'h00 Rz[7:0] = (Rx[7:0] >= Ry[7:0]) ? 8'hFF:8'h00
Syntax:	pcmphs.u8 rz, rx, ry pcmphs.s8 rz, rx, ry
Description:	In bytes, determine whether the 4 bytes of Rx are greater than or equal to the 4 bytes of Ry. For the byte of Ry, the result of the byte corresponding to Rz is 0xFF, otherwise the result of the byte corresponding to Rz is 0x0. No effect
Affected flag:	
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 1 0 0 1 S 1 0	Rz		

Figure 15.48: PCMPHS.(US)8

15.49 PCMPHS.(U/S)16——16-bit parallel (without/with) signed greater than or equal to comparison

instruction

Unified	
instruction syntax	pcmphs.u16 rz, rx, ry pcmphs.s16 rz, rx, ry
Operation	Rz[31:16] = (Rx[31:16] >= Ry[31:16]) ? 16'hFFFF:16'b0000 Rz[15:0] = (Rx[15:0] >= Ry[15:0]) ? 16'hFFFF:16'b0000 Only 32-bit instructions exist. pcmphs.u16 rz, rx, ry pcmphs.s16
Compilation results	rz, rx, ry

Description:	Determine whether the upper and lower halfwords of Rx are greater than or equal to the upper and lower halfwords of Ry. If the halfword of Rx is greater than or equal to the halfword of Ry, the result of the corresponding halfword of Rz is 0xFFFF, otherwise the result of the corresponding byte
Affected flag:	of Rz is 0x0. No effect
Restriction:	
None	Exception: None

32-bit	
instruction operation: Rz[31:16] = (Rx[31:16] >= Ry[31:16]) ? 16'hFFFF:16'b0000 Rz[15:0] = (Rx[15:0] >= Ry[15:0]) ? 16'hFFFF:16'b0000	
Syntax: pcmphs.u16 rz, rx, ry pcmphs.s16 rz, rx, ry	Description: Determine whether
the upper and lower halfwords of Rx are greater than or equal to the upper and lower halfwords of Ry. If the halfword of Rx is greater than or equal to the halfword of Ry, the result of the halfword corresponding to Rz is 0xFFFF, otherwise the result of the byte corresponding to Rz is	
Affected flag:	0x0. No effect
Restriction:	
None	Exception: None

Instruction format:

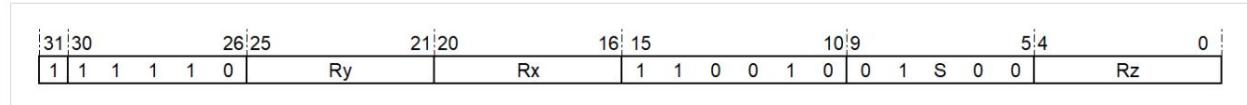


Figure 15.49: PCMPHS.(US)16

15.50 PCMPLT.(U/S)8——8-bit parallel (unsigned/signed) less than comparison select instruction

Unified	
command syntax	pcmplt.u8 rz, rx, ry pcmplt.s8 rz, rx, ry
Operations	Rz[31:24] = (Rx[31:24] < Ry[31:24]) ? 8'hFF:8'h00 Rz[23:16] = (Rx[23:16] < Ry[23:16]) ? 8'hFF:8'h00 Rz[15:8] = (Rx[15:8] < Ry[15:8]) ? 8'hFF:8'h00 Rz[7:0] = (Rx[7:0] < Ry[7:0]) ? 8'hFF:8'h00 Only 32-bit instructions exist. pcmplt.u8 rz, rx, ry pcmpkt.s8 rz, rx, ry
Compilation results	

Description:	In bytes, determine whether the 4 bytes of Rx are less than the 4 bytes of Ry. If the byte of Rx is less than the byte of Ry, the result of the byte corresponding to Rz is 0xFF, otherwise the result of the byte corresponding to Rz is 0x0.
Affected flag:	No effect
Restriction:	
None	Exception: None

32-bit	
instruction operation:	Rz[31:24] = (Rx[31:24] < Ry[31:24]) ? 8'hFF:8'h00 Rz[23:16] = (Rx[23:16] < Ry[23:16]) ? 8'hFF:8'h00 Rz[15:8] = (Rx[15:8] < Ry[15:8]) ? 8'hFF:8'h00 Rz[7:0] = (Rx[7:0] < Ry[7:0]) ? 8'hFF:8'h00 Syntax: pcmplt.u8 rz, rx, ry pcmplt.s8 rz, rx, ry
Description:	In bytes, determine whether the 4 bytes of Rx are less than the 4 bytes of Ry. If the byte of Rx is less than the byte of Ry, the result of the byte corresponding to Rz is 0xFF, otherwise the result of the byte corresponding to Rz is 0x0.
Affected flag:	No effect
Restriction:	
None	Exception: None

Instruction format:

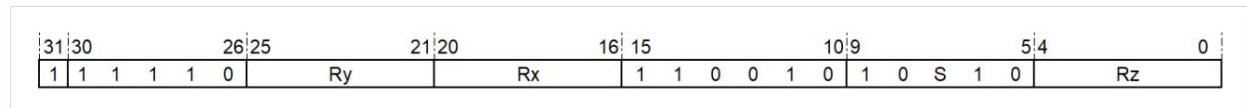


Figure 15.50: PCMPLT.(US)8

15.51 PCMPLT.(U/S)16——16-bit parallel (unsigned/signed) less than comparison instruction

Unified	
command syntax	pcmplt.u16 rz, rx, Ry pcmplt.s16 rz, rx, ry
Operation	Rz[31:16] = (Rx[31:16] < Ry[31:16]) ? 16'hFFFF:16'b0000 Rz[15:0] = (Rx[15:0] < Ry[15:0]) ? 16'hFFFF:16'b0000 Only 32-bit instructions exist. pcmplt.u16 rz, rx, Ry pcmplt.s16
Compilation results	rz, rx, ry

Description:	Determine whether the upper and lower halfwords of Rx are less than the upper and lower halfwords of Ry. If the halfword of Rx is less than the halfword of Ry, the result of the corresponding halfword of Rz is 0xFFFF, otherwise the result of the corresponding byte of Rz
Affected flag:	is 0x0. No effect
Restriction:	
None Exception:	None

32-bit	
instruction operation:	Rz[31:16] = (Rx[31:16] < Ry[31:16]) ? 16'hFFFF:16'b0000 Rz[15:0] = (Rx[15:0] < Ry[15:0]) ? 16'hFFFF:16'b0000 Syntax: pcmplt.u16 rz, rx, Ry pcmplt.s16 rz, rx,
ry Description:	Determine whether the upper and lower halfwords of Rx
are less than	the upper and lower halfwords of Ry. If the halfword of Rx is less than the halfword of Ry, Rz is equal to the lower halfword of Ry. The result for a half-word is 0xFFFF, otherwise the result for the byte corresponding to Rz is
Affected flag:	0x0. No effect
Restriction:	
None Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 1 0	1 0 S 0 0	Rz	

Figure 15.51: PCMPLT.(US)16

15.52 PMAX.(U/S)8——8-bit parallel (without/with) sign-to-value instruction

Unified	
instruction syntax	pmax.u8 rz, rx, ry pmax.s8 rz, rx, ry
Operation	Rz[31:24] = Max(Rx[31:24], Ry[31:24]) Rz[23:16] = Max(Rx[23:16], Ry[23:16]) Rz[15:8] = Max(Rx[15:8], Ry[15:8]) Rz[7:0] = Max(Rx[7:0], Ry[7:0]) Only exists
Compilation results	for 32-bit instructions. pmax.u8 rz, rx, ry pmax.s8 rz, rx, ry

	In units of bytes, select the larger of the corresponding bytes of Rx and Ry, and store the result in the corresponding byte of Rz.
Description: Impact flag:	None
Impact restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:24] = Max(Rx[31:24], Ry[31:24]) Rz[23:16] = Max(Rx[23:16], Ry[23:16]) Rz[15:8] = Max(Rx[15:8], Ry[15:8]) Rz[7:0] = Max(Rx[7:0], Ry[7:0])
Syntax:	pmax.u8 rz, rx, ry pmax.s8 rz, rx, ry Description: In
bytes, select	the larger of the corresponding bytes of Rx and Ry, and store the result in the corresponding byte of Rz. Affected flags:
	No impact
Restrictions:	
None Exceptions:	None

Instruction format:

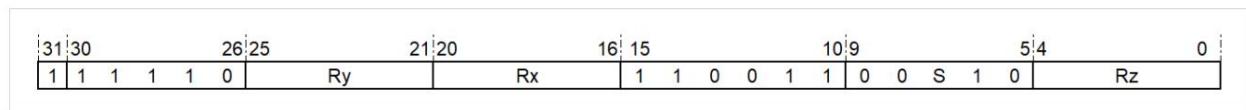


Figure 15.52: PMAX.(US)8

15.53 PMAX.(U/S)16——16-bit parallel (without/with) signed large value instruction

Unified instructions	
grammar	pmax.u16 rz, rx, ry pmax.s16 rz, rx, ry
operate	Rz[31:16] = Max(Rx[31:16], Ry[31:16]) Rz[15:0] = Max(Rx[15:0], Ry[15:0])
The compiled result	contains only 32-bit instructions. pmax.u16 rz, rx, ry pmax.s16 rz, rx, ry

	In half-word units, select the larger of the corresponding half-words of Rx and Ry, and store the result in the corresponding half-word of Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
operate:	Rz[31:16] = Max(Rx[31:16], Ry[31:16]) Rz[15:0] = Max(Rx[15:0], Ry[15:0])
grammar:	pmax.u16 rz, rx, ry pmax.s16 rz, rx, ry
Explanation:	In half-word units, select the larger of the corresponding half-words of Rx and Ry, and store the result in the corresponding half-word of Rz.
Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 1	1 0 0 S 0 0	Rz	

Figure 15.53: PMAX.(US)16

15.54 MAX.(U/S)32——32-bit (unsigned/signed) maximum value instruction

Unified instructions	
grammar	max.u32 rz, rx, ry max.s32 rz, rx, ry
The	Rz[31:0] = Max(Rx[31:0], Ry[31:0])
result of the operation compilation	is only 32-bit instructions. max.u32 rz, rx, ry max.s32 rz, rx, ry

	Take the larger of Rx and Ry and store the result in Rz.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	none

32-bit instructions	
	Rz[31:0] = Max(Rx[31:0], Ry[31:0])
Operation: Syntax:	max.u32 rz, rx, ry max.s32 rz, rx, ry
	Take the larger of Rx and Ry and store the result in Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

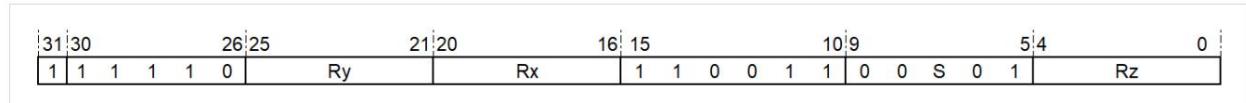


Figure 15.54: MAX.(US)32

15.55 PMIN.(U/S)8——8-bit parallel (without/with) signed minimum instruction

Unified	
instruction syntax	pmin.u8 rz, rx, ry pmin.s8 rz, rx, ry
Operation	Rz[31:24] = Min(Rx[31:24], Ry[31:24]) Rz[23:16] = Min(Rx[23:16], Ry[23:16]) Rz[15:8] = Min(Rx[15:8], Ry[15:8]) Rz[7:0] = Min(Rx[7:0], Ry[7:0]) Only exists
Compilation results	for 32-bit instructions. pmin.u8 rz, rx, ry pmin.s8 rz, rx, ry

	In byte units, select the smaller of the corresponding bytes of Rx and Ry, and store the result in the corresponding byte of Rz.
Description: Impact flag:	None
Impact restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:24] = Min(Rx[31:24], Ry[31:24]) Rz[23:16] = Min(Rx[23:16], Ry[23:16]) Rz[15:8] = Min(Rx[15:8], Ry[15:8]) Rz[7:0] = Min(Rx[7:0], Ry[7:0])
Syntax:	pmin.u8 rz, rx, ry pmin.s8 rz, rx, ry Description: In bytes,
	select the smaller of the corresponding bytes of Rx and Ry, and store the result in the corresponding byte of Rz. Affected flags: Restrictions:
None	No impact
Exceptions:	None

Instruction format:

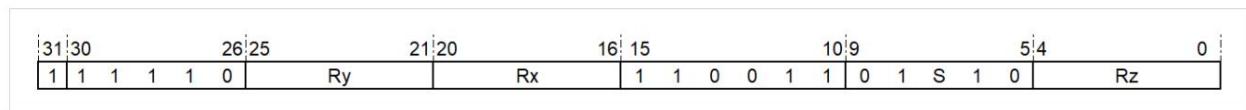


Figure 15.55: PMIN.(US)8

15.56 PMIN.(U/S)16——16-bit parallel (without/with) signed minimum value instruction

Unified instructions	
grammar	pmin.u16 rz, rx, ry pmin.s16 rz, rx, ry
operate	Rz[31:16] = Min(Rx[31:16], Ry[31:16]) Rz[15:0] = Min(Rx[15:0], Ry[15:0])
The compiled result	contains only 32-bit instructions. pmin.u16 rz, rx, ry pmin.s16 rz, rx, ry

	In half-word units, select the smaller of the corresponding half-words of Rx and Ry, and store the result in the corresponding half-word of Rz.
Description:	Impact flag: No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
operate:	Rz[31:16] = Min(Rx[31:16], Ry[31:16]) Rz[15:0] = Min(Rx[15:0], Ry[15:0])
grammar:	pmin.u16 rz, rx, ry pmin.s16 rz, rx, ry
Description:	In half-word units, select the smaller of the corresponding half-words of Rx and Ry, and store the result in the corresponding half-word of Rz.
Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 0 0 1	1 0 1 S 0 0	Rz	

Figure 15.56: PMIN.(US)16

15.57 MIN.(U/S)32——32-bit (without/with) signed minimum value instruction

Unified instructions	
grammar	min.u32 rz, rx, ry min.s32 rz, rx, ry
The	Rz[31:0] = Min(Rx[31:0], Ry[31:0])
result of the operation compilation	is only 32-bit instructions. min.u32 rz, rx, ry min.s32 rz, rx, ry

	Take the smaller of Rx and Ry and store the result in Rz.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	none

32-bit instructions	
	Rz[31:0] = Min(Rx[31:0], Ry[31:0])
Operation: Syntax:	min.u32 Rz, Rx, Ry min.s32 Rz, Rx, Ry
	Take the smaller of Rx and Ry and store the result in Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

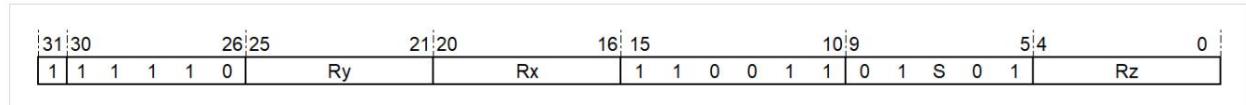


Figure 15.57: MIN.(US)32

15.58 DEXTI -- Digital Intercept Immediate Instruction

The unified	
	dexti rz, rx, ry, imm5
	$Rz[31:0] = \{Ry[31:0], Rx[31:0]\}[imm[4:0]+31:imm[4:0]]$
instruction syntax operation compilation result	only has 32-bit instructions. dexti rz, rx, ry, imm5

Description:	Ry and Rx form a 64-bit number {Ry[31:0], Rx[31:0]}, perform a logical right shift on the number, and store the low word of the result in RZ. The number of right shifts is determined by the value of the 5-bit immediate value
Affected flags:	(IMM5). No effect
Restrictions:	The range of immediate values is
0-31 Exceptions:	None

32-bit	
instruction operation:	$Rz[31:0] = \{Ry[31:0], Rx[31:0]\}[imm[4:0]+31:imm[4:0]]$ Syntax: dexti rz, rx, ry, imm5
Description:	Ry and Rx are combined to form a 64-bit number {Ry[31:0], Rx[31:0]}, and the number is logically shifted right, and the low word of the result is stored in RZ. The number of right shifts is determined by the value of the 5-bit immediate value
Affected flags:	(IMM5). No effect
Restrictions:	The range of immediate values is
0-31 Exceptions:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 1 1 0	Imm5	Rz	

Figure 15.58: DEXTI

15.59 EXT——Word Interception Instruction

The unified	
	dext rz, rx, ry, rs
	Rz[31:0] = {Ry[31:0], Rx[31:0]}[Rs[5:0]+31:Rs[5:0]]
instruction syntax operation compilation result	only has 32-bit instructions. dext rz, rx, ry, rs

Description:	Ry and Rx form a 64-bit number {Ry[31:0], Rx[31:0]}, perform a logical right shift on the number, and store the low word of the result in RZ. The number of right shifts is determined by the value of Rs[5:0]. When Rs[5:0] is greater than 32, the number of right shifts is 32. No effect
Affected flags :	
Restrictions:	The range of Ry[5:0] is 0-32
Exceptions:	None

32-bit	
instruction operation:	Rz[31:0] = {Ry[31:0], Rx[31:0]}[Rs[5:0]+31:Rs[5:0]] Syntax:
flags :	dext rz, rx, ry, rs Description: Ry and Rx form a 64-bit number {Ry[31:0], Rx[31:0]}, perform a logical right shift on the number, and store the low word of the result in RZ. The number of right shifts is determined by the value of Rs[5:0]. When Rs[5:0] is greater than 32, the number of right shifts is 32. No effect
Restrictions:	The range of Ry[5:0] is 0-32
Exceptions:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 0 0 1 1 1	Rs	Rz	

Figure 15.59: EXT

15.60 PKG——Packed Immediate Shift Instructions

The unified	
	pkg rz, rx, imm4a, ry, oimm4b
	Rz[31:0] = {{Ry[31:0] >> oimm4b[3:0])[15:0], (Rx[31:0] >> imm4a[3:0])[15:0]}
instruction syntax operation compilation result	only has 32-bit instructions.
	pkg rz, rx, imm4a, ry, oimm4b

explain bright:	For logical right shift of Rx, the lower half word of the shift result is intercepted and stored in the lower half word of RZ. The number of right shifts is determined by the value of the immediate value (imm4a[3:0]); for logical right shift of RY, the lower half word of the shift result is intercepted and stored in the upper half word of RZ. The number of right shifts is determined by the value of the immediate value with offset
Affected flags:	1 (oimm4b[3:0]). No effect
	The range of imm4a is 0~15 and the range of oimm4b is 1~16
	none
Restrictions: Exceptions:	

32 Bit	
	Rz[31:0] = {{Ry[31:0] >> oimm4b[3:0])[15:0], (Rx[31:0] >> imm4a[3:0])[15:0]}
instruction operation : Syntax :	pkg rz, rx, imm4a, ry, oimm4b
illustrate :	For logical right shift of Rx, the lower half word of the shift result is intercepted and stored in the lower half word of RZ. The number of right shifts is determined by the value of the immediate value (imm4a[3:0]); for logical right shift of RY, the lower half word of the shift result is intercepted and stored in the upper half word of RZ. The number of right shifts is determined by the value of the immediate value with offset
Affected flags:	1 (oimm4b[3:0]). No effect
limit	The range of imm4a is 0~15. The range of oimm4b is 1~16. None
Control: Abnormal:	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0:
1 1 1 1 0	Ry	Rx	1 0 1	oimm4b	oimm4a	Rz

Figure 15.60: PKG

15.61 PKGLL——Pack Lower Halfword Instruction

Unified instructions	
Syntax	pkgll rz, rx, ry
	Rz[31:0] = {Ry[15:0], Rx[15:0]}
Operation Compiled results	contain only 32-bit instructions. pkgll rz, rx, ry

Description: Store the lower halfword of Ry into the upper halfword of Rz, and store the lower halfword of Rx into the lower halfword of Rz.
Impact flag: No impact
Restrictions: None
Abnormal: None

32-bit instructions	
Operation: Rz[31:0] = {Ry[15:0], Rx[15:0]}	
Syntax: pkgll rz, rx, ry	
	The lower halfword of Ry is stored in the upper halfword of Rz, and the lower halfword of Rx is stored in the lower halfword of Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0:
1 1 1 1 0	Ry	Rx	1 1 0 1 1 0 0 0 x 1 0		Rz	

Figure 15.61: PKGLL

15.62 PKGHH——Pack High Halfword Instruction

Unified instructions	
Syntax	pkghh rz, rx, ry
	Rz[31:0] = {Ry[31:16], Rx[31:16]}
Operation Compiled	results contain only 32-bit instructions. pkghh rz, rx, ry

	The high halfword of Ry is stored in the high halfword of Rz, and the high halfword of Rx is stored in the low halfword of Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
	Rz[31:0] = {Ry[31:16], Rx[31:16]}
	pkghh rz, rx, ry
	Store the high halfword of Ry into the high halfword of Rz, and store the high halfword of Rx into the low halfword of Rz.
Operation: Syntax: Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

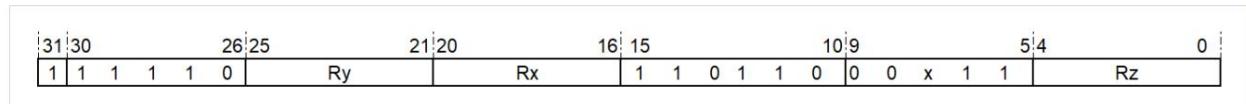


Figure 15.62: PKGHH

15.63 PEXT.U8.E – 8-bit Parallel Unsigned Extension Instruction

Unified	
instruction syntax	<code>pext.u8.e rz, rx</code>
operation	$Rz+1[31:16] = \text{zero_extend}(Rx[31:24])$ $Rz+1[15:0] = \text{zero_extend}(Rx[23:16])$ $Rz[31:16] = \text{zero_extend}(Rx[15:8])$ $Rz[15:0] = \text{zero_extend}(Rx[7:0])$ Only exists for 32-bit instructions. <code>pext.u8.e rz, rx</code>
Compilation results	

illustrate :	Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform unsigned extension on byte 3, byte 2, byte 1, and byte 0 in sequence. The extended data format is unsigned half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence. No effect
	none
	none
Affected flags: Restrictions: Exceptions:	

32	Bit
	<p>Rz+1[31:16] = zero_extend(Rx[31:24]) Instruction operation: Rz+1[15:0] = zero_extend(Rx[23:16]) Rz[31:16] = zero_extend(Rx[15:8]) Rz[15:0] = zero_extend(Rx[7:0])</p>
	pext.u8.e rz, rx
	<p>Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform unsigned extension on byte 3, byte 2, byte 1, and byte 0 in sequence. The extended data format is unsigned half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence. No effect</p>
Affected	flags:
limit	none
Control: Abnormal:	none

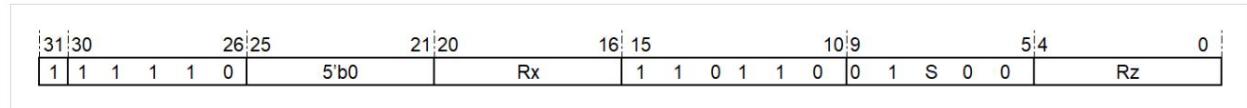
Instruction format:

Figure 15.63: PEXT.U8.E

15.64 PEXT.S8.E – 8-Bit Parallel Sign-Extend Instruction

Unified	
instruction syntax	pext.s8.e rz, rx
operation	<p>Rz+1[31:16] = sign_extend(Rx[31:24])</p> <p>Rz+1[15:0] = sign_extend(Rx[23:16])</p> <p>Rz[31:16] = sign_extend(Rx[15:8])</p> <p>Rz[15:0] = sign_extend(Rx[7:0]) Only exists for</p>
Compilation results	32-bit instructions. pext.s8.e rz, rx

illustrate :	Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform sign extension on byte 3, byte 2, byte 1, and byte 0 in sequence. The extended data format is a signed half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence.
	No effect
	none
Affected flags: Restrictions: Exceptions:	none

32	Bit
	<p>Rz+1[31:16] = sign_extend(Rx[31:24]) Instruction operation: Rz+1[15:0] = sign_extend(Rx[23:16]) Rz[31:16] = sign_extend(Rx[15:8]) Rz[15:0] = sign_extend(Rx[7:0])</p>
	pext.s8.e rz, rx
	<p>Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform sign extension on byte 3, byte 2, byte 1, and byte 0 in sequence. The extended data format is a signed half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence.</p>
	No effect
	Affected flags:
limit	none
	none
Control: Abnormal:	

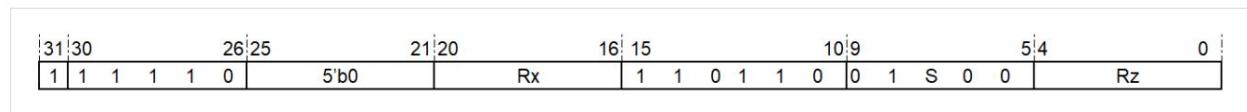
Instruction format:

Figure 15.64: PEXT.S8.E

15.65 PEXTX.U8.E -- 8-bit Parallel Unsigned Crossover Extend Instruction

Unified	
instruction syntax	pextx.u8.e rz, rx
operation	<p>Rz+1[31:16] = zero_extend(Rx[31:24])</p> <p>Rz+1[15:0] = zero_extend(Rx[15:8])</p> <p>Rz[31:16] = zero_extend(Rx[23:16])</p> <p>Rz[15:0] = zero_extend(Rx[7:0]) Only exists for</p>
Compilation results	32-bit instructions. pextx.u8.e rz, rx

illustrate :	Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform unsigned extension on byte 3, byte 1, byte 2, and byte 0 in sequence. The extended data format is unsigned half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence. No effect
	none
	none
Affected flags: Restrictions: Exceptions:	

32	Bit
	<p>Rz+1[31:16] = zero_extend(Rx[31:24]) Instruction operation: Rz[15:0] = zero_extend(Rx[15:8]) Rz[31:16] = zero_extend(Rx[23:16]) Rz[15:0] = zero_extend(Rx[7:0])</p>
	pextx.u8.e rz, rx
	<p>Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform unsigned extension on byte 3, byte 1, byte 2, and byte 0 in sequence. The extended data format is unsigned half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence. No effect</p>
Affected	flags:
limit	none
Control: Abnormal:	none

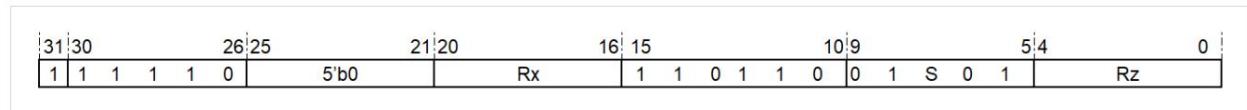
Instruction format:

Figure 15.65: PEXTX.U8.E

15.66 PEXTX.S8.E – 8-Bit Parallel Signed Interleave Extend Instruction

Unified	
instruction syntax	peextx.s8.e rz, rx
operation	<p>Rz+1[31:16] = sign_extend(Rx[31:24])</p> <p>Rz+1[15:0] = sign_extend(Rx[15:8])</p> <p>Rz[31:16] = sign_extend(Rx[23:16])</p> <p>Rz[15:0] = sign_extend(Rx[7:0]) Only exists for</p>
Compilation results	32-bit instructions. peextx.s8.e rz, rx

illustrate :	Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform sign extension on byte 3, byte 1, byte 2, and byte 0 in sequence. The extended data format is a signed half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence.
	No effect
	none
Affected flags: Restrictions: Exceptions:	none

32	
Bit	
	<p>Rz+1[31:16] = sign_extend(Rx[31:24]) Instruction operation: Rz+1[15:0] = sign_extend(Rx[15:8]) Rz[31:16] = sign_extend(Rx[23:16]) Rz[15:0] = sign_extend(Rx[7:0])</p>
	pextx.s8.e rz, rx
	<p>Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0, respectively. Perform sign extension on byte 3, byte 1, byte 2, and byte 0 in sequence. The extended data format is a signed half-word, and the result is stored in the upper half-word of Rz+1, the lower half-word of Rz+1, the upper half-word of Rz, and the lower half-word of Rz in sequence. No</p>
	effect
Affected	flags:
limit	none
	none
Control: Abnormal:	

Instruction format:

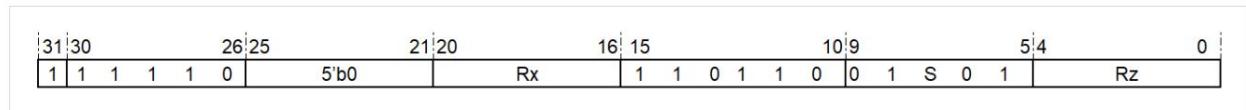


Figure 15.66: PEXTX.S8.E

15.67 NARL——Low-level intercept assembly instruction

The unified	
	narl rz, rx, ry
	Rz[31:0] = {Ry[23:16], Ry[7:0], Rx[23:16], Rx[7:0]}
instruction syntax	operation compilation
	result only has 32-bit instructions. narl rz, rx, ry

Description:	Define the upper halfword and lower halfword of Ry. The upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the lower 8 bits of halfwords 3, 2, 1, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0]
Affected	respectively. No effect
flag:	
Restriction:	
None Exception: None	

32-bit	
instruction operation: Rz[31:0] = { Ry[23:16], Ry[7:0], Rx[23:16], Rx[7:0] } Syntax: narl	
rz, rx, ry Description: Define the upper halfword and lower halfword of Ry, and the upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the lower 8 bits of halfwords 3, 2, 1, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0]	
Affected	respectively. No effect
flag:	
Restriction:	
None Exception: None	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 1 0	1 0 x 0 0	Rz	

Figure 15.67: NARL

15.68 NARH—High-order interception and splicing instructions

The unified	
	narh rz, rx, ry
	Rz[31:0] = {Ry[31:24], Ry[15:8], Rx[31:24], Rx[15:8]}
instruction syntax operation compilation result	only has 32-bit instructions. narh rz, rx, ry

Description:	Define the upper halfword and lower halfword of Ry. The upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the upper 8 bits of halfwords 3, 2, 1, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0]
Affected	respectively. No effect
flag:	
Restriction:	
None Exception: None	

32-bit	
instruction operation: Rz[31:0] = {Ry[31:24], Ry[15:8], Rx[31:24], Rx[15:8]} Syntax: narh	
rz, rx, ry Description: Define the upper halfword and lower halfword of Ry, and the upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the upper 8 bits of halfwords 3, 2, 1, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0]	
Affected	respectively. No effect
flag:	
Restriction:	
None Exception: None	

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 1 0	1 0 x 0 1	Rz	

Figure 15.68: NARH

15.69 NARLX——Low-order cross-interception and splicing instructions

The unified	
	narlx rz, rx, ry
	Rz[31:0] = {Ry[23:16], Rx[23:16], Ry[7:0], Rx[7:0]}
instruction syntax operation compilation result	only has 32-bit instructions. narlx rz, rx, ry

Description:	Define the upper halfword and lower halfword of Ry. The upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the lower 8 bits of halfwords 3, 1, 2, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0]
Affected flag:	respectively. No effect
Restriction:	
None Exception:	None

32-bit	
instruction operation:	Rz[31:0] = {Ry[23:16], Rx[23:16], Ry[7:0], Rx[7:0]} Syntax:
narlx rz, rx, ry	Description: Define the upper halfword and lower halfword of Ry, and the upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the lower 8 bits of halfwords 3, 1, 2, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and
Affected flag:	Rz[7:0] respectively. No effect
Restriction:	
None Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 1 0	1 0 x 1 0	Rz	

Figure 15.69: NARLX

15.70 NARHX—High-order cross-interception and splicing instructions

The unified	
	narhx rz, rx, ry
	Rz[31:0] = {Ry[31:24], Rx[31:24], Ry[15:8], Rx[15:8]}
Instruction syntax	operation compilation result only has 32-bit instructions. narhx rz, rx, ry

Description:	Define the upper halfword and lower halfword of Ry. The upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the upper 8 bits of halfwords 3, 1, 2, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0]
Affected flag:	respectively. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:0] = {Ry[31:24], Rx[31:24], Ry[15:8], Rx[15:8]} Syntax: narhx rz, rx, ry
Description:	Define the upper halfword and lower halfword of Ry, and the upper halfword and lower halfword of Rx are halfwords 3, 2, 1, and 0 respectively. Intercept the upper 8 bits of halfwords 3, 1, 2, and 0 and store them in Rz[31:24], Rz[23:16], Rz[15:8], and Rz[7:0] respectively. No effect
Affected flag:	
Restriction:	None
Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 1 0 1 1 0	1 0 x 1 1	Rz	

Figure 15.70: NARHX

15.71 CLIP.(U/S)32——Immediate (unsigned/signed) clip and saturate instruction

Unified	
instruction syntax	clipi.u32 rz, rx, imm5 clipi.s32 rz, rx, oimm5
Operation	<p>Max = $2(\text{imm5}) - 1$, Min = 0 //unsigned</p> <p>Max = $2(\text{oimm5}-1) - 1$, Min = $-2(\text{oimm5}-1)$ //signed if($\text{Rx}[31:0] > \text{Max}$)</p> <p>$\text{Rz}[31:0] = \text{Max}$ else if($\text{Rx}[31:0] < \text{Min}$) $\text{Rz}[31:0] = \text{Min}$ else $\text{Rz}[31:0] = \text{Rx}[31:0]$ Only 32-</p>
Compilation results	bit instructions exist. clipi.u32 rz, rx, imm5 clipi.s32 rz, rx, oimm5

explain	Save Rx in the target data specified by IMM5/OIMM5, where IMM5/OIMM5 represents the bit width of the target data. If the value of Rx exceeds the numerical range that the target data can represent, saturation processing is performed. For unsigned CLIP instructions, the upper saturation value is $2^{\text{IMM5}} - 1$ and the lower saturation value is 0x0. If Rx is greater than the upper saturation value, the result is $2^{\text{IMM5}} - 1$, otherwise the result is Rx. For the signed CLIP instruction, OIMM5 is an immediate value with a bias of 1, the upper saturation value is $2^{\text{(OIMM5}-1)} - 1$, and the lower saturation value is $-2^{\text{(OIMM5}-1)}$. If Rx is greater than the upper saturation value, the result is $2^{\text{(OIMM5}-1)} - 1$, if Rx is less than the lower saturation value, the result is $-2^{\text{(OIMM5}-1)}$, otherwise the result is Rx. No
effect	
	The range of IMM5 is 0~31 OIMM5 range is 1~32 None
Affected	flags: Restrictions: Exceptions:

32	Bit
	If(unsigned)Max = 2(imm5)-1, Min = 0 // instruction of signed Max = 2(oimm5-1)-1, Min = -2(oimm5-1) // if(Rx[31:0] > Max) Rz[31:0] = Max else if(Rx[31:0] < Min) Rz[31:0] = Min else Rz[31:0] = Rx[31:0] clipi.u32
	rz, rx, imm5 clipi.s32 rz, rx, oimm5 Save the Rx bitstream
Syntax : Description	to the target data specified by IMM5/OIMM5, where IMM5/OIMM5 represents the bit width of the target data. If the value of Rx exceeds the numerical range that the target data can represent, saturation is performed. For unsigned CLIP instructions, the upper saturation value is $2^{\text{IMM5}}-1$ and the lower saturation value is 0x0. If Rx is greater than the upper saturation value, the result is $2^{\text{IMM5}}-1$, otherwise the result is Rx. For the signed CLIP instruction, OIMM5 is an immediate value with a bias of 1, the upper saturation value is $2^{\text{(OIMM5)}}-1$, and the lower saturation value is $-2^{\text{(OIMM5)}}+1$. If Rx is greater than the upper saturation value, the result is $2^{\text{(OIMM5)}}-1$, if Rx is less than the lower saturation value, the result is $-2^{\text{(OIMM5)}}+1$, otherwise the result is
	Rx. No effect
	The range of IMM5 is 0~31 OIMM5 range is 1~32 None
Affected flags: Restrictions: Exceptions:	

Instruction format:

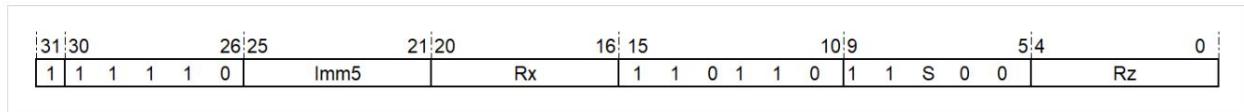


Figure 15.71: CLIP.(US)32

15.72 CLIP.(U/S)32——(Unsigned/Signed) Clip Saturation Instruction

Unified	
instruction syntax	clip.u32 rz, rx, ry clip.s32 rz, rx, ry Operation
	<pre>Max = 2(Ry[4:0])-1 Min = 0 //unsigned Max = 2(Ry[4:0]-1)-1 Min = -2(Ry[4:0]-1) //signed if(Rx[31:0] > Max) Rz[31:0] = Max else if(Rx[31:0] < Min) Rz[31:0] = Min else Rz[31:0] = Rx[31:0] Only 32-</pre>
Compilation result	bit instructions exist. clip.u32 result\$tx, ry clip.s32 rz, rx, ry

explain	Save Rx in the target data specified by Ry[4:0], where Ry[4:0] represents the bit width of the target data. If the value of Rx exceeds the numerical range that the target data can represent, saturation processing is performed. For unsigned CLIP instructions, the upper saturation value is $2^{Ry[4:0]}-1$, and the lower saturation value is 0x0. If Rx is greater than the upper saturation value, the result is $2^{Ry[4:0]}-1$, otherwise the result is Rx. For the signed CLIP instruction, the upper saturation value is $2^{(Ry[4:0]-1)}-1$, and the lower saturation value is $-2^{(Ry[4:0]-1)}$. If Rx is greater than the upper saturation value, the result is $2^{(Ry[4:0]-1)}-1$, if Rx is less than the lower saturation value, the result is $-2^{(Ry[4:0]-1)}$, otherwise the result is Rx. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32	Bit
	<p>Max = $2(Ry[4:0]-1)$ Min = 0 //unsigned instruction of Max = $2(Ry[4:0]-1)$ Min = $-2(Ry[4:0]-1)$ //signed if($Rx[31:0] > Max$)</p> <pre> Rz[31:0] = Max else if($Rx[31:0] < Min$) Rz[31:0] = Min else Rz[31:0] = Rx[31:0] clip.u32 </pre>
	rz, rx, ry clip.s32 rz, rx, ry Save Rx in the target
Syntax : Described	data specified by Ry[4:0], where Ry[4:0] represents the bit width of the target data. If the value of Rx exceeds the numerical range that the target data can represent, saturation is performed. For unsigned CLIP instructions, the upper saturation value is $2^{Ry[4:0]-1}$ and the lower saturation value is 0x0. If Rx is greater than the upper saturation value, the result is $2^{Ry[4:0]-1}$, otherwise the result is Rx. For the signed CLIP instruction, the upper saturation value is $2^{(Ry[4:0]-1)-1}$, and the lower saturation value is $-2^{(Ry[4:0]-1)}$. If Rx is greater than the upper saturation value, the result is $2^{(Ry[4:0]-1)-1}$, if Rx is less than the lower saturation value, the result is $-2^{(Ry[4:0]-1)}$, otherwise the result
	is Rx. No effect
	none
	none
Affected flags: Restrictions: Exceptions:	

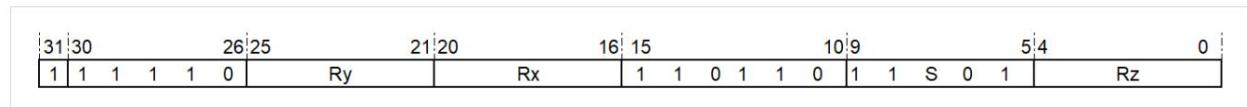
Instruction format:

Figure 15.72: CLIP.(US)32

15.73 PCLPI.(U/S)16——16-bit parallel immediate value (without/with) sign clipping and saturation

instruction

System one	
	<p>pclip.u16 rz, rx, imm4 pclip.s16</p> <p>rz, rx, oimm4 Max = 2(imm4)-1,</p>
Optimize inst2(pclip.u16)//signed if(Rx[31:16] Rx[15:0] > Max)	<p>Min = 0 //unsigned Max = 2(oimm4-1)-1, Min =</p> <p>Rz[31:16] Rz[15:0] = Max else</p> <p>if(Rx[31:16] Rx[15:0] < Min)</p> <p>Rz[31:16] Rz[15:0] = Min</p> <p>else</p> <p>Rz[31:16] Rz[15:0] = Rx[31:16] Rx[15:0] Only 32-bit</p>
Compilation results	<p>instructions exist. pclip.u16</p> <p>rx, rx, imm4 pclip.s16 rx, rx,</p> <p>oimm4</p>

Illustrate : represents the bit width of the target data. If the value of the original data exceeds the numerical range that the target data can represent, saturation processing is performed. For the unsigned CLIP instruction, the upper saturation value is $2^{\text{IMM4}} - 1$, and the lower saturation value is 0x0. If the original data is greater than the upper saturation value, the result is $2^{\text{IMM4}} - 1$, otherwise the result is the original data itself. For the signed CLIP instruction, OIM M4 is an immediate value with offset 1, the upper saturation value is $2^{(\text{OIMM4}-1)} - 1$, and the lower saturation value is $-2^{(\text{OIMM4}-1)}$. If the original data is greater than the upper saturation value, the result is $2^{(\text{OIMM4}-1)} - 1$, if the original data is less than the lower saturation value, the result is $-2^{(\text{OIMM4}-1)}$, otherwise the result is the original data	
	itself. No impact
	The range of Imm4 is 0~15 Oimm4 ranges from 1 to 16.
Affected	flags: Restrictions: Exceptions:

32	Bit
	<p>Max = $2(\text{imm4}) - 1$, Min = 0 //unsigned Instruction operation: Max = $2(\text{oimm4}-1)$, Min = $-2(\text{oimm4}-1)$ //signed if($\text{Rx}[31:16] \& \text{Rx}[15:0] > \text{Max}$) $\text{Rz}[31:16] \& \text{Rz}[15:0] = \text{Max}$ else if($\text{Rx}[31:16] \& \text{Rx}[15:0] < \text{Min}$) $\text{Rz}[31:16] \& \text{Rz}[15:0] = \text{Min}$ else $\text{Rz}[31:16] \& \text{Rz}[15:0] = \text{Rx}[31:16] \& \text{Rx}[15:0]$ pclip.u16 rx, rx,</p>
	<p>imm4 pclip.s16 rx, rx, oimm4 The upper and lower halfwords of Rx</p>
Syntax : Dest	<p>are used as the original data and are shrunk and saved in the target data specified by IMM4/OIMM4, where IMM4/O IMM4 represents the bit width of the target data. If the value of the original data exceeds the numerical range that the target data can represent, saturation processing is performed. For unsigned CLIP instructions, the upper saturation value is $2^{\text{IMM4}} - 1$ and the lower saturation value is 0x0. If the original data is greater than the upper saturation value, the result is $2^{\text{IMM4}} - 1$, otherwise the result is the original data itself.</p> <p>For signed CLIP instructions, OIMM4 is an immediate value with offset 1, the upper saturation value is $2^{\text{OIMM4}} - 1$, and the lower saturation value is $-2^{\text{OIMM4}} - 1$. If the original data is greater than the upper saturation value, the result is $2^{\text{OIMM4}} - 1$, if the original data is less than the lower saturation value, the result is $-2^{\text{OIMM4}} - 1$, otherwise the result is the original data.</p>
	<p>data itself. No impact</p>
	<p>The range of Imm5 is 0~15 Oimm5 ranges from 1 to 16.</p>
Affected	flags: Restrictions: Exceptions:

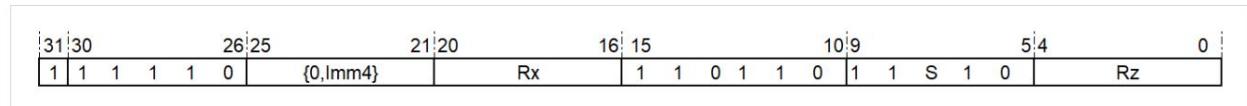
Instruction format:

Figure 15.73: PCLIP1.(US)16

15.74 PCLIP.(U/S)16——16-bit parallel (un/with) signed clipping and saturation instruction

System one	
	<pre>pclip.u16 rz, rx, ry pclip.s16 rz, rx, ry Max = 2(Ry[3:0])-1, Min = 0 //</pre>
Optimize instruction syntax	<pre>unsigned Max = 2(Ry[3:0]-1), Min = -2(Ry[3:0]-1) //signed if(Rx[31:16]) Rz[31:16] Rx[15:0] = Max else if(Rx[31:16]) Rx[15:0] <Min) Rz[31:16] Rx[15:0] = Min else Rz[31:16] Rx[15:0] = Rx[31:16] Rx[15:0] Only 32-bit instructions exist.</pre>
Compilation results	<pre>pclip.u16 rx, ry pclip.s16 rx, ry</pre>

explain bright:	The upper and lower half words of Rx are used as the original data and are saved in the target data specified by Ry[3:0], where Ry[3:0]4 represents the bit width of the target data. If the value of the original data exceeds the numerical range that the target data can represent, saturation processing is performed. For the unsigned CLIP instruction, the upper saturation value is $2^{\text{Ry}[3:0]}-1$, and the lower saturation value is 0x0. If the original data is greater than the upper saturation value, the result is $2^{\text{Ry}[3:0]}-1$. In other cases, the result is the original data itself. For the signed CLIP instruction, the upper saturation value is $2^{\text{Ry}[3:0]-1}-1$, and the lower saturation value is $-2^{\text{Ry}[3:0]-1}$. If the original data is greater than the upper saturation value, the result is $2^{\text{Ry}[3:0]-1}-1$. If the original data is less than the lower saturation value, the result is $-2^{\text{Ry}[3:0]-1}$. In other cases, the result is the original data itself. No impact
	none
Affected flags: Restrictions: Exceptions:	none

32	Bit
	<p>Max = $2(Ry[3:0])-1$, Min = 0 //unsigned Instruction operation: Max = $2(Ry[3:0])-1$, Min = $-2(Ry[3:0]-1)$ //signed if($Rx[31:16] Rx[15:0] > Max$) $Rz[31:16] Rz[15:0] = Max$ else if($Rx[31:16] Rx[15:0] < Min$) $Rz[31:16] Rz[15:0] = Min$ else $Rz[31:16] Rz[15:0] = Rx[31:16] Rx[15:0]$ pclip.u16 rx, rx, ry</p>
	pclip.s16 rx, rx, ry The upper and lower halfwords of Rx
	<p>are used as the original data and are shrunk and saved in the target data specified by Ry[3:0], where Ry[3:0]4 represents the bit width of the target data. Syntax : Dest Rz Ry Rx If the value of the original data exceeds the numerical range that the target data can represent, saturation processing is performed. For unsigned CLIP instructions, the upper saturation value is $2^Ry[3:0]-1$ and the lower saturation value is 0x0. If the original data is greater than the upper saturation value, the result is $2^Ry[3:0]-1$, otherwise the result is the original data itself.</p> <p>For signed CLIP instructions, the upper saturation value is $2^{(Ry[3:0]-1)}-1$ and the lower saturation value is $-2^{(Ry[3:0]-1)}$. If the original data is greater than the upper saturation value, the result is $2^{(Ry[3:0]-1)}-1$. If the original data is less than the lower saturation value, the result is $-2^{(Ry[3:0]-1)}$. In other cases, the result is the</p>
	original data itself. No effect
	none
	none
	Affected flags: Restrictions: Exceptions:

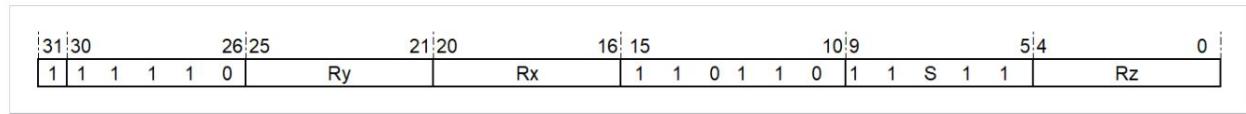
Instruction format:

Figure 15.74: PCLIP.(US)16

15.75 PABS.S8.S – 8-bit Parallel Absolute Value Instruction with Saturation

Unified	
instruction syntax	
operation pabs.s8.s Rz[31:24] = Saturate(abs(Rx[31:24]))	
Rz, rx	Rz[23:16] = Saturate(abs(Rx[23:16])) Rz[15:8] = Saturate(abs(Rx[15:8])) Rz[7:0] = Saturate(abs(Rx[7:0]))
The compiled result contains only 32-bit instructions. pabs.s8.s rz, rx	

Description:	Take the absolute value of the bytes of Rx in units of bytes and perform saturation processing. When the byte is 0x80, the absolute value is saturated. The result of the processing is 0x7F, and the rest is the same as the absolute value
Affected flag:	result. No impact
Restriction:	None
Exception:	None

32-bit	
instruction operation: Rz[31:24] = Saturate(abs(Rx[31:24])) Rz[23:16] = Saturate(abs(Rx[23:16])) Rz[15:8] = Saturate(abs(Rx[15:8])) Rz[7:0] = Saturate(abs(Rx[7:0])) Syntax:	
pabs.s8.s rz, rx	Description: In bytes,
	take the absolute value of the bytes of Rx in turn and perform saturation processing. When the byte is 0x8000, the absolute value is saturated. The result of the processing is 0xFFFF, and the rest is the same as the absolute value result.
Affected flag:	
Restriction: None	
Exception: None	

Instruction format:

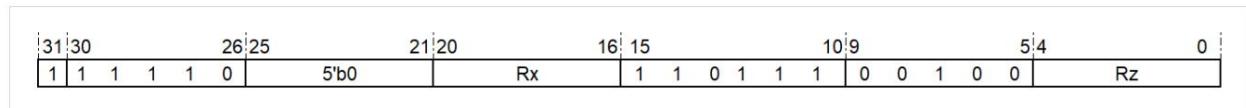


Figure 15.75: PABS.S8.S

15.76 PABS.S16.S – 16-bit Parallel Absolute Value Instruction with Saturation

Unified command	
syntax	pabs.s16.s rz, rx Rx[31:16]
operation	= Saturate(abs(Rx[31:16])) The compiled result of Rx[15:0] = Saturate(abs(Rx[15:0]))
only contains 32-bit instructions.	pabs.s16.s rz, rx

Description: Take the absolute value of the upper and lower half words of Rx in turn and perform saturation processing. When the half word is 0x8000, the result of the absolute value saturation processing is If the result is 0x7FFF, the rest is the same as the absolute value
Affected flag:
Restriction: None
Exception: None

32-bit	
instruction operation:	Rz[31:16] = Saturate(abs(Rx[31:16])) Rz[15:0] = Saturate(abs(Rx[15:0])) Syntax: pabs.s16.s
rz, rx Description:	Take the absolute value of the upper and lower half words of Rx in turn and perform saturation processing. When the half word is 0x8000, the result of the absolute value saturation processing is If the result is 0x7FFF, the rest is the same as the absolute value
Affected flag:	result. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	5'b0	Rx	1 1 0 1 1 1 0 0 1 0 1	Rz		

Figure 15.76: PABS.S16.S

15.77 ABS.S32.S – 32-bit Absolute Value Instruction with Saturation

The unified	
	abs.s32.s rz, rx Rz[31:0]
	= Saturate(abs(Rx[31:0]))
instruction syntax operation compilation result	only has 32-bit instructions. abs.s32.s rz, rx

Description:	Take the absolute value of Rx and perform saturation processing. When Rx is 0x8000 0000, the result of the absolute value saturation processing is 0x7FFF FFFF, the rest are the same as the absolute value results.
Affected flag:	No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:0] = Saturate(abs(Rx[31:0]))
Syntax:	
abs.s32.s rz, rx	Description: Take the absolute value of Rx and perform saturation processing. When Rx is 0x8000 0000, the result of the absolute value saturation processing is 0x7FFF FFFF, the rest are the same as the absolute value results.
Affected flag:	No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	5'b0	Rx	1 1 0 1 1 1 0 0 1 1 0	Rz		

Figure 15.77: ABS.S32.S

15.78 PNEG.S8.S – 8-Bit Parallel Negate Instruction with Saturation

Unified	
instruction syntax	pneg.s8.s rz, rx
operation	Rz[31:24] = Saturate(neg(Rx[31:24])) Rz[23:16] = Saturate(neg(Rx[23:16])) Rz[15:8] = Saturate(neg(Rx[15:8])) Rz[7:0] = Saturate(neg(Rx[7:0])) Only exists
Compilation results	for 32-bit instructions. pneg.s8.s rz, rx

32-bit	
instruction operation:	Rz[31:24] = Saturate(neg(Rx[31:24])) Rz[23:16] = Saturate(neg(Rx[23:16])) Rz[15:8] = Saturate(neg(Rx[15:8])) Rz[7:0] = Saturate(neg(Rx[7:0])) Syntax:
pneg.s8.s rz, rx	Description: In byte units, the bytes of Rx are inverted and saturated. When the byte is 0x80, the result of the inverted saturation process is 0x7F, and the rest is the same as the inverted result. No effect
Affected flag:	
Restriction:	None
Exception:	None

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	1	1	1	1	0	5'b0	Rx	1	1	0	1	Rz

Figure 15.78: PNEG.S8.S

15.79 PNEG.S16.S – 16-Bit Parallel Negation Instruction with Saturation

Unified command	
syntax	pneg.s16.s rz, rx
operation	Rz[31:16] = Saturate(neg(Rx[31:16])) Rz[15:0] = Saturate(neg(Rx[15:0])) The compiled
result only has 32-bit instructions.	pneg.s16.s rz, rx

Description:	Invert the upper and lower halfwords of Rx in turn and perform saturation processing. When the halfword is 0x8000, the result of the inverted saturation processing is 0x7FFF, the rest are the same as the inverted result.
Affected flag:	No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	Rz[31:16] = Saturate(neg(Rx[31:16])) Rz[15:0] = Saturate(neg(Rx[15:0])) Syntax:
pneg.s16.s rz, rx	Description: Negate
	the upper and lower halfwords of Rx in turn and perform saturation processing. When the halfword is 0x8000, the result of the negation and saturation processing is 0x7FFF, the rest are the same as the inverted result.
Affected flag:	No effect
Restriction:	None
Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	5'b0	Rx	1 1 0 1 1 1	0 1 1 0 1	Rz	

Figure 15.79: PNEG.S16.S

15.80 NEG.S32.S – Negation with Saturation

The unified	
	neg.s32.s rz, rx
	Rz[31:0] = Saturate(neg(Rx[31:0]))
Instruction syntax operation compilation result only has 32-bit instructions. neg.s32.s rz, rx	

Description: Invert Rx and perform saturation processing. When Rx is 0x8000 0000, the result of inverted saturation processing is 0x7FFF FFFF, and the rest is the same as the absolute value
Affected flag: result. No effect
Restriction: None
Exception: None

32-bit	
instruction operation: Rz[31:0] = Saturate(neg(Rx[31:0]))	Syntax:
neg.s32.s rz, rx	Description: Negate
Rx and perform saturation processing. When Rx is 0x8000 0000, the result of the negated saturation processing is 0x7FFF FFFF, and the rest is the same as the absolute value result. No effect	
Affected flag:	
Restriction: None	
Exception: None	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	5'b0	Rx	1 1 0 1 1 1 0 1 1 0	Rz		

Figure 15.80: NEG.S32.S

15.81 DUP.8——8-bit operand copy instruction

Unified	
Instruction syntax	dup.8 rz, rx, index
Operation	Element = Rx[7:0] //index= 0 Element = Rx[15:8] //index= 1 Element = Rx[23:16] // index= 2 Element = Rx[31:24] //index= 3 Rz[31:0] = {Element, Element, Element, Element } Only 32-bit instructions exist. dup.8 rz, rx, index
Compilation results	

Description:	Define Rx[31:24], Rx[23:16], Rx[15:8], and Rx[7:0] as byte 3, byte 2, byte 1, and byte 0 respectively. Select the corresponding byte and copy it to the 4 bytes of Rx. No
Affected flag:	effect
Restriction:	Index range is 0~3
Exception:	None

32-bit	
instruction operation:	Element = Rx[7:0] //index= 0 Element = Rx[15:8] //index = 1 Element = Rx[23:16] //index= 2 Element = Rx[31:24] //index= 3 Rz[31:0] = { Element, Element, Element, Element }
Syntax:	dup.8 rz, rx, index Description:
	Define Rx[31:24], Rx[23:16], Rx[15 :8], Rx[7:0] as byte 3, byte 2, byte 1, and byte 0 respectively. Select the corresponding byte and copy it to the 4 bytes of Rx. No
Affected flag :	effect
Restriction:	Index range is 0~3
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	5'b0	Rx	1 1 0 1 1 1	1 0 x	index	Rz

Figure 15.81: DUP.8

15.82 DUP.16—16-bit operand copy instruction

Unified	
instruction syntax	dup.16 rz, rx, index
operation	Element = Rx[15:0] //index= 0 Element = Rx[31:16] // index= 1 Rz[31:0] = { Element, Element }
The compiled result	contains only 32-bit instructions. dup.16 rz, rx, index

Description:	Define Rx[31:16] and Rx[15:0] as halfword 1 and halfword 0 respectively. Select the corresponding halfword by index and copy it to the 2 halfwords of Rx.
Affected flags:	No effect
	Index ranges from 0 to 1
Restrictions: Exceptions:	None

32-bit instruction	
operation:	Element = Rx[15:0] //index= 0 Element = Rx[31:16] //index= 1 Rz[31:0] = { Element, Element } dup.16 rz, rx, index
Syntax:	
Description:	Define Rx[31:16] and Rx[15:0] as halfword 1 and halfword 0 respectively. Select the corresponding halfword by index and copy it to Rx No effect
Affected flags:	
	Index ranges from 0 to 1
Restrictions: Exceptions:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	5'b0	Rx	1 1 0 1 1 1	1 1 x index	Rz	

Figure 15.82: DUP.16

15.83 MULS.(U/S)32——32-bit (unsigned/signed) multiply-accumulate-subtract instruction

Unified instructions	
grammar	muls.u32 rz, rx, ry muls.s32 rz, rx, ry
Operation {Rz+1[31:0], Rz[31:0]} = {Rz+1[31:0], Rz[31:0]} - Rx[31:0] X Ry[31:0]	
The compiled result contains only 32-bit instructions.	muls.u32 rz, rx, ry muls.s32 rz, rx, ry

illustrate:	Rx is multiplied by Ry to obtain a 64-bit result, which is then subtracted from {Rz+1, Rz}. The upper 32 bits of the subtraction result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

32-bit instructions	
	{Rz+1[31:0], Rz[31:0]} = {Rz+1[31:0], Rz[31:0]} - Rx[31:0] X Ry[31:0]
Operation: Syntax:	muls.u32 rz, rx, ry muls.s32 rz, rx, ry
illustrate:	Rx is multiplied by Ry to obtain a 64-bit result, which is then subtracted from {Rz+1, Rz}. The upper 32 bits of the subtraction result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0	S 0 1 1 0	Rz	

Figure 15.83: MULS.(US)32

15.84 MULA.(U/S)32.S -- 32-bit (un/signed) multiply-accumulate with saturation

instruction

Unified command	
syntax	mula.u32.s rz, rx, ry mula.s32.s rz, rx, ry {Rz+1[31:0],Rz[31:0]}
The	= Saturate({Rz+1[31:0],Rz[31:0]} + Rx[31:0] X Ry[31:0])
compiled result only	contains 32-bit instructions. mula.u32.s rz, rx, ry mula.s32.s rz, rx, ry

illustrate	Rx is multiplied by Ry to obtain a 64-bit result, which is then added to {Rz+1, Rz}. After the addition result is saturated, the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz. For unsigned operations, the saturation process is as follows: if the addition result is greater than the saturation value 0xFFFF FFFF FFFF FFFF, the result is the saturation value; otherwise, it is the addition result itself. For signed operations, the saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF FFFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000 0000 0000, the result is the lower saturation value; otherwise, it is the addition result itself. No effect
	none
Affected	flags: Restrictions: Exceptions:

32	
Bit	
	$\{Rz+1[31:0], Rz[31:0]\} = \text{Saturate}(\{Rz+1[31:0], Rz[31:0]\} + Rx[31:0] \times Ry[31:0])$
	mula.u32.s rz, rx, ry mula.s32.s rz, rx, ry Multiply Rx and Ry to
	get a 64-bit result, and add it to $\{Rz+1, Rz\}$. After the addition result is saturated, the upper 32 bits are stored in $Rz+1$ and the lower 32 bits are stored in Rz . For signed operations, the saturation process is as follows: if the addition result is greater than the saturation value 0xFFFF FFFF FFFF FFFF, the result is the saturated value; otherwise, it is the addition result itself. For signed operations, the saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF FFFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000 0000 0000, the result is the lower saturation value; otherwise, it is the addition result itself. No effect
	none
	none
Affected	flags: Restrictions: Exceptions:

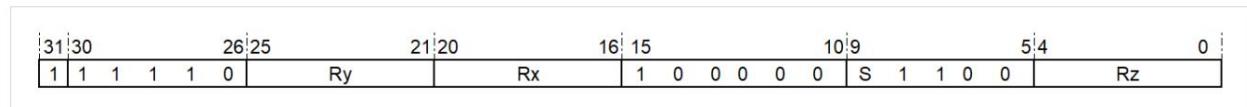
Instruction format:

Figure 15.84: MULA.(US)32.S

15.85 MULS.(U/S)32.S -- 32-bit (un/signed) multiply-accumulate-subtract with saturation

instruction

Unified command	
syntax	muls.u32.s rz, rx, ry muls.s32.s rz, rx, ry {Rz+1[31:0],Rz[31:0]} =
The	Saturate({Rz+1[31:0],Rz[31:0]} - Rx[31:0] X Ry[31:0])
compiled result only contains 32-bit instructions. muls.u32.s rz, rx, ry muls.s32.s rz, rx, ry	

illustrate	Rx is multiplied by Ry to obtain a 64-bit result, which is then subtracted from {Rz+1, Rz}. After the subtraction result is saturated, the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz. For unsigned operations, the saturation process is as follows: if the subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise it is the subtraction result itself. For signed operations, the saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF FFFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000 0000 0000, the result is the lower saturation value, otherwise it is the subtraction result itself. No effect
	none
Affected	flags: Restrictions: Exceptions:

32 Bit	
	$\{Rz+1[31:0], Rz[31:0]\} = \text{Saturate}(\{Rz+1[31:0], Rz[31:0]\} - Rx[31:0] \times Ry[31:0])$
	muls.u32.s rz, rx, ry muls.s32.s rz, rx, ry Multiply Rx by Ry to
	get a 64-bit result, and subtract it from {Rz+1, Rz}. After the subtraction result is saturated, the upper 32 bits are stored in Rz+1 and the lower 32 bits are stored in Rz. For unsigned operations, the saturation process is as follows: if the subtraction result is less than the saturation value 0x0, the result is 0x0, otherwise it is the subtraction result itself. For signed operations, the saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF FFFF FFFF, the result is the upper saturation value, if the subtraction result is less than the lower saturation value 0x8000 0000 0000 0000, the result is the lower saturation value, otherwise it is the subtraction result itself. No effect
	none
Affected flags: Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0 0	S 1 1 1 0	Rz	

Figure 15.85: MULS.(US)32.S

15.86 MUL.S32.H——32-bit signed multiplication with high 32 bits

Unified instruction	
syntax muls32.h rz, rx, ry operation Rz[31:0] =	
{Rx[31:0] X Ry[31:0]}[63:32] Compiled result Only 32-bit instructions exist.	
	mul.s32.h rz, rx, ry

	Rx is multiplied by Ry to obtain a 64-bit result, of which the upper 32 bits are stored in Rz.
Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

32-bit instructions	
	Rz[31:0] = {Rx[31:0] X Ry[31:0]}[63:32]
	mul.s32.h rz, rx, ry
	Rx is multiplied by Ry to obtain a 64-bit result, of which the upper 32 bits are stored in Rz.
Operation: Syntax: Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

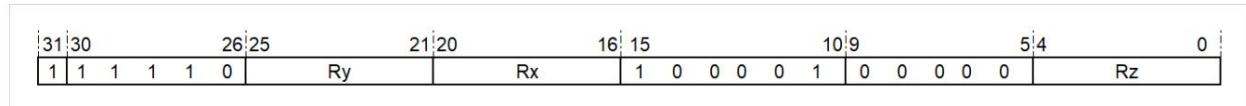
Instruction format:

Figure 15.86: MUL.S32.H

15.87 MUL.S32.RH -- 32-bit signed multiplication with rounding

make

Unified instructions	
Syntax	mul.s32.rh rz, rx, ry
	Rz[31:0] = (Rx[31:0] X Ry[31:0] + 32'h8000 0000)[63:32]
Operation Compiled results	contain only 32-bit instructions. mul.s32.rh rz, rx, ry

Explanation:	The 64-bit result obtained by multiplying Rx and Ry is added to 0x8000 0000, and the upper 32 bits of the addition result are intercepted and stored in Rz. In the example, 0x8000 0000 is added to the multiplication result to implement the rounding operation.
Impact Mark	No impact
Position:	
Restrictions:	None
Abnormal:	None

32-bit	
instruction operation:	Rz[31:0] = {Rx[31:0] X Ry[31:0] + 32'h8000 0000}[63:32] Syntax: mul.s32.rh
rz, rx, ry Description:	The 64-bit result of multiplying Rx and Ry is added to 0x8000 0000, and the upper 32 bits of the addition result are intercepted and stored in Rz.
	In the multiplication, 0x8000 0000 is added to the result of the multiplication to implement the rounding
Affected flag:	operation. No effect
Restriction:	None
Exception:	None

Instruction format:

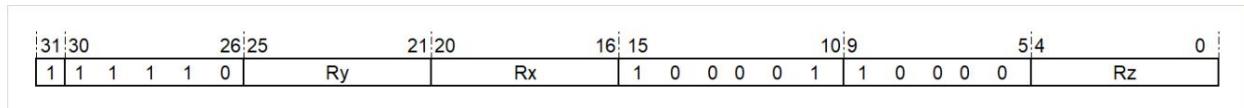


Figure 15.87: MUL.S32.RH

15.88 RMUL.S32.H—32-bit signed fractional multiplication and high 32-bit instruction

Unified	
instruction syntax	rmul.s32.h rz, rx, ry
operation if (Rx[31:0] == 32'h8000 0000 && Ry[31:0] == 32'h8000 0000)	Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:0]}[62:31] Only 32-bit
Compilation results	instructions exist. rmul.s32.h rz, rx, ry

Note: Rx and Ry are regarded as 32-bit signed decimals, where the 31st bit is the sign bit and the 30th to 0th bits are the decimal bits. When Rx=0x8000 0000 and Ry=0x8000 0000, the result is 0x7FFF FFFF; in other cases, the 62nd to 31st bits of the multiplication result of Rx and Ry are stored in Rz.
Affected flag:
Restriction:
None Exception: None

32-bit	
instruction operation: if(Rx[31:0] == 32'h8000 0000 && Ry[31:0] == 32'h8000 0000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:0]}[62:31] Syntax: rmul.s32.h rz, rx, ry Description: Rx	
	and Ry are regarded as 32-bit signed decimals, where the 31st bit is the sign bit and the 30th to 0th bits are the decimal bits. When Rx=0x8000 0000 and Ry=0x8000 0000, the result is 0x7FFF FFFF; in other cases, the 62nd to 31st bits of the multiplication result of Rx and Ry are stored in Rz. No effect
Affected flag:	
Restriction:	
None	Exception: None

Instruction format:

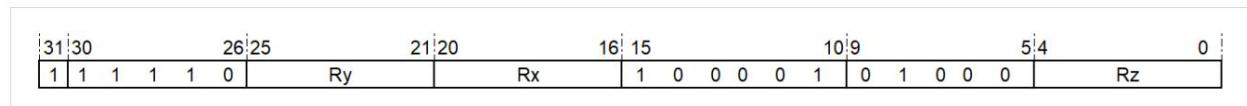


Figure 15.88: RMUL.S32.H

15.89 RMUL.S32.RH -- 32-bit signed fractional multiplication with rounding

Bit Instructions

Unified	
instruction syntax	rmul.s32.rh rz, rx, ry
operation	if(Rx[31:0] == 32'h8000 0000 && Ry[31:0] == 32'h8000 0000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:0] + 32'h4000 0000}[62:31] Only 32-bit instructions
Compilation results	exist. rmul.s32.rh rz, rx, ry

	Illustrate: Rx and Ry are regarded as 32-bit signed decimals, where the 31st bit is the sign bit and the 30th to 0th bits are the decimal bits. When Rx=0x8000 0000 and Ry=0x8000 0000, the result is 0x7FFF FF FF; in other cases, the 64-bit result obtained by multiplying Rx and Ry is added to 0x4000 0000, and the 62nd to 31st bits of the addition result are intercepted and stored in Rz. Among them, the multiplication result plus 0x4000 0000 is used to implement the decimal rounding operation. No impact
Affected flags:	
limit	none
Control: Abnormal:	none

32 Bit	
instruction operation:	if(Rx[31:0] == 32'h8000 0000 && Ry[31:0] == 32'h8000 0000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:0] + 32'h4000 0000}[62:31] rmul.s32.rh rz, rx, ry
Syntax : Description:	Rx and Ry are regarded as 32-bit signed decimals, where the 31st bit is the sign bit and the 30th to 0th bits are the decimal bits. When Rx=0x8000 0000 and Ry=0x8000 0000, the result is 0x7FFF FF FF; in other cases, the 64-bit result obtained by multiplying Rx and Ry is added to 0x4000 0000, and the 62nd to 31st bits of the addition result are intercepted and stored in Rz. Among them, the multiplication result plus 0x4000 0000 is used to implement the decimal rounding operation. No impact
Affected flags: Restrictions: Exceptions:	none
Affected flags: Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0	1 1 0 0 0	Rz	

Figure 15.89: RMUL.S32.RH

15.90 MULA.S32.HS – 32-bit signed multiply-accumulate with saturation

instruction

The unified	
	mula.s32.hs rz, rx, ry Rz[31:0] =
	Saturate(Rz[31:0] + {Rx[31:0] X Ry[31:0]}[63:32])
instruction syntax operation compilation result	only has 32-bit instructions. mula.s32.hs rz, rx, ry

illustrate :	The 64-bit result obtained by multiplying Rx and Ry is taken from the upper 32 bits and added to Rz. The addition result is saturated and stored in Rz. The saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result
	itself. No impact
	none
	none
Affected flags: Restrictions: Exceptions:	

32	
Bit	
	Rz[31:0] = Saturate(Rz[31:0] + {Rx[31:0] X Ry[31:0]}[63:32])
	mula.s32.hs rz, rx, ry
	The 64-bit result obtained by multiplying Rx and Ry is taken from the upper 32 bits and added to Rz. The addition result is saturated and stored in Rz. The saturation instruction operates System follows if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result
	itself. No impact
Affected	flags:
limit	none
	none
Control: Abnormal:	

Instruction format:

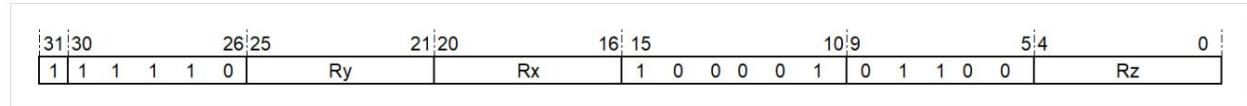


Figure 15.90: MULA.S32.HS

15.91 MULS.S32.HS – 32-bit signed multiply-accumulate subtract high 32 bits with saturation

instruction

The unified	
	muls.s32.hs rz, rx, ry Rz[31:0]
	= Saturate(Rz[31:0] - {Rx[31:0] X Ry[31:0]}[63:32])
instruction syntax operation compilation	result only has 32-bit instructions. muls.s32.hs rz, rx, ry

illustrate :	The 64-bit result obtained by multiplying Rx and Ry is taken from the upper 32 bits and subtracted from Rz. The subtraction result is saturated and stored in Rz. The saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the subtraction result itself. No effect
Impact Mark	
Zhi	
Bit:	
	none
Limitation:	
Exception:	

32	
Bit	
	Rz[31:0] = Saturate(Rz[31:0] - {Rx[31:0] X Ry[31:0]}[63:32])
instruction	muls.s32.hs rz, rx, ry
operation : Syntax :	
illustrate :	The 64-bit result obtained by multiplying Rx and Ry is taken from the upper 32 bits and subtracted from Rz. The subtraction result is saturated and stored in Rz. The saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the subtraction result itself. No effect
Affected flags:	
limit	none
	none
Control: Abnormal:	

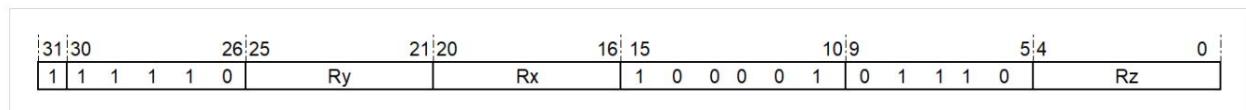
Instruction format:

Figure 15.91: MULS.S32.HS

15.92 MULA.S32.RHS – 32-bit signed multiply-accumulate with rounding and saturation**High 32-bit instruction**

The unified	
	mula.s32.rhs rz, rx, ry
instruction syntax	operation Rz[31:0] = Saturate((Rz[31:0], 32'h0000 0000) + Rx[31:0] X Ry[31:0] + 32'h8000 0000)[63:32]) only has 32-bit instructions in the compilation result .
	mula.s32.rhs rz, rx, ry

illustrate	The 64-bit result of multiplying Rx and Ry is accumulated with {Rz, 32'h0000 0000}, and 0x8000 0000 is added to the accumulated result. The upper 32 bits of the addition processing process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value, if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value, and in other cases, it is the addition result itself. No impact
	none
Affected flags: Restrictions: Exceptions:	none

32 Bit	
	Rz[31:0] = Saturate((Rz[31:0], 32'h0000 0000) + Rx[31:0] X Ry[31:0] + 32'h8000 0000)[63:32]) mula.s32.rhs Rz, Rx, Ry
	The 64-bit result of multiplying Rx and Ry is accumulated with (Rz, 32'h0000 0000), and 0x8000 0000 is added to the accumulated result. The upper 32 bits of the addition result are stored in Rz. Among them, 0x8000 0000 is added to the accumulated result to implement the rounding operation. The saturation processing process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value, if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value, and in other cases, it is the addition result itself. No impact
Affected	flags:
limit	none
Control: Abnormal:	none

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 0 0 0 0 1	1 1 1 0 0	Rz	

Figure 15.92: MULA.S32.RHS

15.93 MULS.S32.RHS – 32-bit signed multiply-accumulate-subtract with rounding and saturation**High 32-bit instruction**

The unified	
	muls.s32.rhs rz, rx, ry
instruction syntax	operation Rz[31:0] = Saturate((Rz[31:0], 32'h0000 0000) - Rx[31:0] X Ry[31:0] + 32'h8000 0000)[63:32]) only has 32-bit instructions in the compilation result .
	muls.s32.rhs rz, rx, ry

illustrate	The 64-bit result of multiplying Rx and Ry is subtracted from {Rz, 32'h0000 0000}, and 0x8000 0000 is added to the result of the subtraction. The upper 32 bits of the addition result are intercepted and saturated and stored in Rz. Among them, 0x8000 0000 is added to the result of the subtraction to implement the rounding operation. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the addition result
	itself. No impact
	none
Affected flags:	none

Restrictions: **Exceptions:**

32 Bit	
	$Rz[31:0] = \text{Saturate}((Rz[31:0], 32'h0000 0000) - Rx[31:0] \times Ry[31:0] + 32'h8000 0000)[63:32]$
	muls.s32.rhs rz, rx, ry
	The 64-bit result of multiplying Rx and Ry is subtracted from {Rz, 32'h0000 0000}, and 0x8000 0000 is added to the result of the subtraction. The upper 32 bits of the instruction operation: Syntax is interpreted and saturated and stored in Rz. Among them, 0x8000 0000 is added to the result of the subtraction to implement the rounding operation. The saturation process is that if the addition result is greater than the upper saturation value 0xFFFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the addition result
Affected flags:	itself. No impact
limit	none
Control: Abnormal:	none

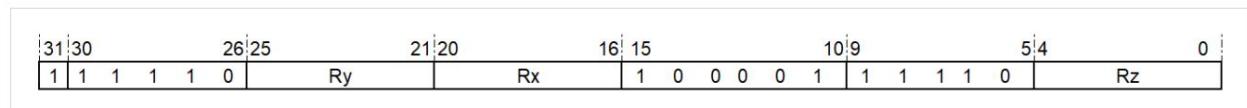
Instruction format:

Figure 15.93: MUL.S32.RHS

15.94 MULXL.S32 – 32-bit signed lower halfword unaligned multiply instruction

Unified instruction	
syntax	mulxl.s32 Rz, Rx, Ry
operation	$Rz[31:0] = (Rx[31:0] \times Ry[31:0])[47:16]$
The compiled result	only has 32-bit instructions.
	mulxl.s32 Rz, Rx, Ry

	Rx is multiplied by the lower halfword of Ry, and the upper 32 bits of the result are stored in Rz.
Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

32-bit instructions	
	Rz[31:0] = {Rx[31:0] X Ry[15:0]}[47:16]
	mulxl.s32 Rz, Rx, Ry
	Rx is multiplied by the lower halfword of Ry, and the upper 32 bits of the result are stored in Rz.
Operation: Syntax: Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

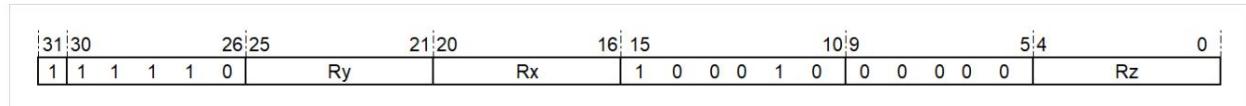
Instruction format:

Figure 15.94: MULXL.S32

15.95 MULXL.S32.R -- 32-bit signed lower halfword unaligned multiply with rounding**instruction**

Unified instructions	
Syntax	mulxl.s32.r rz, rx, ry
	Rz[31:0] = {Rx[31:0] X Ry[15:0] + 16'h8000}[47:16]
Operation Compiled results	contain only 32-bit instructions. mulxl.s32.r rz, rx, ry

illustrate:	Multiply the lower halfwords of Rx and Ry, add 0x8000, and store the upper 32 bits of the result in Rz. Used to implement rounding operations.
Impact Signs Bit:	No impact
Restrictions: None	
Abnormal: None	

32-bit pointer make	
Operation:	Rz[31:0] = {Rx[31:0] X Ry[15:0] + 16'h8000}[47:16]
Syntax:	mulxl.s32.r rz, rx, ry
Description:	Multiply the lower halfwords of Rx and Ry, add 0x8000, and store the upper 32 bits of the result in Rz. Used to implement rounding operations.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 0	1 0 0 0 0	Rz	

Figure 15.95: MULXL.S32.R

15.96 MULXH.S32——32-bit signed upper halfword unaligned multiply instruction

Unified instructions	
Syntax	mulxh.s32 rz, rx, ry
Operation	Rz[31:0] = {Rx[31:0] X Ry[31:16]}[47:16]
The compiled result	contains only 32-bit instructions. mulxh.s32 rz, rx, ry

	Rx is multiplied by the upper halfword of Ry, and the upper 32 bits of the result are stored in Rz.
Description: Impact flag:	No impact
Restrictions:	None
abnormal:	none

32-bit instructions	
	Rz[31:0] = {Rx[31:0] X Ry[31:16]}[47:16]
	mulxh.s32 rz, rx, ry
	Rx is multiplied by the upper halfword of Ry, and the upper 32 bits of the result are stored in Rz.
Operation: Syntax: Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	0 0 0 0 0 0	Rz	

Figure 15.96: MULXH.S32

15.97 MULXH.S32.R -- 32-bit signed upper halfword unaligned multiply with rounding

instruction

Unified instructions	
Syntax	mulxh.s32.r rz, rx, ry
	$Rz[31:0] = \{Rx[31:0] X Ry[31:16] + 16'h8000\}[47:16]$
Operation Compiled results	contain only 32-bit instructions. mulxh.s32.r rz, rx, ry

Illustrate:	Multiply the upper halfword of Rx and Ry, add 0x8000, and store the upper 32 bits of the result in Rz. Used to implement rounding operations.
Impact Signs Bit:	No impact
Restrictions: None	
Abnormal: None	

32-bit pointer make	
Operation:	$Rz[31:0] = \{Rx[31:0] X Ry[31:16] + 16'h8000\}[47:16]$
Syntax:	mulxh.s32.r rz, rx, ry
Description:	Multiply the upper halfword of Rx and Ry, add 0x8000, and store the upper 32 bits of the result in Rz. Used to implement rounding operations.
Impact Signs Bit:	No impact
Restrictions: None	
Abnormal: None	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	1 0 0 0 0	Rz	

Figure 15.97: MULXH.S32.R

15.98 RMULXL.S32 – 32-bit signed lower halfword unaligned fractional multiply instruction

Unified	
instruction syntax	rmulxl.s32 rz, rx, ry
operation	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[15:0]}[46:15] Only 32-bit</pre>
Compilation results	instructions exist. rmulxl.s32 rz, rx, ry

Illustrate :	The lower halfword of Rx and Ry is regarded as a signed decimal, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the 46th to 15th bits of the multiplication result of Rx and Ry are stored in Rz. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32 Bit	
	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[15:0] == 16'h8000) instruction op[46:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[15:0]}[46:15] rmulxl.s32 rz, rx, ry</pre>
	<p>The lower halfword of Rx and Ry is regarded as a signed decimal, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the 46th to 15th bits of the multiplication result of Rx and Ry are stored in</p>
	Rz. No effect
Affected flags:	
limit	none
Control: Abnormal:	none

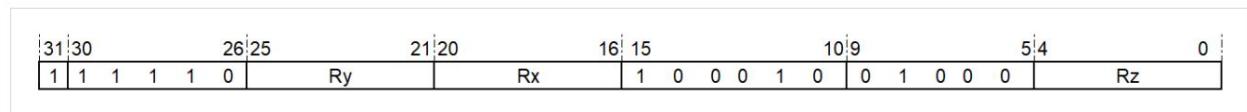
Instruction format:

Figure 15.98: RMULXL.S32

15.99 RMULXL.S32.R -- 32-bit signed lower halfword unaligned small integer with rounding.

Multiplication Instructions

Unified	
instruction syntax	rmulxl.s32.r rz, rx, ry
operation	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[15:0] + 16'h4000}[46:15] Only 32-bit instructions exist.</pre>
Compilation results	rmulxl.s32.r rz, rx, ry

illustrate	The lower halfwords of Rx and Ry are regarded as signed decimals, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfwords of Rx and Ry are multiplied, and 0x4000 is added, and the 46th to 15th bits of the result are stored in Rz. The multiplication result is added with 0x4000 to implement the decimal rounding
	operation. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32	Bit
	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[15:0] == 16'h8000) Instruction op[Ry[15:0]] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[15:0] + 16'h4000}[46:15] rmulxl.s32.r rz, rx, ry</pre>
	<p>Syntax : Description: The lower halfwords of Rx and Ry are regarded as signed decimals, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfwords of Rx and Ry are multiplied, and 0x4000 is added, and the 46th to 15th bits of the result are stored in Rz. The multiplication result is added with 0x4000 to implement the decimal rounding</p>
	operation. No effect
	none
	none
Affected	flags: Restrictions: Exceptions:

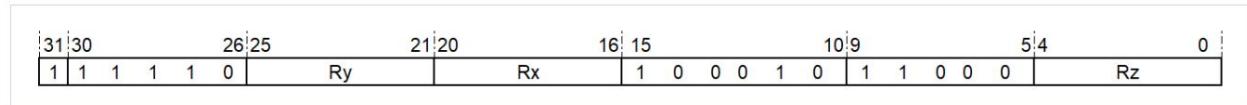
Instruction format:

Figure 15.99: RMULXL.S32.R

15.100 RMULXH.S32——32-bit signed upper halfword unaligned fractional multiply instruction

Unified	
instruction syntax	rmulxh.s32 rz, rx, ry
operation if	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[31:16] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:16]}[46:15] Only 32-bit</pre>
Compilation results	instructions exist. rmulxh.s32 rz, rx, ry

illustrate :	The lower halfword of Rx and Ry is regarded as a signed decimal, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 31st bit of Ry is the sign bit and the 30th to 16th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[31:16]=0x8000, the result is 0x7FFF FFFF; in other cases, the 46th to 15th bits of the multiplication result of Rx and Ry are stored in Rz. No effect
	none
	none
Affected flags: Restrictions: Exceptions:	

32	Bit	
		if(Rx[31:0] == 32'h8000 0000 && Ry[31:16] == 16'h8000) instruction op[41:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:16]}[46:15] rmulxh.s32 rz, rx, ry
		The lower halfword of Rx and Ry is regarded as a signed decimal, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal Syntax : Description , and the 31st bit of Ry is the sign bit and the 30th to 16th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[31:16]=0x8000, the result is 0x7FFF FFFF; in other cases, the 46th to 15th bits of the multiplication result of Rx and Ry are stored in
		Rz. No effect
	Affected flags:	
limit	none	
	none	
Control: Abnormal:		

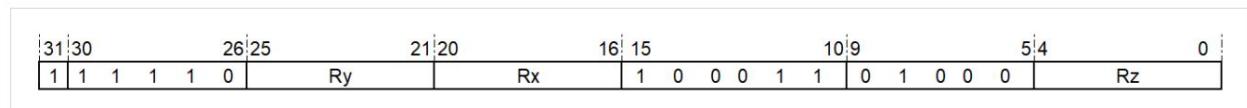
Instruction format:

Figure 15.100: RMULXH.S32

15.101 RMULXH.S32.R -- 32-bit signed upper halfword misalignment with rounding

Decimal multiplication instruction

Unified	
instruction syntax	rmulxh.s32.r rz, rx, ry
operation	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[31:16] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:16] + 16'h4000}[46:15] Only 32-bit instructions exist.</pre>
Compilation results	rmulxh.s32.r rz, rx, ry

illustrate	The lower halfwords of Rx and Ry are regarded as signed decimals, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfwords of Rx and Ry are multiplied, and 0x4000 is added, and the 46th to 15th bits of the result are stored in Rz. The multiplication result is added with 0x4000 to implement the decimal rounding
	operation. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32 Bit	
	<pre>if(Rx[31:0] == 32'h8000 0000 && Ry[31:16] == 16'h8000) instruction op Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[31:0] X Ry[31:16] + 16'h4000}[46:15] rmulxh.s32.r rz, rx, ry</pre>
Syntax : Description	The lower halfwords of Rx and Ry are regarded as signed decimals, where the 31st bit of Rx is the sign bit and the 30th to 0th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:0]=0x8000 0000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfwords of Rx and Ry are multiplied, and 0x4000 is added, and the 46th to 15th bits of the result are stored in Rz. The multiplication result is added with 0x4000 to implement the decimal rounding
Affected flags: Restrictions: Exceptions:	operation. No effect
	none
	none

Instruction format:

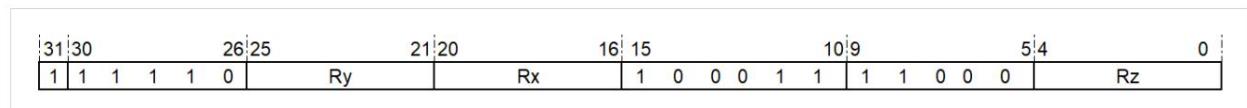


Figure 15.101: RMULXH.S32.R

15.102 MULAXL.S32.S -- 32-bit signed lower halfword misalignment with saturation

Multiply-accumulate instructions

The unified	
	mulaxl.s32.s rz, rx, ry Rz[31:0] =
	Saturate(Rz[31:0] + {Rx[31:0] X Ry[15:0]}[47:16])
instruction syntax	operation compilation result
	only has 32-bit instructions. mulaxl.s32.s rz, rx, ry

Illustrate :	Rx is multiplied by the lower halfword of Ry and the upper 32 bits of the result are added to Rz. The result of the addition is saturated and stored in Rz. The saturation process is as follows: if the addition result is greater than the upper saturation value 0xFFFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No effect
Affected flags:	
	none
Restrictions: Exceptions:	none

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + (Rx[31:0] X Ry[15:0])[47:16])
	mulaxl.s32.s rz, rx, ry
instruction operations Syntax Description	Rx is multiplied by the lower halfword of Ry and the upper 32 bits of the result are added to Rz. The result of the addition is saturated and stored in Rz. The saturation addition result is greater than the upper saturation value 0xFFFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No effect
Affected flags:	
	none
Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 0	0 1 1 0 0	Rz	

Figure 15.102: MULAXL.S32.S

15.103 MULAXL.S32.RS – 32-bit signed lower halfword unaligned multiply-accumulate instruction with rounding and saturation

The unified	
	mulaxl.s32.rs rz, rx, ry Rz[31:0] = Saturate((Rz[31:0], 16'h0000) + Rx[31:0] X Ry[15:0] + 16'h8000)[47:16])
instruction syntax operation compilation result	only has 32-bit instructions. mulaxl.s32.rs rz, rx, ry

illustrate	The lower halfwords of Rx and Ry are multiplied and added with {Rz, 16'h0000}, and 0x8000 is added to the result. The upper 32 bits of the result are taken and saturated and stored in Rz. The result of the addition is added with 0x8000 to implement the rounding operation. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value, if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value, otherwise the result is the addition result
	itself. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32 Bit	
	Rz[31:0] = Saturate((Rz[31:0], 16'h0000) + Rx[31:0] X Ry[15:0] + 16'h8000)[47:16]
	mulaxl.s32.rs rz, rx, ry
	The lower halfwords of Rx and Ry are multiplied and added with (Rz, 16'h0000), and 0x8000 is added to the result. The upper 32 bits of the result are taken and saturated instruction syntax operation compilation result of the addition is added with 0x8000 to implement the rounding operation. The saturation process is if the addition result is greater than the upper saturation value 0xFFFF FFFF, the result is the upper saturation value, if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value, otherwise the result is the addition result
Impact Mark Zhi Bit:	itself. No effect
	none
	none
Limitation: Exception:	

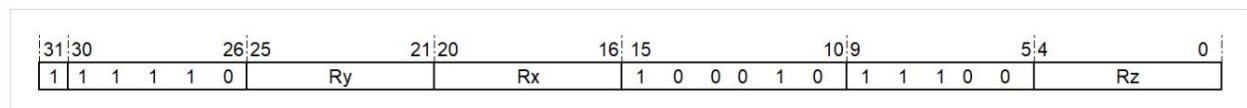
Instruction format:

Figure 15.103: MULAXL.S32.RS

15.104 MULAXH.S32.S -- 32-bit signed upper halfword misalignment with saturation

Multiply-accumulate instructions

The unified	
	mulaxh.s32.s rz, rx, ry Rz[31:0] =
	Saturate(Rz[31:0] + {Rx[31:0] X Ry[31:16]}[47:16])
instruction syntax operation compilation result	only has 32-bit instructions. mulaxh.s32.s rz, rx, ry

Illustrate :	Rx is multiplied by the upper halfword of Ry and the upper 32 bits of the result are added to Rz. The result of the addition is saturated and stored in Rz. The saturation process is as follows: if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No effect
Affected flags:	
	none
Restrictions: Exceptions:	none

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + (Rx[31:0] X Ry[31:16])[47:16])
	mulaxh.s32.s rz, rx, ry
instruction operations Syntax Description	Rx is multiplied by the upper halfword of Ry and the upper 32 bits of the result are added to Rz. The result of the addition is saturated and stored in Rz. The saturation addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No effect
Affected flags:	
	none
	none
Restrictions: Exceptions:	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	0 1 1 0 0	Rz	

Figure 15.104: MULAXH.S32.S

15.105 MULAXH.S32.RS – 32-bit signed high half with rounding and saturation

Word-unaligned multiply-accumulate instructions

The unified	
	mulaxh.s32.rs rz, rx, ry Rz[31:0] = Saturate(([Rz[31:0], 16'h0000] + Rx[31:0] X Ry[31:16] + 16'h8000)[47:16])
instruction syntax operation compilation result	only has 32-bit instructions. mulaxh.s32.rs rz, rx, ry

explain bright:	Rx is multiplied by the upper halfword of Ry, and then added to {Rz, 16'h0000}, and the result is added with 0x8000. The upper 32 bits of the result are taken and saturated and stored in Rz. The result is added with 0x8000 to implement rounding. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result itself. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32	
Bit	
	Rz[31:0] = Saturate(({Rz[31:0], 16'h0000} + Rx[31:0] X Ry[31:16] + 16'h8000)[47:16])
	mulaxh.s32.rs rz, rx, ry
	Rx is multiplied by the upper halfword of Ry, and then added to {Rz, 16'h0000}, and the result is added with 0x8000. The upper 32 bits of the result are taken and instruction operation saturates and does not in Rz . The result is added with 0x8000 to implement rounding. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the addition result
Impact Mark	itself. No effect
Zhi	
Bit:	
	none
	none
Limitation: Exception:	

Instruction format:

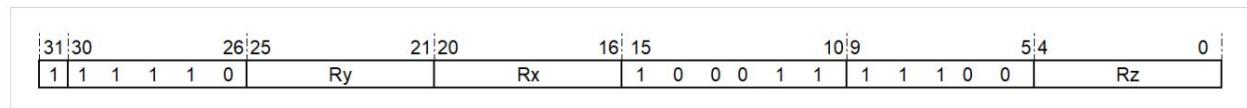


Figure 15.105: MULAXH.S32.RS

15.106 MULLL.S16 – 16-bit signed low halfword multiply instruction

The unified	
	mulll.s16 rz, rx, ry Rz[31:0] =
	Rx[15:0] X Ry[15:0]
instruction syntax operation compilation result	only has 32-bit instructions. mulll.s16 rz, rx, ry

	The lower halfword of Rx is multiplied by the lower halfword of Ry and the result is stored in Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
	Rz[31:0] = Rx[15:0] X Ry[15:0]
	mull.s16 rz, rx, ry
	The lower halfword of Rx is multiplied by the lower halfword of Ry and the result is stored in Rz.
Operation: Syntax: Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

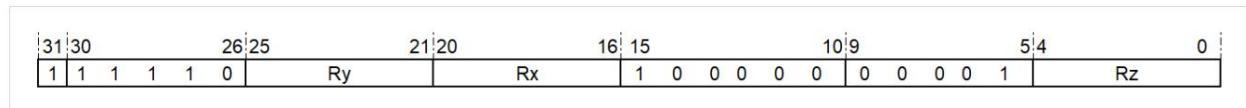
Instruction format:

Figure 15.106: MULL.S16

15.107 MULHH.S16 – 16-bit signed upper halfword multiplication instruction

Unified instructions	
Syntax	mulhh.s16 rz, rx, ry
	Rz[31:0] = Rx[31:16] X Ry[31:16]
Operation Compiled results	contain only 32-bit instructions. mulhh.s16 rz, rx, ry

	The high halfword of Rx is multiplied by the high halfword of Ry and the result is stored in Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
Operation: Rz[31:0] = Rx[31:16] X Ry[31:16]	
Syntax: mulhh.s16 rz, rx, ry	
	The high halfword of Rx is multiplied by the high halfword of Ry and the result is stored in Rz.
Description: Impact flag: No impact	
Limitations:	none
Exceptions:	none

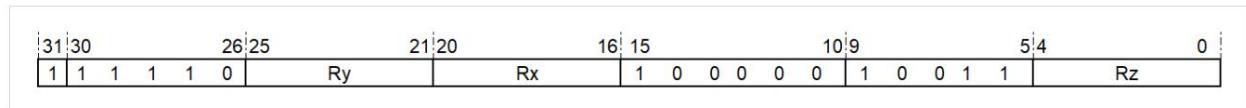
Instruction format:

Figure 15.107: MULHH.S16

15.108 MULHL.S16 – 16-bit signed high and low halfword multiplication instruction

Unified instructions	
Syntax	mulhl.s16 rz, rx, ry
	Rz[31:0] = Rx[31:16] X Ry[15:0]
Operation Compiled results	contain only 32-bit instructions. mulhl.s16 rz, rx, ry

	The upper halfword of Rx is multiplied by the lower halfword of Ry and the result is stored in Rz.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
	Rz[31:0] = Rx[31:16] X Ry[15:0]
	mulhl.s16 rz, rx, ry
	The upper halfword of Rx is multiplied by the lower halfword of Ry and the result is stored in Rz.
Operation: Syntax: Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0	1 0 0 0 1	Rz	

Figure 15.108: MULHL.S16

15.109 RMULLL.S16 – 16-bit Signed Lower Halfword Fractional Multiply Instruction

Unified	
instruction syntax	rmulll.s16 rz, rx, ry
operation	<pre>if(Rx[15:0] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[15:0] X Ry[15:0]} << 1 Only 32-bit</pre>
Compilation results	instructions exist. rmulll.s16 rz, rx, ry

Note:	The lower halfwords of Rx and Ry are regarded as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx [15:0]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FF FF; in other cases, the result of multiplying Rx and Ry is shifted left by one bit and stored in
Affected flag:	Rz. No effect
Restriction:	
None Exception:	None

32-bit	
instruction operation: if(Rx[15:0] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] ={Rx[15:0] X Ry[15:0]} << 1 Syntax: rmull.s16	
rz, rx, ry	Description: The lower halfwords of Rx and Ry are regarded as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx [15:0]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FF FF; in other cases, the result of multiplying Rx and Ry is shifted left by one bit and stored in Rz. No effect
Affected flag:	
Restriction:	
None	Exception: None

Instruction format:

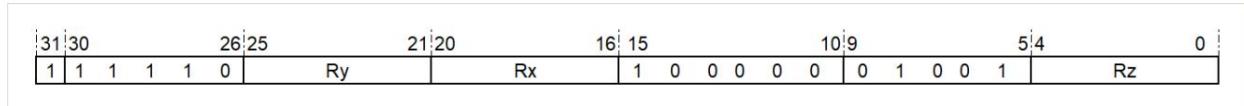


Figure 15.109: RMULLL.S16

15.110 RMULHH.S16 – 16-bit signed upper halfword fractional multiplication instruction

Unified	
instruction syntax rmulhh.s16 rz, rx, ry	
operation if(Rx[31:16] == 16'h8000 && Ry[31:16] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] ={Rx[31:16] X Ry[31:16]} << 1 Only 32-bit	
Compilation results	instructions exist. rmulhh.s16 rz, rx, ry

Note:	The upper halfword of Rx and Ry is regarded as a signed decimal, where the 31st bit is the sign bit and the 30th to 16th bits are the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF FF FF; in other cases, the result of multiplying Rx and Ry is shifted left by one bit and stored in Rz.
Affected flag:	Rz. No effect
Restriction:	
None	Exception: None

32-bit	
instruction operation: if(Rx[31:16] == 16'h8000 && Ry[31:16] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] ={Rx[31:16] X Ry[31:16]} << 1 Syntax: rmulhh.s16	
rz, rx, ry	Description: The upper halfwords
of Rx and Ry	are regarded as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF FF FF; in other cases, the result of multiplying Rx and Ry is shifted left by one bit and stored in Rz. No effect
Affected flag:	
Restriction:	
None	Exception: None

Instruction format:

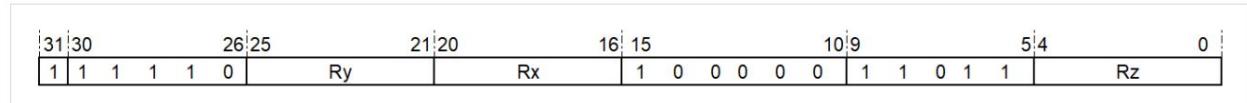


Figure 15.110: RMULHH.S16

15.111 RMULHL.S16 – 16-bit signed high and low halfword fractional multiplication instruction

Unified	
instruction syntax	rmulhl.s16 rz, rx, ry
operation	<pre>if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] ={Rx[31:16] X Ry[15:0]} << 1 Only 32-bit instructions</pre>
Compilation results	exist. rmulhl.s16 rz, rx, ry

illustrate :	The high halfword of Rx and the low halfword of Ry are regarded as signed decimals, where the 31st bit of Rx is the sign bit and the 30th to 16th bits are the decimal bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:16]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FF FF; in other cases, the result of multiplying Rx and Ry is shifted left by one bit and stored in Rz. No effect
	none
Affected flags: Restrictions: Exceptions:	none

32 Bit	
	<pre>if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) instruction op[Rz[31:0]] = 32'h7FFFF FFFF else Rz[31:0] ={Rx[31:16] X Ry[15:0]} << 1 rmulhl.s16 rz, rx, ry</pre>
	<p>Syntax : Description: The high halfword of Rx and the low halfword of Ry are regarded as signed decimals, where the 31st bit of Rx is the sign bit and the 30th to 16th bits are the additional bits, and the 15th bit of Ry is the sign bit and the 14th to 0th bits are the decimal bits. When Rx[31:16]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FF FF; in other cases, the result of multiplying Rx and Ry is shifted left by one bit and stored in Rz. No effect</p>
Affected flags:	
limit	none
Control: Abnormal:	none

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0	Ry	Rx	1 0 0 0 0 0	1 1 0 0 1	Rz	

Figure 15.111: RMULHL.S16

15.112 MULAHH.S16.S – 16-bit signed upper halfword multiply-accumulate with saturation**instruction**

The unified	
	mulahh.s16.s rz, rx, ry Rz[31:0] =
	Saturate(Rz[31:0] + Rx[31:16] X Ry[31:16])
instruction syntax operation compilation result	only has 32-bit instructions. mulahh.s16.s rz, rx, ry

Illustrate :	The high halfword of Rx is multiplied by the high halfword of Ry, and the multiplication result is added to Rz. The addition result is saturated and stored in Rz. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value. If the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value. In other cases, it is the addition result itself. No effect
Affected flags:	
	none
	none
Restrictions: Exceptions:	

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[31:16])
	mulahh.s16.s rz, rx, ry
instruction operations: <small>Source Description</small>	The high halfword of Rx is multiplied by the high halfword of Ry, and the multiplication result is added to Rz. The addition result is saturated and stored in Rz. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value. If the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value. In other cases, it is the addition result itself. No effect
Affected flags:	
	none
	none
Restrictions: Exceptions:	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0 0	1 1 1 1 1 1	Rz	

Figure 15.112: MULAHH.S16.S

15.113 MULAH.L.S16.S – 16-bit signed high and low halfword multiply-accumulate with saturation**Add instruction**

The unified	
	mulahl.s16.s rz, rx, ry Rz[31:0] =
	Saturate(Rz[31:0] + Rx[31:16] X Ry[15:0])
instruction syntax operation compilation result only has	32-bit instructions. mulahl.s16.s Rz, Rx, Ry

illustrate :	The high halfword of Rx is multiplied by the low halfword of Ry, and the multiplication result is added to Rz. The addition result is saturated and stored in Rz. The saturation process is that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value. If the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value. In other cases, it is the addition result itself. No effect
Affected flags:	
	none
	none
Restrictions: Exceptions: Instruction format :	

32	
Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[15:0])
	mulahl.s16.s Rz, Rx, Ry
instruction operation	The high halfword of Rx is multiplied by the low halfword of Ry, and the multiplication result is added to Rz. The addition result is saturated and stored in Rz. Note: saturation description that if the addition result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value. If the addition result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value. In other cases, it is the addition result itself. No effect
Affected flags:	
	none
Restrictions: Exceptions:	none

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0 Ry Rx 1 0 0 0 0 0 1 1 1 0 1 Rz						

Figure 15.113: MULAHL.S16.S

15.114 MULALL.S16.E – 16-bit signed low halfword multiply-accumulate with extend operation

instruction

Unified	
instruction syntax	mulall.s16.e rz, rx, ry operation
	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[15:0] X Ry[15:0] The compiled result only has 32-bit instructions.
	mulall.s16.e rz, rx, ry

illustrate:	The lower halfword of Rx is multiplied by the lower halfword of Ry, the multiplication result is added to {Rz+1, Rz}, and the upper 32 bits of the result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

32-bit pointer	
make	
Operation:	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[15:0] \times Ry[15:0]$
Syntax:	mulall.s16.e rz, rx, ry
Description:	The lower halfword of Rx is multiplied by the lower halfword of Ry, the multiplication result is added to {Rz+1, Rz}, and the upper 32 bits of the result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0 0	0 0 1 0 1	Rz	

Figure 15.114: MULALL.S16.E

15.115 MULAHH.S16.E – 16-bit signed upper halfword multiply-accumulate with extend operation**instruction**

Unified instructions	
Grammar	mulahh.s16.e rz, rx, ry
Operation	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[31:16] \times Ry[31:16]$
The compiled result	contains only 32-bit instructions. mulahh.s16.e rz, rx, ry

illustrate:	The high halfword of Rx is multiplied by the high halfword of Ry, the result of the multiplication is added to (Rz+1, Rz), and the high 32 bits of the result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

32-bit pointer	
make	
Operation:	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[31:16] \times Ry[31:16]$
Syntax:	mulahh.s16.e rz, rx, ry
Description:	The high halfword of Rx is multiplied by the high halfword of Ry, the result of the multiplication is added to (Rz+1, Rz), and the high 32 bits of the result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0 0	1 0 1 1 1	Rz	

Figure 15.115: MULAHH.S16.E

15.116 MULAHL.S16.E – 16-bit signed high and low halfword multiply-accumulate with extend operation**Add instruction**

Unified instructions	
Grammar	mulahl.s16.e rz, rx, ry
Operation	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[31:16] \times Ry[15:0]$
The compiled result	contains only 32-bit instructions. mulahl.s16.e rz, rx, ry

illustrate:	The high halfword of Rx is multiplied by the low halfword of Ry, the result of the multiplication is added to {Rz+1, Rz}, and the high 32 bits of the result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

32-bit pointer	
make	
Operation:	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[31:16] \times Ry[15:0]$
Syntax:	mulahl.s16.e rz, rx, ry
Description:	The high halfword of Rx is multiplied by the low halfword of Ry, the result of the multiplication is added to {Rz+1, Rz}, and the high 32 bits of the result are stored in Rz+1. The lower 32 bits are stored in Rz.
Impact Signs	No impact
Bit:	
Restrictions:	None
Abnormal:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0 0	0 0 1 1 1	Rz	

Figure 15.116: MULAHL.S16.E

15.117 PMUL.(U/S)16 -- 16-bit parallel (unsigned/signed) multiplication instruction

Unified instructions	
grammar	pmul.u16 rz, rx, ry pmul.s16 rz, rx, ry
operate	$Rz+1[31:0] = Rx[31:16] \times Ry[31:16]$ $Rz[31:0] = Rx[15:0] \times Ry[15:0]$
The compiled result	contains only 32-bit instructions. pmul.u16 rz, rx, ry pmul.s16 rz, rx, ry

illustrate:	The high halfword of Rx is multiplied by the high halfword of Ry, and the result is stored in Rz+1; The lower halfword of Rx is multiplied by the lower halfword of Ry and the result is stored in Rz.
Impact Signs	No impact
Bit:	
Limitations:	none
Exceptions:	none

32-bit instructions	
operate:	$Rz+1[31:0] = Rx[31:16] \times Ry[31:16]$ $Rz[31:0] = Rx[15:0] \times Ry[15:0]$
grammar:	pmul.u16 rz, rx, ry pmul.s16 rz, rx, ry
illustrate:	The high halfword of Rx is multiplied by the high halfword of Ry, and the result is stored in Rz+1; The lower halfword of Rx is multiplied by the lower halfword of Ry and the result is stored in Rz.
Impact Signs	No impact
Bit:	
Limitations:	none
Exceptions:	none

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	1	1	1	1	0	Ry	Rx	1	0	0	0	1

Figure 15.117: PMUL_(US)16

Note: When S is 1, it is unsigned, and when S is 0, it is signed.

15.118 PMULX.(U/S)16 -- 16-bit parallel (unsigned/signed) crossover multiplication instruction

Unified instructions	
grammar	pmulx.u16 rz, rx, ry pmulx.s16 rz, rx, ry
operate	Rz+1[31:0] = Rx[31:16] X Ry[15:0] Rz[31:0] = Rx[15:0] X Ry[31:16]
The compiled result	contains only 32-bit instructions. pmulx.u16 rz, rx, ry pmulx.s16 rz, rx, ry

illustrate:	Multiply the high halfword of Rx by the low halfword of Ry and store the result in Rz+1; The lower halfword of Rx is multiplied by the upper halfword of Ry and the result is stored in Rz;
Impact Signs	No impact
Bit:	
Limitations:	none
Exceptions:	none

32-bit instructions	
operate:	Rz+1[31:0] = Rx[31:16] X Ry[15:0] Rz[31:0] = Rx[15:0] X Ry[31:16]
grammar:	pmulx.u16 rz, rx, ry pmulx.s16 rz, rx, ry
illustrate:	Multiply the high halfword of Rx by the low halfword of Ry and store the result in Rz+1; The lower halfword of Rx is multiplied by the upper halfword of Ry and the result is stored in Rz;
Impact Signs	No impact
Bit:	
Limitations:	none
Exceptions:	none

Instruction format:

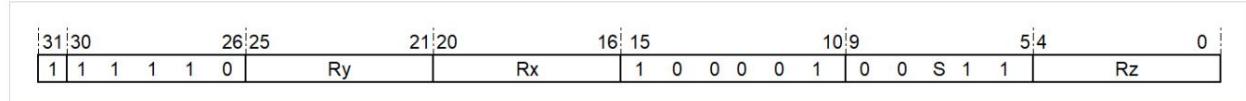


Figure 15.118: PMULX.(US)16

Note: When S is 1, it is unsigned, and when S is 0, it is signed.

15.119 PRMUL.S16 – 16-bit Parallel Signed Fractional Multiply Instruction

System one	
Instructions prmul.s16 rz, rx, ry	
Law	
operate	<pre> if(Rx[31:16] == 16'h8000 && Ry[31:16] == 16'h8000) Rz+1[31:0] = 32'h7FFF FFFF else Rz+1[31:0] = {Rx[31:16] X Ry[31:16]} << 1 if(Rx[15:0] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[15:0] X Ry[15:0]}<< 1 Only 32-bit </pre>
Compilation results	instructions exist. prmul.s16 rz, rx, ry

Illustrate: Ry are considered signed decimals, with the 15th bit being the sign bit and the 14th to 0th bits being the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x8000, the result is 0xFFFF FFFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz+1. When Rx[15:0]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz. No effect	
Affected flags:	
limit none	
Control: Abnormal: none	

32	Bit	
		<pre> if(Rx[31:16] == 16'h8000 && Ry[31:16] == 16'h8000) Instruction op Rz+1[31:0] = 32'h7FFF FFFF else Rz+1[31:0] = {Rx[31:16] X Ry[31:16]} << 1 if(Rx[15:0] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[15:0] X Ry[15:0]}<< 1 prmul.s16 rz, rx, </pre>
	ry	
		<p>The upper halfwords of Rx and Ry are both considered signed decimals, with the 31st bit being the sign bit and the 30th to 16th bits being the decimal bits; the lower halfwords of Rx and Ry are considered signed decimals, with the 15th bit being the sign bit and the 14th to 0th bits being the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF FFFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz+1. When Rx[15:0]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the result of the multiplication is shifted left by one position</p>
		and stored in Rz. No effect
		none
		none
Affected	flags: Restrictions: Exceptions:	

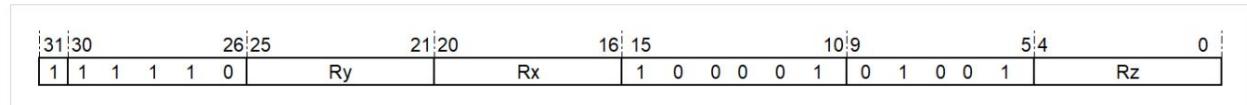
Instruction format:

Figure 15.119: PRMUL.S16

15.120 PRMULX.S16 – 16-bit Parallel Signed Interleaved Fractional Multiply Instruction

System one	
	prmulx.s16 rz, rx, ry
Optimized instructions	<pre> if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz+1[31:0] = 32'h7FFF FFFF else Rz+1[31:0] = {Rx[31:16] X Ry[15:0]} << 1 if(Rx[15:0] == 16'h8000 && Ry[31:16] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[15:0] X Ry[31:16]} << 1 Only 32-bit </pre>
Compilation results	<p>instructions exist.</p> <p>prmulx.s16 rz, rx, ry</p>

Illustrate:	The upper halfwords of Rx and Ry are both considered signed decimals, with the 31st bit being the sign bit and the 30th to 16th bits being the decimal bits; the lower halfwords of Rx and Ry are considered signed decimals, with the 15th bit being the sign bit and the 14th to 0th bits being the decimal bits. When Rx[31:16]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the upper halfword of Rx is multiplied by the lower halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz+1. When Rx[15:0]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfword of Rx is multiplied by the upper halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz. No effect
Affected flags:	none
Restrictions:	none
Exceptions:	

32	Bit
	<pre> if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Instruction op Rz+1[31:0] = 32'h7FFF FFFF else Rz+1[31:0] = {Rx[31:16] X Ry[15:0]} << 1 if(Rx[15:0] == 16'h8000 && Ry[31:16] == 16'h8000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = {Rx[15:0] X Ry[31:16]} << 1 prmulx.s16 rz, rx, </pre>
	ry
	<p>Syntax : Description:</p> <p>The upper halfwords of Rx and Ry are both considered signed decimals, with the 31st bit being the sign bit and the 30th to 16th bits being the decimal bits; the lower halfwords of Rx and Ry are considered signed decimals, with the 15th bit being the sign bit and the 14th to 0th bits being the decimal bits. When Rx[31:16]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF FFFF; in other cases, the upper halfword of Rx is multiplied by the lower halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz+1. When Rx[15:0]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF FFFF; in other cases, the lower halfword of Rx is multiplied by the upper halfword of Ry, and the result of the multiplication is shifted left by one position and stored in Rz.</p>
	stored in Rz. No effect
	none
	none
Affected	flags: Restrictions: Exceptions:

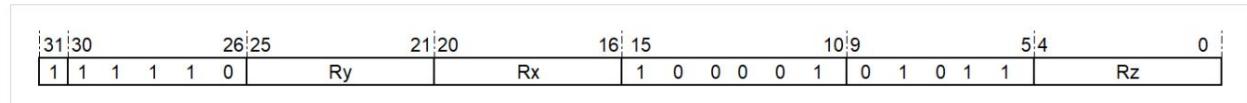
Instruction format:

Figure 15.120: PRMULX.S16

15.121 PRMUL.S16.H – 16-bit Parallel Signed Fractional Multiply Upper 16-bit Instruction

System one	
	prmul.s16.h rz, rx, ry
Optimize instruction Rz[31:16] = 16'h8000 && Ry[31:16] == 16'h8000 Rz[15:0] = 16'hFFFF else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[15:0])[30:15] Only	
Compilation results	exists for 32-bit instructions. prmul.s16.h rz, rx, ry

Illustrate	The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x80 00, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the upper halfword of Rz. When Rx[15:0]=0x8000 and Ry[15:0]=0x80 00, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the lower halfword of Rz. No impact
	none
Affected flags: Restrictions: Exceptions:	none

32	Bit
	<pre> if(Rx[31:16] == 16'h8000 && Ry[31:16] == 16'h8000) Rz[31:16] = 16'h7FFF else Rz[31:16] = {Rx[31:16]} Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[15:0])[30:15] </pre>
	prmul.s16.h rz, rx, ry
	<p>The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x80 00, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the upper halfword of Rz. When Rx[15:0]=0x8000 and Ry[15:0]=0x80 00, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the lower halfword of Rz.</p>
	halfword of Rz. No impact
	none
	none
Affected flags: Restrictions: Exceptions:	

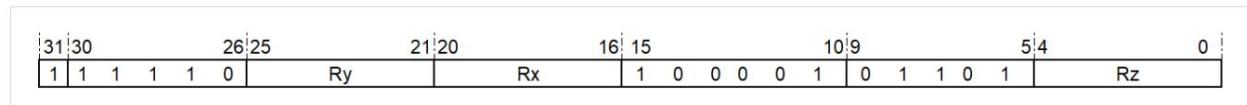
Instruction format:

Figure 15.121: PRMUL.S16.H

15.122 PRMUL.S16.RH – 16-bit Parallel Signed Fractional Multiplication with Rounding

Fetch high 16 bits of instruction

System one	
	prmul.s16.rh rz, rx, ry
Optimize instruction Rz[31:16] = 16'h8000 Rz[15:0] = 16'h7FFF else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[15:0] + 16'h4000}[30:15] Only 32-	
Compilation results	bit instructions exist. prmul.s16.rh rz, rx, ry

	<p>The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the upper halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. When Rx[15:0]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the lower halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. No effect</p>
	none
Affected flags:	none

32	Bit
	<pre> if(Rx[31:16] == 16'h8000 && Ry[31:16] == 16'h8000) Rz[31:16] = 16'h7FFF else Rz[31:16] = {Rx[31:16]} Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[15:0] + 16'h4000}[30:15] </pre>
	prmul.s16.rh rz, rx, ry
	<p>The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the upper halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. When Rx[15:0]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied with the lower halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the lower halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. No effect</p>
	none
	none
Affected flags: Restrictions: Exceptions:	

Instruction format:

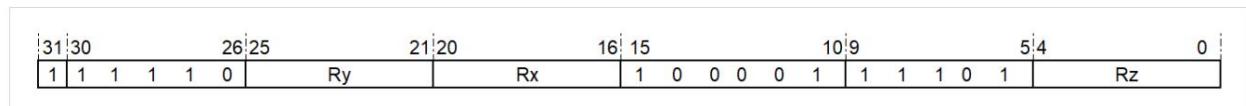


Figure 15.122: PRMUL.S16.RH

15.123 PRMULX.S16.H -- 16-bit parallel signed interleaved fractional multiplication high 16**Bit Instructions**

System one	
	prmulx.s16.h rz, rx, ry
Optimize instruction if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:16] = 16'h7FFF else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[31:16])[30:15] Only	<pre>if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:16] = 16'h7FFF else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[31:16])[30:15] Only</pre>
Compilation results	32-bit instructions exist. prmulx.s16.h rz, rx, ry

explain bright:	The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x80 00, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the upper halfword of Rz. When Rx[15:0]=0x8000 and Ry[15:0]=0x80 00, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the lower halfword of Rz. No impact
	none
Affected flags: Restrictions: Exceptions:	none

32	Bit
	<pre> if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:16] = 16'h7FFF else Rz[31:16] = {Rx[31:16]} Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[31:16])[30:15]} </pre>
	prmulx.s16.h rz, rx, ry
	<p>The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[31:16]=0x80 00, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the upper halfword of Rz. When Rx[15:0]=0x8000 and Ry[15:0]=0x80 00, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the 30th to 15th bits of the multiplication result are stored in the lower halfword of Rz.</p>
	halfword of Rz. No impact
	none
	none
Affected flags: Restrictions: Exceptions:	

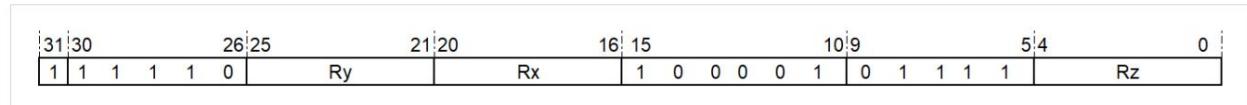
Instruction format:

Figure 15.123: PRMULX.S16.H

15.124 PRMULX.S16.RH – 16-bit parallel signed interleaved small integer with rounding**Multiplication of numbers with high 16 bits**

System one	
	prmulx.s16.rh rz, rx, ry
Optimize instruction Rz[31:16] = 16'h8000 else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[31:16] + 16'h4000}[30:15] Only 32-	<pre> if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:16] = 16'h8000 else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[31:16] + 16'h4000}[30:15] Only 32- </pre>
Compilation results	<p>bit instructions exist.</p> <p>prmulx.s16.rh rz, rx, ry</p>

	<p>The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the lower halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the upper halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. When Rx[15:0]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the upper halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the lower halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. No effect</p>
	none
Affected flags:	none

32 Bit	
	<pre> if(Rx[31:16] == 16'h8000 && Ry[15:0] == 16'h8000) Rz[31:16] = 16'h7FFF else Rz[31:16] = {Rx[31:16] Rz[15:0] = 16'h7FFF else Rz[15:0] = {Rx[15:0] X Ry[31:16] + 16'h4000}[30:15] </pre>
	prmulx.s16.rh rz, rx, ry
	<p>The upper halfwords of Rx and Ry are both considered as signed decimals, with the 31st bit as the sign bit and the 30th to 16th bits as the decimal bits; the lower halfwords of Rx and Ry are considered as signed decimals, with the 15th bit as the sign bit and the 14th to 0th bits as the decimal bits. When Rx[31:16]=0x8000 and Ry[15:0]=0x8000, the result is 0x7FFF; in other cases, the upper halfword of Rx is multiplied by the lower halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the upper halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. When Rx[15:0]=0x8000 and Ry[31:16]=0x8000, the result is 0x7FFF; in other cases, the lower halfword of Rx is multiplied by the upper halfword of Ry, 0x4000 is added to the multiplication result, and the 30th to 15th bits of the addition result are stored in the lower halfword of Rz. The multiplication result is added with 0x4000 to implement the rounding operation. No effect</p>
	none
	none
Affected flags: Restrictions: Exceptions:	

Instruction format:

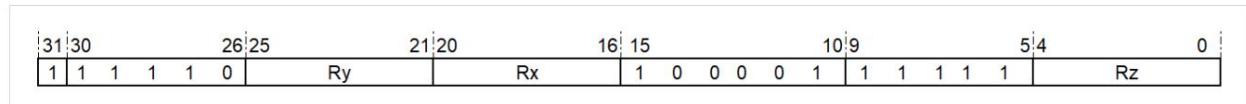


Figure 15.124: PRMULX.S16.RH

15.125 MULCA.S16.S – 16-bit signed multiply-chain-add instruction with saturation

Unified	
instruction syntax	mulca.s16.s rz, rx, ry
operation if	(Rx[31:0] = 32'h80008000 && Ry[31:0] = 32'h80008000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = Rx[31:16] X Ry[31:16] + Rx[15:0] X Ry[15:0] Only 32-bit instructions
Compilation results	exist. mulca.s16.s rz, rx, ry

Explanation:	The lower halfword of Rx is multiplied by the lower halfword of Ry, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the two multiplication results are added. When Rx=0x80008000 and Ry=0x80008000, the multiplication chain addition result overflows and is saturated to 0x7FFF
Affected flag:	FFFF. No effect
Restriction:	
None Exception:	None

32-bit	
instruction operation:	if(Rx[31:0] = 32'h80008000 && Ry[31:0] = 32'h80008000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = Rx[31:16] X Ry[31:16] + Rx[15:0] X Ry[15:0] Syntax: mulca.s16.s
rz, rx, ry Description:	Multiply the lower halfword of Rx by the lower halfword of Ry, multiply the upper halfword of Rx by the upper halfword of Ry, and add the two multiplication results.
	When Rx=0x80008000 and Ry= 0x80008000, the multiplication chain addition result overflows and is saturated to 0x7FFF
Affected flag:	FFFF. No effect
Restriction:	
None Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 0	1 1 1 1 1	Rz	

Figure 15.125: MULCA.S16.S

15.126 MULCAX.S16.S – 16-bit signed cross-multiply chain add with saturation

make

Unified	
instruction syntax	mulcax.s16.s rz, rx, ry
operation	if(Rx[31:0] = 32'h80008000 && Ry[31:0] = 32'h80008000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = Rx[31:16] X Ry[15:0] + Rx[15:0] X Ry[31:16] Only 32-bit instructions
Compilation results	exist. mulcax.s16.s rz, rx, ry

Explanation:	The lower halfword of Rx is multiplied by the upper halfword of Ry, and the upper halfword of Rx is multiplied by the lower halfword of Ry, and the two multiplication results are added. When Rx=0x80008000 and Ry=0x80008000, the multiplication chain addition result overflows and is saturated to 0x7FFF FFFF. No effect
Affected flags:	
Restrictions:	
No exceptions:	
No	
Instruction format:	

32-bit	
instruction operation:	if(Rx[31:0] = 32'h80008000 && Ry[31:0] = 32'h80008000) Rz[31:0] = 32'h7FFF FFFF else Rz[31:0] = Rx[31:16] X Ry[15:0] + Rx[15:0] X Ry[31:16] Syntax: mulcax.s16.s rz, rx, ry
Description:	Multiply the lower halfword of Rx by the upper halfword of Ry, and the upper halfword of Rx by the lower halfword of Ry, and add the two multiplication results. When Rx=0x80008000 and Ry= 0x80008000, the multiplication chain addition result overflows and is saturated to 0x7FFF FFFF. No effect
Affected flag:	
Restriction:	None
Exception:	None

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0 Ry Rx 1 0 0 0 1 0 0 1 0 1 1 Rz						

Figure 15.126: MULCAX.S16.S

15.127 MULCS.S16 – 16-bit signed multiply, chain and subtract instruction

The unified	
	mulcs.s16 rz, rx, ry Rz[31:0] =
	Rx[15:0] X Ry[15:0] - Rx[31:16] X Ry[31:16]
instruction syntax operation compilation result	only has 32-bit instructions. mulcs.s16 rz, rx, ry

subtracted.	The lower halfword of Rx is multiplied by the lower halfword of Ry, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the two multiplication results are
Description: Affected flags:	None
Affected	none
restrictions: Exceptions:	none

32-bit instructions	
Operation:	$Rz[31:0] = Rx[15:0] \times Ry[15:0] - Rx[31:16] \times Ry[31:16]$
Syntax:	mulcs.s16 rz, rx, ry
	The lower halfword of Rx is multiplied by the lower halfword of Ry, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the two multiplication results are subtracted.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	none

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 0	1 0 0 0 1	Rz	

Figure 15.127: MULCS.S16

15.128 MULCSR.S16 – 16-bit Signed Reverse Multiply Chain Subtract Instruction

Unified instructions	
Syntax	mulcsr.s16 rz, rx, ry
	$Rz[31:0] = Rx[31:16] \times Ry[31:16] - Rx[15:0] \times Ry[15:0]$
Operation Compiled results	contain only 32-bit instructions. mulcsr.s16 rz, rx, ry

	The upper halfword of Rx is multiplied by the upper halfword of Ry, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the two multiplication results are subtracted.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
	$Rz[31:0] = Rx[31:16] \times Ry[31:16] - Rx[15:0] \times Ry[15:0]$
	mulcsr.s16 rz, rx, ry
	The upper halfword of Rx is multiplied by the upper halfword of Ry, the lower halfword of Rx is multiplied by the lower halfword of Ry, and the two multiplication results are subtracted.
Operation: Syntax: Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0:
1 1 1 1 0	Ry	Rx	1 0 0 0 1	1 0 0 1 1	Rz	0

Figure 15.128: MULCSR.S16

15.129 MULCSX.S16 – 16-bit signed cross-multiply chain subtract instruction

Unified instructions	
Syntax	mulcsx.s16 rz, rx, ry
	$Rz[31:0] = Rx[15:0] \times Ry[31:16] - Rx[31:16] \times Ry[15:0]$
Operation Compiled	results contain only 32-bit instructions. mulcsx.s16 rz, rx, ry

	The low word of Rx is multiplied by the high half word of Ry, the high half word of Rx is multiplied by the low half word of Ry, and the two multiplication results are subtracted.
Description: Impact flag:	No impact
Restrictions:	None
Abnormal:	None

32-bit instructions	
Operation:	$Rz[31:0] = Rx[15:0] \times Ry[31:16] - Rx[31:16] \times Ry[15:0]$
Syntax:	mulcsx.s16 rz, rx, ry
	The low word of Rx is multiplied by the high half word of Ry, the high half word of Rx is multiplied by the low half word of Ry, and the two multiplication results are subtracted.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0:
1 1 1 1 0	Ry	Rx	1 0 0 0 1	1 0 0 0 1	Rz	0

Figure 15.129: MULCSX.S16

15.130 MULACA.S16.S – 16-bit signed multiply-chain-add-accumulate instruction with saturation

make

Unified instruction	
syntax	mulaca.s16.s rz, rx, ry operation Rz[31:0] = Saturate(Rz[31:0]
+ Rx[31:16] X Ry[31:16] + Rx[15:0] X Ry[15:0]) The compiled result only has 32-bit instructions.	
	mulaca.s16.s rz, rx, ry

illustrate:	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and accumulated with Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
	none
	none
Affected	Flags: Restrictions: Exceptions:

32	
Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[31:16] + Rx[15:0] X Ry[15:0])
	mulaca.s16.s rz, rx, ry
instruction operation	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
Impact Mark	
Zhi	
Bit:	
	none
	none
Limitation: Exception:	

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 1 0	Ry	Rx	1 0 0 0 1 0	0 1 1 0 1	Rz	

Figure 15.130: MULACA.S16.S

15.131 MULACAX.S16.S – 16-bit signed cross-multiply chain add-accumulate instruction with saturation

Unified	
instruction syntax	mulacax.s16.s rz, rx, ry operation
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[31:16] + Rx[15:0] X Ry[15:0]) The compilation result only has 32-bit instructions.
	mulacax.s16.s rz, rx, ry

illustrate :	The lower halfword of Rx is multiplied by the upper halfword of Ry, and the upper halfword of Rx is multiplied by the lower halfword of Ry. The two multiplication results are added and accumulated with Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
Affected flags:	
limit	none
Control: Abnormal:	none

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[15:0] + Rx[15:0] X Ry[31:16])
instruction operation : Syntax :	mulacax.s16.s rz, rx, ry
explain bright:	The lower halfword of Rx is multiplied by the upper halfword of Ry, and the upper halfword of Rx is multiplied by the lower halfword of Ry. The two multiplication results are added and accumulated with Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
Affected flags: Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0 Ry		Rx	1 0 0 0 1 0	0 1 1 1 1 Rz		

Figure 15.131: MULACAX.S16.S

15.132 MULACS.S16.S – 16-bit signed multiply-chain-subtract-accumulate instruction with saturation

make

The unified	
instruction syntax	mulacs.s16.s rz, rx, ry operation Rz[31:0] =
Saturate(Rz[31:0] + Rx[15:0] X Ry[15:0] - Rx[31:16] X Ry[31:16]) only has 32-bit instructions in the compiled result .	
	mulacs.s16.s rz, rx, ry

illustrate:	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are subtracted and accumulated with Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
Affected flags:	
limit	none
Control: Abnormal:	none

32	
Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[15:0] X Ry[15:0] - Rx[31:16] X Ry[31:16])
	mulacs.s16.s rz, rx, ry
instruction operation	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are subtracted and accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result
Impact Mark Zhi	itself. No effect
Bit:	
	none
	none
Limitation: Exception:	

Instruction format:

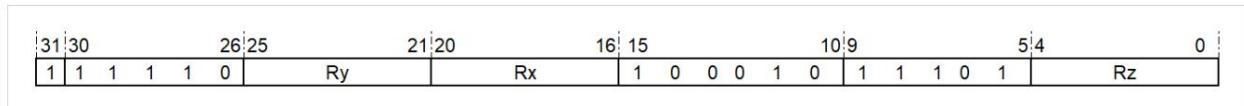


Figure 15.132: MULACS.S16.S

15.133 MULACSR.S16.S – 16-bit Signed Reverse Multiply Chain Subtract with Saturation

Accumulate instruction

The unified	
instruction syntax	mulacsr.s16.s rz, rx, ry operation
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0]) only has 32-bit instructions as a result of compilation.
	mulacsr.s16.s rz, rx, ry

illustrate :	The high halfword of Rx is multiplied by the high halfword of Ry, and the low halfword of Rx is multiplied by the low halfword of Ry. The two multiplication results are subtracted and added to Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
Affected flags:	
limit	none
Control: Abnormal:	none

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0])
instruction operation : Syntax :	mulcsr.s16.s rz, rx, ry
explain bright:	The high halfword of Rx is multiplied by the high halfword of Ry, and the low halfword of Rx is multiplied by the low halfword of Ry. The two multiplication results are subtracted and added to Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result itself. No effect
Affected flags: Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 0	1 1 1 1 1	Rz	

Figure 15.133: MULACSR.S16.S

15.134 MULACSX.S16.S -- 16-bit signed cross-multiply chain subtract with saturation

Accumulate instruction

Unified instructions	
Syntax	mulacsx.s16.s rz, rx, ry
Operation	$Rz[31:0] = \text{Saturate}(Rz[31:0] + Rx[15:0] \times Ry[31:16] - Rx[31:16] \times Ry[15:0])$
The compiled result	contains only 32-bit instructions. mulacsx.s16.s rz, rx, ry

illustrate:	The lower halfword of Rx is multiplied by the upper halfword of Ry, and the upper halfword of Rx is multiplied by the lower halfword of Ry. The two multiplication results are subtracted and added to Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result
Affected flags:	itself. No effect
limit	none
Control: Abnormal:	none
32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] + Rx[15:0] X Ry[31:16] - Rx[31:16] X Ry[15:0])
	mulacsx.s16.s rz, rx, ry
instruction operation Syntax:	Rx is multiplied by the upper halfword of Ry, and the upper halfword of Rx is multiplied by the lower halfword of Ry. The two multiplication results are subtracted and added to Rz. The accumulated result is saturated and stored in Rz. The saturation process is as follows: if the accumulated result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the accumulated result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the accumulated result
bright:	itself. No effect
	none
	none
Affected flags: Restrictions: Exceptions:	

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	1	1	1	1	0	Ry	Rx	1	0	0	1	1

Figure 15.134: MULACSX.S16.S

15.135 MULSCA.S16.S – 16-bit signed multiply-chain add-accumulate-subtract with saturation

make

Unified	
instruction syntax	<code>mulsca.s16.s rz, rx, ry</code>
operation	$Rz[31:0] = \text{Saturate}(Rz[31:0] - Rx[31:16] \times Ry[31:16] - Rx[15:0] \times Ry[15:0])$ The compiled
result only has	32-bit instructions. <code>mulsca.s16.s rz, rx, ry</code>

illustrate :	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and subtracted from Rz. The subtraction result is saturated and stored in Rz. The saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the subtraction
	result itself. No effect
	none
Affected	Flags: Restrictions: Exceptions:

32	
Bit	
	Rz[31:0] = Saturate(Rz[31:0] - Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0])
	mulscas16.s rz, rx, ry
instruction syntax:	The lower halfword of Rx is multiplied by the lower halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and subtraction result is saturated and stored in Rz.
operation:	The saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, the result is the subtraction
Impact Mark	result itself. No effect
Zhi	
Bit:	
	none
	none
Limitation:	Exception:

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	0 1 1 1 1	Rz	

Figure 15.135: MULSCA.S16.S

15.136 MULSCAX.S16.S -- 16-bit signed cross-multiply chained add-accumulate with saturation

Unified	
instruction syntax	mulscax.s16.s rz, rx, ry operation
	Rz[31:0] = Saturate(Rz[31:0] - Rx[31:16] X Ry[15:0] - Rx[15:0] X Ry[31:16]) The compiled result only has 32-bit instructions.
	mulscax.s16.s rz, rx, ry

illustrate :	The high halfword of Rx is multiplied by the low halfword of Ry, and the low halfword of Rx is multiplied by the high halfword of Ry. The two multiplication results are added and subtracted from Rz. The subtraction result is saturated and stored in Rz. The saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the subtraction result itself. No effect
Affected flags:	
limit	none
Control: Abnormal:	none

32 Bit	
	Rz[31:0] = Saturate(Rz[31:0] - Rx[31:16] X Ry[15:0] - Rx[15:0] X Ry[31:16])
instruction operation : Syntax :	mulscax.s16.s rz, rx, ry
explain bright:	The high halfword of Rx is multiplied by the low halfword of Ry, and the low halfword of Rx is multiplied by the high halfword of Ry. The two multiplication results are added and subtracted from Rz. The subtraction result is saturated and stored in Rz. The saturation process is as follows: if the subtraction result is greater than the upper saturation value 0x7FFF FFFF, the result is the upper saturation value; if the subtraction result is less than the lower saturation value 0x8000 0000, the result is the lower saturation value; otherwise, it is the subtraction result itself. No effect
Affected flags: Restrictions: Exceptions:	none

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	1 1 1 0 1	Rz	

Figure 15.136: MULSCAX.S16.S

15.137 MULACA.S16.E – 16-bit signed multiply-chain-add-accumulate instruction with extension operation

make

Unified	
	mulaca.s16.e rz, rx, ry
command syntax operation:	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[31:16] \times Ry[31:16] + Rx[15:0] \times Ry[15:0]$ The compiled result contains only 32-bit instructions.
	mulaca.s16.e rz, rx, ry

Description:	The lower halfword of Rx is multiplied by the lower halfword of Ry, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the two multiplication results are added and combined. $(Rz+1, Rz)$ Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	$\{Rz+1[31:0], Rz[31:0]\} = \{Rz+1[31:0], Rz[31:0]\} + Rx[31:16] \times Ry[31:16] + Rx[15:0] \times Ry[15:0]$ Syntax: mulaca.s16.e rz, rx, ry Description:
	The lower halfword of Rx is multiplied by the lower
	halfword of Ry, and the upper halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and merged. $(Rz+1, Rz)$ Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1	0 0 1 0 1	Rz	

Figure 15.137: MULACA.S16.E

15.138 MULACAX.S16.E -- 16-bit signed cross-multiply chain-add-accumulate instruction

with extend operation

Unified	
	mulacax.s16.e rz, rx, ry
command syntax operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[31:16] X Ry[15:0] + Rx[15:0] X Ry[31:16] The compiled result contains only 32-bit instructions.
	mulacax.s16.e rz, rx, ry

Description:	The upper halfword of Rx is multiplied by the lower halfword of Ry, and the lower halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and combined. (Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[31:16] X Ry[15:0] + Rx[15:0] X Ry[31:16] Syntax: mulacax.s16.e rz, rx, ry Description:
	The upper halfword of Rx is multiplied by the lower
	halfword of Ry, and the lower halfword of Rx is multiplied by the upper halfword of Ry. The two multiplication results are added and summed.
	(Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1	0 0 1 1 1	Rz	

Figure 15.138: MULACAX.S16.E

15.139 MULACS.S16.E – 16-bit signed multiply-chain-subtract-accumulate instruction with extended operation

make

Unified	
	mulacs.s16.e rz, rx, ry
command syntax operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[15:0] X Ry[15:0] - Rx[31:16] X Ry[31:16] The compiled result contains only 32-bit instructions.
	mulacs.s16.e rz, rx, ry

Description:	The lower halfword of Rx is multiplied by the lower halfword of Ry, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the two multiplication results are subtracted and combined. (Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[15:0] X Ry[15:0] - Rx[31:16] X Ry[31:16] Syntax: mulacs.s16.e rz, rx, ry Description:
Multiply the lower halfword of Rx by the lower	
halfword of Ry, multiply the upper halfword of Rx by the upper halfword of Ry, subtract the two multiplication results and sum them.	
	(Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1	0 0 1 1 1	Rz	

Figure 15.139: MULACAX.S16.E

15.140 MULACSR.S16.E – 16-bit signed reverse multiply-chain-subtract with extend operation

Accumulate instruction

Unified	
	mulacsr.s16.e rz, rx, ry
command syntax operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0] The compiled result only has 32-bit instructions.
	mulacsr.s16.e rz, rx, ry

Description:	The upper halfword of Rx is multiplied by the upper halfword of Ry, and the lower halfword of Rx is multiplied by the lower halfword of Ry. The two multiplication results are subtracted and accumulated with (Rz+1, Rz). The upper 32 bits of the accumulated result are stored in Rz+1, and the lower 32 bits are stored
Affected flag:	in Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0] Syntax: mulacsr.s16.e rz, rx, ry Description:
Multiply the upper halfword of Rx by the upper	
halfword of Ry, multiply the lower halfword of Rx by the lower halfword of Ry, subtract the two multiplication results and combine them	
(Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in	
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1	0 1 0 1 1 1	Rz	

Figure 15.140: MULACSR.S16.E

15.141 MULACSRX.S16.E -- 16-bit signed cross-product chain with extend operation

Subtract and accumulate instructions

Unified	
	mulacsx.s16.e rz, rx, ry
command syntax operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[15:0] X Ry[31:16] - Rx[31:16] X Ry[15:0] The compiled result only has 32-bit instructions.
	mulacsx.s16.e rz, rx, ry

Description:	The lower halfword of Rx is multiplied by the upper halfword of Ry, the upper halfword of Rx is multiplied by the lower halfword of Ry, and the two multiplication results are subtracted and combined. (Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} + Rx[15:0] X Ry[31:16] - Rx[31:16] X Ry[15:0] Syntax: mulacsx.s16.e rz, rx, ry Description:
Multiply the lower halfword of Rx by the upper	
halfword of Ry, multiply the upper halfword of Rx by the lower halfword of Ry, subtract the two multiplication results and sum them.	
(Rz+1, Rz) Accumulate. The high 32 bits of the accumulation result are stored in Rz+1, and the low 32 bits are stored in	
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	0 0 1 0 1	Rz	

Figure 15.141: MULACSRX.S16.E

15.142 MULSCA.S16.E – 16-bit signed multiply-chain add-accumulate-subtract instruction with extended operation

make

Unified command	
syntax	mulsca.s16.e rz, rx, ry
operation	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} - Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0] The compiled result only has 32-bit instructions.
	mulsca.s16.e rz, rx, ry

Description:	The lower halfword of Rx is multiplied by the lower halfword of Ry, the upper halfword of Rx is multiplied by the upper halfword of Ry, and the two multiplication results are added and combined. (Rz+1, Rz) Cumulative subtraction. The high 32 bits of the subtraction result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} - Rx[31:16] X Ry[31:16] - Rx[15:0] X Ry[15:0] Syntax: mulsca.s16.e rz, rx, ry Description:
Multiply the lower halfword of Rx by the lower	
halfword of Ry, multiply the upper halfword of Rx by the upper halfword of Ry, add the two multiplication results and sum them.	
	(Rz+1, Rz) Cumulative subtraction. The high 32 bits of the subtraction result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	0 0 1 1 1	Rz	

Figure 15.142: MULSCA.S16.E

15.143 MULSCAX.S16.E -- 16-bit signed cross-multiply chained add-accumulate instruction**with extend operation**

Unified command	
syntax	mulscax.s16.e rz, rx, ry
operation	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} - Rx[31:16] X Ry[15:0] - Rx[15:0] X Ry[31:16] The compiled result only has 32-bit instructions.
	mulscax.s16.e rz, rx, ry

Description:	The lower halfword of Rx is multiplied by the upper halfword of Ry, and the upper halfword of Rx is multiplied by the lower halfword of Ry. The two multiplication results are added and combined. (Rz+1, Rz) Cumulative subtraction. The high 32 bits of the subtraction result are stored in Rz+1, and the low 32 bits are stored in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	{Rz+1[31:0],Rz[31:0]} = {Rz+1[31:0],Rz[31:0]} - Rx[31:16] X Ry[15:0] - Rx[15:0] X Ry[31:16] Syntax: mulscax.s16.e rz, rx, ry Description:
Multiply the lower halfword of Rx by the upper	
halfword of Ry, and multiply the upper halfword of Rx by the lower halfword of Ry. Add the two multiplication results and sum them together.	
(Rz+1, Rz) Cumulative subtraction. The high 32 bits of the subtraction result are stored in Rz+1, and the low 32 bits are stored in	
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 0 0 0 1 1	1 0 1 0 1	Rz	

Figure 15.143: MULSCAX.S16.E

15.144 PSABSA.U8 - 8-bit Parallel Unsigned Subtract Absolute Value Chain Add Instruction

Unified	
instruction syntax	psabsa.u8 rz, rx, ry
Operation	$Rz[31:0] = \text{abs}(Rx[7:0] - Ry[7:0]) + \text{abs}(Rx[15:8] - Ry[15:8]) + \text{abs}(Rx[23:16] - Ry[23:16]) + \text{abs}(Rx[31:24] - Ry[31:24])$ Only 32-bit instructions exist. psabsa.u8 rz, rx, ry
Compilation results	

Description:	In bytes, subtract the 4 bytes of Rx from the 4 bytes of Ry, take the absolute value of the subtraction result, and then chain-add the result. The chain-add result is stored
Affected flag:	in Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	$Rz[31:0] = \text{abs}(Rx[7:0] - Ry[7:0]) + \text{abs}(Rx[15:8] - Ry[15:8]) + \text{abs}(Rx[23:16] - Ry[23:16]) + \text{abs}(Rx[31:24] - Ry[31:24])$ Syntax: psabsa.u8 rz, rx, ry Description: In
bytes, subtract the 4 bytes of Rx from the 4	
bytes of Ry, take the absolute value of the subtraction result, and then chain addition.	The result of addition is stored
Affected flag:	in Rz. No effect
Restriction:	None
Exception:	None

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 1 0 0 0	0 0 x 1 0	Rz	

Figure 15.144: PSABSA.U8

15.145 PSABSAA.U8—8-bit parallel signed subtraction absolute value chain addition and accumulation instruction

Unified	
instruction syntax	psabsaa.u8 rz, rx, ry
operation	$Rz[31:0] = Rz[31:0] + \text{abs}(Rx[7:0] - Ry[7:0]) + \text{abs}(Rx[15:8] - Ry[15:8]) + \text{abs}(Rx[23:16] - Ry[23:16]) + \text{abs}(Rx[31:24] - Ry[31:24])$ Only 32-bit
Compilation results	instructions exist. psabsaa.u8 rz, rx, ry

Description:	In bytes, subtract the 4 bytes of Rx from the 4 bytes of Ry, take the absolute value of the subtraction result, and then chain-add the chain-add result and Rz, and store it in
Affected flag:	Rz. No effect
Restriction:	None
Exception:	None

32-bit	
instruction operation:	$Rz[31:0] = Rz[31:0] + \text{abs}(Rx[7:0] - Ry[7:0]) + \text{abs}(Rx[15:8] - Ry[15:8]) + \text{abs}(Rx[23:16] - Ry[23:16]) + \text{abs}(Rx[31:24] - Ry[31:24])$ Syntax: psabsaa.u8 rz, rx, ry Description:
In bytes, subtract the 4 bytes of Rx from the	
4 bytes of Ry, take the absolute value of the subtraction result, and then chain add. The chain addition result is accumulated with Rz and stored in	
Affected flags:	Rz. No effect
Restrictions:	No
exceptions:	No
Instruction format :	

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 1 1 0	Ry	Rx	1 1 1 0 0 0	0 1 x 1 0	Rz	

Figure 15.145: PSABSAA.U8

15.146 DIVSL -- Signed Long Division Instruction

Unified command	
syntax	divsl rz, rx, ry Signed
operation	division $\{Rz+1[31:0], Rz[31:0]\} = \{Rx+1[31:0], Rx[31:0]\} / Ry[31:0]$
The compiled result	contains only 32-bit instructions. divsl rz, rx, ry

explain bright:	{Rx+1,Rx} forms a 64-bit signed number, which is divided by the value of RY. The upper 32 bits of the result are stored in Rz+1, and the lower 32 bits are stored in Rz. The values of Rx+1, Ry, and Rz+1 are all considered signed numbers. Note that when 0x80000000_00000000 is divided by 0xFFFF FFFF, the result is undefined. No effect
Affected flags:	
	none
Restrictions: Exceptions:	Division by zero exception

32 Bit	
	Signed division $\{Rz+1[31:0], Rz[31:0]\} = \{Rx+1[31:0], Rx[31:0]\} / Ry[31:0]$ divsl32 rz, rx, ry
	{Rx+1,Rx} forms a 64-bit signed number, which is divided by the value of RY. The upper 32 bits of the result are stored in Rz+1, and the lower 32 bits are stored in Rz. The syntax Ry, Ry+1, and Rz+1 are all considered signed numbers. Note that when 0x80000000_00000000 is divided by 0xFFFF FFFF, the result is undefined. No effect
Affected flags:	
	none
Restrictions: Exceptions:	Division by zero exception

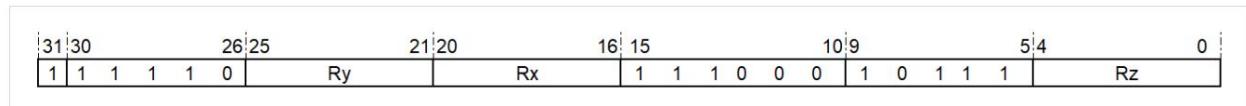
Instruction format:

Figure 15.146: DIVSL

15.147 DIVUL -- Unsigned Long Division Instruction

Unified command	
syntax	divul rz, rx, ry unsigned
operation	division $\{Rz+1[31:0], Rz[31:0]\} = \{Rx+1[31:0], Rx[31:0]\} / Ry[31:0]$ The compiled result only contains 32-bit
instructions	divul rz, rx, ry

Description:	Rx+1, Rx) is formed into a 64-bit unsigned number, divided by the value of RY, and the high 32 bits of the division result are stored in Rz+1, and the low 32 bits are stored in Rz. The values of Rx+1, Ry and Rz+1 are all considered unsigned numbers. No effect
Affected flags:	
Restrictions:	No
exceptions:	Division by zero exception

32-bit	
instruction operation:	Unsigned division {Rz+1[31:0], Rz[31:0]} = {Rx+1[31:0], Rx[31:0]} / Ry[31:0] Syntax: divul rz, rx, ry
Description:	Rx+1, Rx) is divided into
	a 64-bit unsigned number and divided by the value of RY. The high 32 bits of the division result are stored in Rz+1 and the low 32 bits are stored in Rz. The values of Rx+1, Ry and Rz+1 are all considered unsigned numbers. No effect
Affected flags:	
Restrictions:	No
exceptions:	Division by zero
exception	
Instruction format :	

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 1 0 Ry Rx 1 1 1 0 0 0 1 0 0 1 1 Rz						

Figure 15.147: DIVUL

15.148 MULACA.S8 – 8-bit parallel signed multiply chain add accumulate instruction

Unified command	
syntax	mulaca.s8 rz, rx, ry Rz = Rx[7:0]
operation	X Ry[7:0] + Rx[15:8] X Ry[15:8] + Rx[23:16] X Ry[23:16] + Rx[31:24] X Ry[31:24] The compiled result only contains 32-bit instructions. mulaca.s8 rz, rx, ry

	In byte units, the 4 bytes of Rx are multiplied by the 4 bytes of Ry respectively, and the multiplication results are chain-thickened and stored in Rz.
Description: Impact flag: No impact	
Restrictions: None	
Abnormal: None	

32-bit instructions	
operate:	Rz = Rx[7:0] * Ry[7:0] + Rx[15:8] * Ry[15:8] + Rx[23:16] * Ry[23:16] + Rx[31:24] * Ry[31:24];
Syntax:	mulaca.s8 rz, rx, ry
	In byte units, the 4 bytes of Rx are multiplied by the 4 bytes of Ry respectively, and the multiplication results are chain-thickened and stored in Rz.
Description: Affects flags: impact	
Bit:	
Limitations:	none
Exceptions:	none

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 1 1 0	Ry	Rx	1 1 1 0 0 0	1 0 0 1 1		Rz

Figure 15.148: MULACA.S8

15.149 BLOOP——Body of loop acceleration instruction

unified	
Chemical	
make	
Syntax	bloop rx, label1, label2
Operation	<p>Rx: Loop count register</p> <p>label1: the starting address of the loop body</p> <p>label2: the end address of the loop body</p> <p>Determine whether Rx is equal to 0. If not, repeatedly execute the program fragment on [label1, label2]. The number of executions is determined by Rx.</p> <p>The value is determined.</p>
Compile	Only 32-bit instructions exist.
result	bloop rx, label1, label2

Illustrate :	The last two instructions of a typical loop body are: loop count decrement instruction and comparison jump instruction. The BLOOP instruction is used to accelerate the loop body, and its function is to avoid executing the loop count decrement instruction and comparison jump instruction in the loop body. In terms of specific usage, the program fragment on [label1, label2] should not include the loop count register decrement instruction and comparison jump instruction. None
Affected flags:	
	The program fragment on [label1, label2] should not modify Rx, otherwise the program result is unpredictable. The number of loop iterations Rx is not greater than 0x10000. None
Restrictions: Exceptions: Instruction format :	

32 Bit	
instruction operation address label1: loop body	Rx: loop count register label1: loop body whether Rx is equal to 0. If not, repeatedly execute the program fragment on [label1, label2]. The execution count is determined by the value of Rx. bloop rx, label1, label2
Grammar : bright:	SayThe last two instructions of a typical loop body are: loop count decrement instruction and comparison jump instruction. The BLOOP instruction is used to accelerate the loop body, and its function is to avoid executing the loop count decrement instruction and comparison jump instruction in the loop body. In terms of specific usage, the program fragment on [label1, label2] should not include the loop count register decrement instruction and comparison jump instruction. None
Affected flags:	
	The program fragment on [label1, label2] should not modify Rx, otherwise the program results will be unpredictable. The number of loop iterations Rx is not greater than 0x10000.
Restrictions: Exceptions:	none

Instruction format:

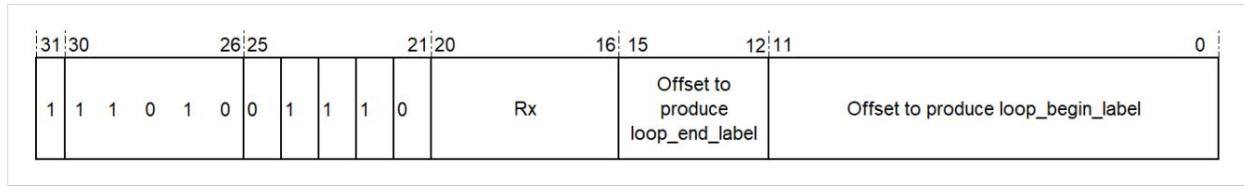


Figure 15.149: BLOOP

15.150 LDBI.W -- Word Load Instruction with Incremented Address

Unified instructions	
Syntax	ldbi.w rz, (rx)
Operations	Rz[31:0] <- mem(Rx) Rx[31:0] <- Rx[31:0] + 4
The compiled result	contains only 32-bit instructions. ldbi.w rz, (rx)

	Load a word from the address corresponding to Rx and store it in Rz. At the same time, add 4 to Rx and store it in Rx.
Description:	Impact flag: No impact
Limitations:	none
Exceptions:	Access Error Exception

32-bit instructions	
operate:	Rz[31:0] <- mem(Rx) Rx[31:0] <- Rx[31:0] + 4
Syntax:	ldbi.w rz, (rx)
Description:	Load a word from the address corresponding to Rx and store it in Rz. At the same time, add 4 to Rx and store it in Rx.
Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

Instruction format:

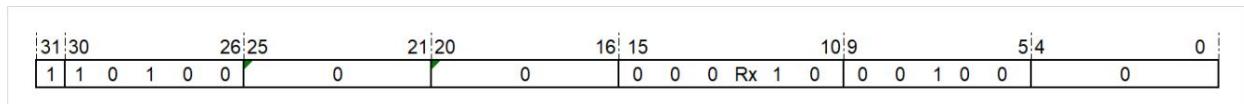


Figure 15.150: LDBI.W

15.151 LDBI.H -- Unsigned Load Halfword Instruction with Incrementing Address

Unified instructions	
Syntax	ldbi.h rz, (rx)
	Rz[31:0] <- zero_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 2
Operation Compiled results	contain only 32-bit instructions. ldbi.h rz, (rx)

	Load a half word from the address corresponding to Rx, fill the upper 16 bits with 0 and store it in Rz, and at the same time add 2 to Rx and store it in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	Rz[31:0] <- zero_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 2
Syntax:	ldbi.h rz, (rx)
	Load a half word from the address corresponding to Rx, fill the upper 16 bits with 0 and store it in Rz, and at the same time add 2 to Rx and store it in Rx.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	Access Error Exception

Instruction format:

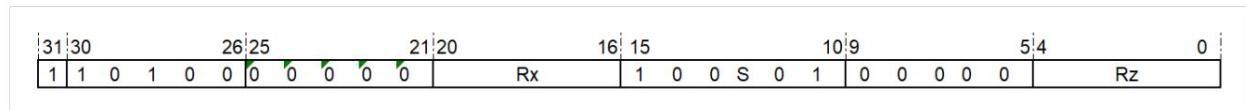


Figure 15.151: LDBI.H

15.152 LDBI.HS -- Signed Load Halfword with Incrementing Address

Unified instructions	
Syntax	ldbi.hs rz, (rx)
Operations	Rz[31:0] <- sign_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 2
The compiled result	contains only 32-bit instructions. ldbi.hs rz, (rx)

Description:	Load a half word from the address corresponding to Rx, extend the sign bit of the upper 16 bits, and store the extended result in Rz. Add 2 and store in Rx.
Impact Signs	No impact
Bit:	
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	Rz[31:0] <- sign_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 2
Syntax:	ldbi.hs rz, (rx)
Description:	Load a half word from the address corresponding to Rx, sign-extend the upper 16 bits, and store the extended result in Rz. Add 2 and store in Rx.
Impact Signs	No impact
Bit:	
Restrictions:	None
Exception:	Access error exception

Instruction format:

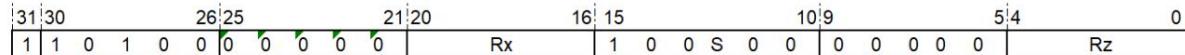


Figure 15.152: LDBI.HS

15.153 LDBI.B -- Load Unsigned Byte with Incrementing Address

Unified instructions	
Syntax	ldbi.b rz, (rx)
Operations	Rz[31:0] <- zero_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 1
The compiled result	contains only 32-bit instructions. ldbi.b rz, (rx)

	Load a byte from the address corresponding to Rx, fill the upper 24 bits with zeros and store them in Rz. At the same time, Rx is incremented by 1 and stored in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	Rz[31:0] <- zero_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 1
Syntax:	ldbi.b rz, (rx)
	Load a byte from the address corresponding to Rx, fill the upper 24 bits with zeros and store them in Rz. At the same time, Rx is incremented by 1 and stored in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

Instruction format:

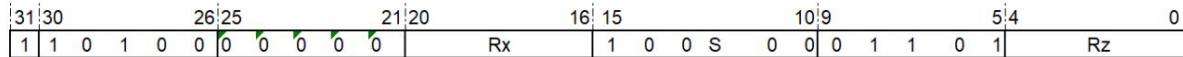


Figure 15.153: LDBI.B

15.154 LDBI.BS -- Load Signed Byte with Incrementing Address

Unified instructions	
Syntax	ldbi.bs rz, (rx)
Operations	Rz[31:0] <- sign_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 1
The compiled result	contains only 32-bit instructions. ldbi.bs rz, (rx)

Description:	Load a byte from the address corresponding to Rx, sign-extend the upper 24 bits, and store the extended result in Rz. Add 1 and store in Rx.
Impact Signs	No impact
Bit:	
Restrictions:	None
Exception:	Access error exception

32-bit instruction	
operation:	Rz[31:0] <- sign_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + 1 ldbi.bs rz,
Syntax:	(rx)
Description:	Load a byte from the address corresponding to Rx, extend the sign bit of the upper 24 bits, store the extended result in Rz, and add 1 to Rx and store it in
Affected	Rx. No effect
flags : Restrictions:	
No exceptions:	Access error exceptions

Instruction format:

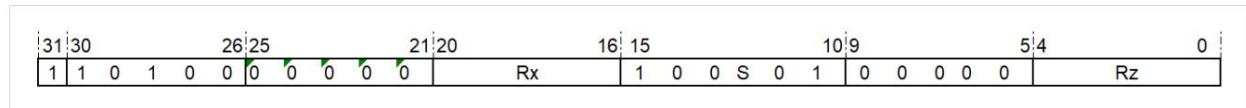


Figure 15.154: LDBI.BS

15.155 PLDBI.D -- Load Double Word with Incremented Address

Unified	
instruction syntax	pldbi.d rz, (rx)
operation	Rz[31:0] <- mem(Rx) Rz+1[31:0] <- mem(Rx+4) Rx[31:0] <- Rx[31:0] + 8 and pre-fetch mem(Rx+8), the data on mem(Rx+12) is stored in the processor's pre-fetch cache. Only 32-bit
Compilation results	instructions exist. pldbi.d rz, (rx)

Description:	Load a word from the address corresponding to Rx and store it in Rz, load a word from the address corresponding to Rx+1 and store it in Rz+1, and add 8 to Rx and store it in Rx. This instruction will also pre-fetch the data on mem(Rx+8) and mem(Rx+12) and store them in the processor's pre-fetch cache.
Affected flags :	cache. No impact
Restrictions:	
None Exceptions:	Unaligned access exception, Access error exception
exception	
Instruction format:	

32-bit	
instruction	operation: Rz[31:0] <- mem(Rx) Rz+1[31:0] <- mem(Rx+4) Rx[31:0] <- Rx[31:0] + 8 and pre-fetch mem(Rx+8), the data on mem(Rx+12) is stored in the processor's pre-fetch cache
Syntax: pldbi.d rz, (rx) Description:	Load a word from the address corresponding to Rx and store it in Rz, load a word from the address corresponding to Rx+1 and store it in Rz+1, and add 8 to Rx and store it in Rx. This instruction will also pre-fetch the data on mem(Rx+8) and mem(Rx+12) and store them in the processor's pre-fetch cache. No effect
Affected flags :	
Restrictions:	
None Exceptions:	Unaligned access exception, Access error exception

Instruction format:

31	30	26	25	21	20	16	15	10	9	5	4	0
1	1	0	1	0	0	0	0	0	Rx	1	0	0

Figure 15.155: PLDBI.D

15.156 STBI.W——Word storage instruction with self-increment address

Unified instructions	
Syntax	stbi.w rz, (rx)
Operations	mem(Rx) <- Rz[31:0] Rx[31:0] <- Rx[31:0] + 4
The compiled result	contains only 32-bit instructions. stbi.w rz, (rx)

	Store Rz into the address corresponding to Rx, and add 4 to Rx and store it into Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	mem(Rx) <- Rz[31:0] Rx[31:0] <- Rx[31:0] + 4
Syntax:	stbi.w rz, (rx)
	Store Rz into the address corresponding to Rx, and add 4 to Rx and store it into Rx.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	Access Error Exception

Instruction format:

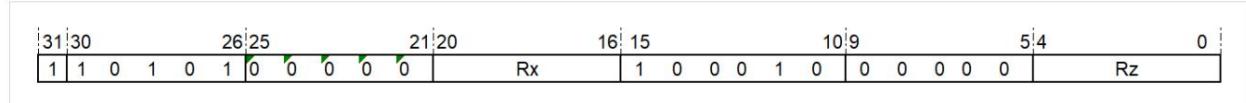


Figure 15.156: STBI.W

15.157 STBI.H --- half-word store instruction with auto-increment address

Unified instructions	
Syntax	stbi.h rz, (rx)
Operations	mem(Rx) <- Rx[15:0] Rx[31:0] <- Rx[31:0] + 2
The compiled result	contains only 32-bit instructions. stbi.h rz, (rx)

	The lower half word of Rz is stored in the address corresponding to Rx, and Rx is incremented by 2 and then stored in Rx.
Description: Impact flag: No impact	
Restrictions: None	
Exception: Access error exception	

32-bit instructions	
operate:	mem(Rx) <- Rx[15:0] Rx[31:0] <- Rx[31:0] + 2
Syntax: stbi.h rz, (rx)	
Description: Store the lower half word of Rz into the address corresponding to Rx, and add 2 to Rx and store it into Rx.	
Impact flag: No impact	
Restrictions: None	
Exception: Access error exception	

Instruction format:

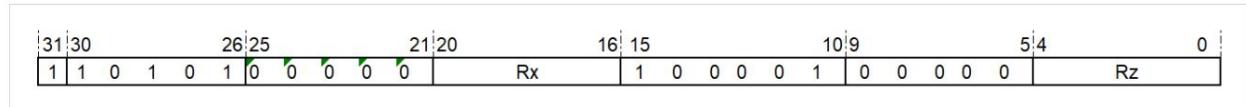


Figure 15.157: STBI.H

15.158 STBI.B -- Byte Store Instruction with Incrementing Address

Unified instructions	
Syntax	stbi.b rz, (rx)
Operations	mem(Rx) <- Rx[7:0] Rx[31:0] <- Rx[31:0] + 1
The compiled result	contains only 32-bit instructions. stbi.b rz, (rx)

	The lower 8 bits of Rz are stored in the address corresponding to Rx, and Rx is incremented by 1 and then written into Rx.
Description: Impact flag: No impact	
Restrictions: None	
Exception: Access error exception	

32-bit instructions	
operate:	mem(Rx) <- Rx[7:0] Rx[31:0] <- Rx[31:0] + 1
Syntax:	stbi.b rz, (rx)
	The lower 8 bits of Rz are stored in the address corresponding to Rx, and Rx is incremented by 1 and then written into Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

Instruction format:

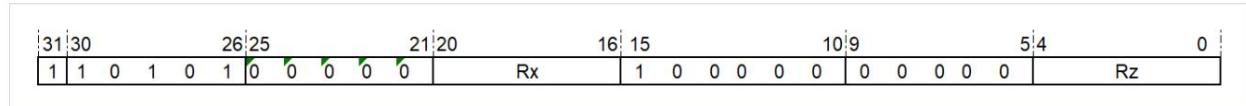


Figure 15.158: STBI.B

15.159 LDBIR.W - Word load instruction with register address increment

Unified instructions	
Syntax	ldbir.w rz, (rx), ry
Operations	Rz[31:0] <- mem(Rx) Rx[31:0] <- Rx[31:0] + Ry[31:0]
The compiled result	contains only 32-bit instructions. ldbir.w rz, (rx), ry

	Load a word from the address corresponding to Rx and store it in Rz. At the same time, add Rx to Ry and store it in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	Rz[31:0] <- mem(Rx) Rx[31:0] <- Rx[31:0] + Ry[31:0]
Syntax:	ldbir.w rz, (rx), ry
	Load a word from the address corresponding to Rx and store it in Rz. At the same time, add Rx to Ry and store it in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 0 1 0 0	Ry	Rx	1 0 1 0 1 0	0 0 0 0 0 0	Rz	

Figure 15.159: LDBIR.W

15.160 LDBIR.H -- Unsigned halfword load instruction with incremented register address

Unified instructions	
Syntax	ldbir.h rz, (rx), ry
Operation	$Rz[31:0] \leftarrow \text{zero_extend}(\text{mem}(Rx))$ $Rx[31:0] \leftarrow Rx[31:0] + Ry[31:0]$
The compiled result	contains only 32-bit instructions. ldbir.h rz, (rx), ry

	Load a half word from the address corresponding to Rx, fill the upper 16 bits with 0 and store it in Rz. At the same time, add Rx to Ry and store it in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	$Rz[31:0] \leftarrow \text{zero_extend}(\text{mem}(Rx))$ $Rx[31:0] \leftarrow Rx[31:0] + Ry[31:0]$
Syntax:	ldbir.h rz, (rx), ry
Description:	Load a half word from the address corresponding to Rx, fill the upper 16 bits with 0 and store it in Rz, and add Rx to Ry and store it in Rx.
Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 0 1 0 0	Ry	Rx	1 0 1 S 0 1	0 0 0 0 0 0	Rz	

Figure 15.160: LDBIR.H

15.161 LDBIR.HS -- Signed Load Halfword with Increment Register Address

The unified	
	Idbir.hs rz, (rx), ry Rz[31:0]
	<- zero_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + Ry[31:0]
instruction syntax operation compilation result	only has 32-bit instructions.
	Idbir.hs rz, (rx), ry

Description:	Load a half word from the address corresponding to Rx, extend the sign bit of the upper 16 bits, store the extended result in Rz, and add Rx to Ry and store it in
Affected	Rx. No effect
flags : Restrictions:	
No exceptions:	Access error exceptions

32-bit	
instruction operation	Rz[31:0] <- zero_extend(mem(Rx)) Rx[31:0] <- Rx[31:0] + Ry[31:0] Syntax:
Idbir.hs rz, (rx), ry	Description: Load a half word from the address corresponding to Rx, extend the sign bit of the upper 16 bits, store the extended result in Rz, and add Rx to Ry and store it in
	Rx. No effect
Affected	
flags : Restrictions:	
No exceptions:	Access error exceptions

Instruction format:

31:30	26:25	21:20	16:15	10:9	5:4	0
1 1 0 1 0 0	Ry	Rx	1 0 1 S 0 1	0 0 0 0 0	Rz	

Figure 15.161: LDBIR.HS

15.162 LDBIR.B -- Unsigned Byte Load Increment Register Address

Unified instructions	
Syntax	<code>Idbir.b rz, (rx), ry</code>
	$Rz[31:0] \leftarrow \text{zero_extend}(\text{mem}(Rx))$ $Rx[31:0] \leftarrow Rx[31:0] + Ry[31:0]$
Operation Compiled results	contain only 32-bit instructions. <code>Idbir.b rz, (rx), ry</code>

	Load a byte from the address corresponding to Rx, fill the upper 24 bits with zeros and store them in Rz. At the same time, add Rx to Ry and store them in Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	$Rz[31:0] \leftarrow \text{zero_extend}(\text{mem}(Rx))$ $Rx[31:0] \leftarrow Rx[31:0] + Ry[31:0]$
Syntax:	<code>Idbir.b rz, (rx), ry</code>
	Load a byte from the address corresponding to Rx, fill the upper 24 bits with zeros and store them in Rz. At the same time, add Rx to Ry and store them in Rx.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	Access Error Exception

Instruction format:

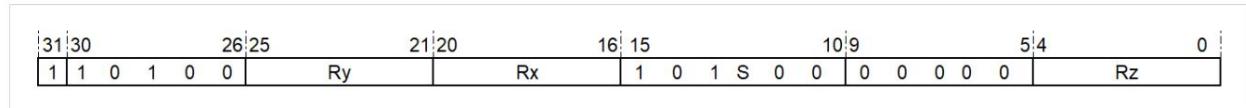


Figure 15.162: LDBIR.B

15.163 LDBIR.BS -- Signed Byte Load Increment Register Address

Unified instructions	
Syntax	<code>Idbir.bs rz, (rx), ry</code>
	$Rz[31:0] \leftarrow \text{sign_extend}(\text{mem}(Rx))$ $Rx[31:0] \leftarrow Rx[31:0] + Ry[31:0]$
Operation Compiled results	contain only 32-bit instructions. <code>Idbir.bs rz, (rx), ry</code>

Description:	Load a byte from the address corresponding to Rx, extend the sign bit of the upper 24 bits, store the extended result in Rz, and add Rx to Ry and store it in Rx.
Affected	No effect
flags : Restrictions:	
No exceptions:	Access error exceptions

32-bit	
instruction operation:	$Rz[31:0] \leftarrow \text{sign_extend}(\text{mem}(Rx))$ $Rx[31:0] \leftarrow Rx[31:0] + Ry[31:0]$ ldbir.bs rz,
Syntax:	(rx), ry
Description:	Load a byte from the address corresponding to Rx, extend the sign bit of the upper 24 bits, store the extended result in Rz, and add Rx to Ry and store it in Rx.
Affected	No effect
flags : Restrictions:	
No exceptions:	Access error exceptions

Instruction format:

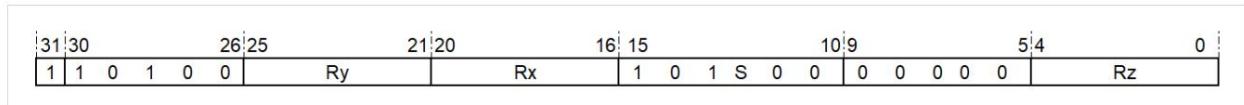


Figure 15.163: LDBIR.BS

15.164 PLDBIR.D -- Double-word load instruction with register address increment

Unified	
instruction syntax	pldbir.d rz, (rx) , ry
operation	$Rz[31:0] \leftarrow \text{mem}(Rx)$ $Rz+1[31:0] \leftarrow \text{mem}(Rx+Ry)$ $Rx[31:0] \leftarrow Rx[31:0] + 2XRy$ and prefetch the data on $\text{mem}(Rx+2XRy)$, $\text{mem}(Rx+3XRy)$ and store it in the processor's prefetch cache. Only 32-bit instructions
Compilation results	exist. pldbir.d rz, (rx), ry

Description:	Load a word from the address corresponding to Rx and store it in Rz, load a word from the address corresponding to Rx+Ry and store it in Rz+1, and add 2XRy to Rx and store it in Rx. This instruction will also pre-fetch the data on mem(Rx+2XRy) and mem(Rx+3XRy) and store them in the processor's pre-fetch cache.
Affected flags :	No impact
Restrictions:	
None Exceptions:	Unaligned access exception, Access error exception

32-bit	
instruction operation:	Rz[31:0] <- mem(Rx) Rz+1[31:0] <- mem(Rx+Ry) Rx[31:0] <- Rx[31:0] + 2XRy and pre-fetch the data on mem(Rx+2XRy), mem(Rx+3XRy) and store them in the processor's pre-fetch cache. Syntax: pldbir.d rz, (rx),
ry Description:	Load a word from the address corresponding to Rx and store it in Rz, load a word from the address corresponding to Rx+Ry and store it in Rz+1, and add 2XRy to Rx and store it in Rx. This instruction will also pre-fetch the data on mem(Rx+2XRy), mem(Rx+3XRy) and store them in the processor's pre-fetch cache. No effect
Affected flags :	
Restrictions:	
None Exceptions:	Unaligned access exception, Access error exception

Instruction format:

31 30	26 25	21 20	16 15	10 9	5 4	0
1 1 0 1 0 0 Ry Rx 1 0 1 0 1 1 0 0 0 0 0 Rz						

Figure 15.164: PLDBIR.D

15.165 STBIR.W——Word storage instruction with register address increment

Unified instructions	
Syntax	stbir.w rz, (rx) , ry
Operations	mem(Rx) <- Rx[31:0] Rx[31:0] <- Rx[31:0] + Ry[31:0]
The compiled result	contains only 32-bit instructions. stbir.w rz, (rx) , ry

	Store Rz into the address corresponding to Rx, and add Rx to Ry and store it into Rx.
Description: Impact flag:	No impact
Restrictions:	None
Exception:	Access error exception

32-bit instructions	
operate:	mem(Rx) <- Rx[31:0] Rx[31:0] <- Rx[31:0] + Ry[31:0]
Syntax:	stbir.w rz, (rx) , ry
	Store Rz into the address corresponding to Rx, and add Rx to Ry and store it into Rx.
Description: Impact flag:	No impact
Limitations:	none
Exceptions:	Access Error Exception

Instruction format:

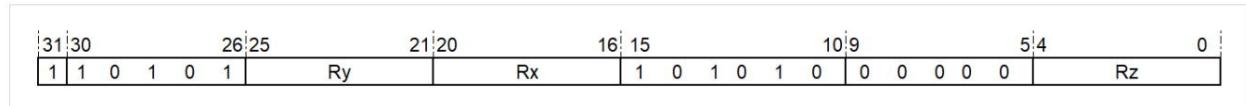


Figure 15.165: STBIR.W

15.166 STBIR.H——Half-word store instruction with register address increment

Unified instructions	
Syntax	stbir.h rz, (rx) , ry
Operations	mem(Rx) <- Rx[15:0] Rx[31:0] <- Rx[31:0] + Ry[31:0]
The compiled result	contains only 32-bit instructions. stbir.h rz, (rx) , ry

	The lower half word of Rz is stored in the address corresponding to Rx, and Rx plus Ry is stored in Rx.
Description: Impact flag: No impact	
Restrictions: None	
Exception: Access error exception	

32-bit instructions	
operate:	mem(Rx) <- Rx[15:0] Rx[31:0] <- Rx[31:0] + Ry[31:0]
Syntax: stbir.h rz, (rx) ry	
Description: Store the lower half word of Rz into the address corresponding to Rx, and add Rx to Ry and store it into Rx.	
Impact flag: No impact	
Restrictions: None	
Exception: Access error exception	

Instruction format:

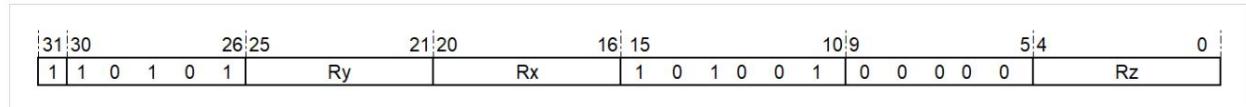


Figure 15.166: STBIR.H

15.167 STBIR.B——Byte Store Instruction with Register Address Increment

Unified instructions	
Syntax	stbir.b rz, (rx) , ry
Operations	mem(Rx) <- Rx[7:0] Rx[31:0] <- Rx[31:0] + Ry[31:0]
The compiled result	contains only 32-bit instructions. stbir.b rz, (rx) , ry

	The lower 8 bits of Rz are stored in the address corresponding to Rx, and Rx plus Ry is stored in Rx.
Description: Impact flag: No impact	
Restrictions: None	
Exception: Access error exception	

32-bit instructions	
operate:	mem(Rx) <- Rx[7:0] Rx[31:0] <- Rx[31:0] + Ry[31:0]
Syntax:	stbir.b rz, (rx) , ry
	The lower 8 bits of Rz are stored in the address corresponding to Rx, and Rx plus Ry is stored in Rx.
Description: Impact	flag: No impact
Restrictions: None	
Exception: Access error	exception

Instruction format:

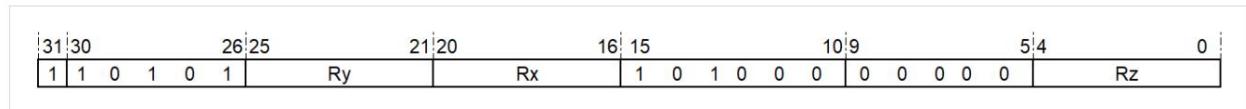


Figure 15.167: STBIR.B