XBurst® Instruction Set Architecture

MIPS extension/enhanced SIMD V3

Programming Manual

Release Date: Oct 11, 2019

北京君正集成电路股份有限公司

Ingenic Semiconductor Co.,Ltd.

XBurst® Instruction Set Architecture

Programming Manual

Disclaimer

This documentation is provided for use with Ingenic products. No license to Ingenic property rights is granted. Ingenic assumes no liability, provides no warranty either expressed or implied relating to the usage, or intellectual property right infringement except as provided for by Ingenic Terms and Conditions of Sale.

Ingenic products are not designed for and should not be used in any medical or life sustaining or supporting equipment.

All information in this document should be treated as preliminary. Ingenic may make changes to this document without notice. Anyone relying on this documentation should contact Ingenic for the current documentation and errata.

Ingenic Semiconductor Co., Ltd.

Ingenic Headquarters, East Bldg. 14, Courtyard #10
Xibeiwang East Road, Haidian District, Beijing, China,
Tel: 86-10-56345000
Fax:86-10-56345001
Http: //www.ingenic.com

CONTENTS

1 Overview of XBurst® ISA

Basically XBurst® CPU's Instruction Set Architecture (ISA) is fully compatible with MIPS32 ISA. Please refer to the following documentations:

- MD00082-2B-MIPS32INT-AFP-03.50 provides an introduction to the MIPS32 Architecture
- MD00086-2B-MIPS32BIS-AFP-03.51 provides detailed descriptions of the MIPS32 instruction set
- MD00090-2B-MIPS32PRA-AFP-03.50 defines the behavior of the privileged resources

Moreover, the SIMD extensions introduced into the XBurst® ISA as the enhanced MIPS32 ISA is called MIPS eXtension/enhanced Unit (MXU). Up to now, MXU has evolved to version 3 (MXU3).The MXU3 instruction set is encoded with field codes which are available to licensed MIPS partner. The MXU3 floating point implementation is compliant with the IEEE Standard 754-2008 for Floating point Arithmetic. All standard floating point operations are provided for 32-bit and 64-bit floating point data.

XBurst® MXU3 instruction set is designed by Ingenic to address the need by video, graphical, image, signal processing which has inherent parallel computation feature. This document provides detailed description of XBurst® MXU3 instruction set.

2 MXU3 Programming Model

This chapter describes MXU3 Programming model, including the following sections:

- MXU3 Data Formats
- MXU3 Reigster File
- MXU3 Control Registers
- MXU3 Extception

2.1 MXU3 Data Formats

The MXU3 instructions support the following data types:

- 1-bit, 2-bit, 4-bit, 8-bit, 16-bit, 32-bit, 64-bit, 128-bit, 256-bit, 512-bit, 1024-bit signed and unsigned integers
- 32-bit single-precision

2.2 MXU3 Register File

In MXU3, a vector purpose register file(VPR) is composed of 32 MLEN*[1] bit general purpose registers: vr0 ~ vr31. And a vector sum register cluster (VSR) is composed of 4 MLEN bit accumulating purpose registers: vs0 ~ vs3. In addition, a vector write purpose register file(VWR) is composed of MLEN/32 32-bit vector write purpose registers. In fact,VWR is the alias of vr31 for dedicated write purpose, that is, vw0 is the 0th word in vr31, vw1 is the 1th word in vr31,and so on.

*[1] Please refer to Definitions for Instruction Description for meaning of MLEN.

2.3 MXU3 Control Registers

There are two control registers that can be accessed through CTCMXU and CFCMXU instructions. They are:

MIR - MXU implementation and revision register

MCSR - MXU control and status register

2.3.1 MXU Implementation Register (MIR, register 0)

The MIR Register is a 32-bit read-only register.

MIR

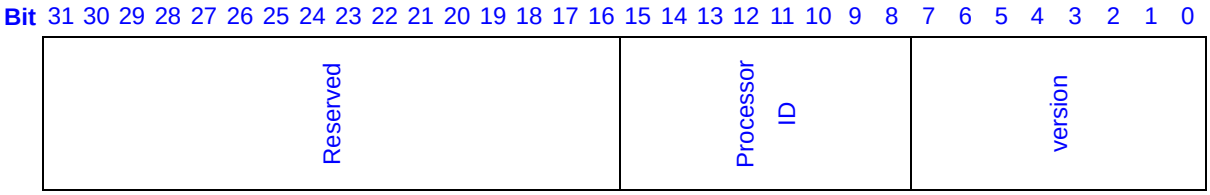


Table 2-1 MXU_MIR Register Field Description

Bits	Name	Description	R/W
31:16	Reserved	Writing has no effect, read as zero.	R
15:8	Processor ID	Processor ID number	R
7:0	Version	Version number.	R

2.3.2 MXU Control and Status Register (MCSR, register 31)

MCSR

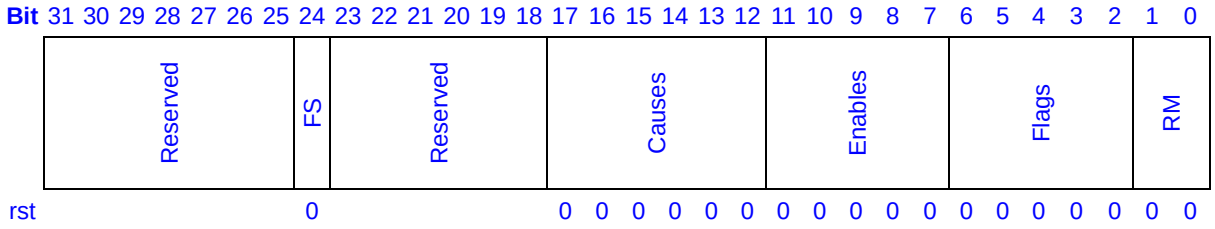


Table 2-2 MXU_MCSR Register Field Description

Bits	Name	Description	R/W
31:25	Reserved	Writing has no effect, read as zero.	R
24	FS	Flush to zero for denormalized number result. 0: do not flush to zero; 1: flush to zero.	RW
23:18	Reserved	Writing has no effect, read as zero.	R
17:12	Cause	Cause bits. These bits indicate the exception conditions arise during the execution of FPU arithmetic instructions. Setting 1 if corresponding exception condition arises otherwise setting 0.	RW
11:7	Enables	Enable exception (in IEEE754, enable trap) bits. The exception shall occur when both enable bit and corresponding cause bit are set during the execution of an arithmetic FPU instruction.	RW
6:2	Flags	Flag bits. When an arithmetic FPU operation arises an exception condition but does not trigger exception due to corresponding Enable bit is set value 0, then the corresponding bit in the Flag field is set value 1, while the others remains unchanged. The value of Flag field can last until software explicitly modifies it.	RW
1:0	RM	Round mode. 0: round to nearest. 1: round toward zero. 2: round toward $+\infty$. 3: round toward $-\infty$.	RW

2.4 Exceptions

Execution of the MXU3 instructions can generate the following exceptions:

Reserved Instruction

If Config1.C2 (CP0 Register 16, Select 1, bit 6) is not set, executing an MXU3 instruction will trigger a Reserved Instruction exception with the ExcCode field of CP0 Cause register being set to 0xa.

Coprocessor Unusable

If Status.CU2 (CP0 Register 12, Select 0, bit 30) is not set and Config1.C2 is set, executing an MXU3 instruction will trigger a Coprocessor Unusable exeption with the ExcCode field of CP0 Cause register being set to 0xb as well as CE field being set to 0x2.

MXU3 Floating point Exception

Like MIPS floating point instruction, an MXU3 floating point instruction may trigger a floating point exception withthe ExcCode field of CP0 Cause register being set to 0x10.

Address Error

an load or store access to an protected address space will trigger an Address Error exception with the ExcCode field of CP0 Cause register being set to 0x4(load) or 0x5(store).

TLB Exception

an load or store access trigger an TLB-relative exception, the reason may be TLB page miss, TLB page invalid, reading an Read-Inhibited page or writing a clean page.

Table 2-3 MXU3 Exception Code Values

Mnemonic	ExcCode		Description
	Decimal	Hexadecimal	
Modify	1	0x1	TLB Modification exception (store)
TLBL	2	0x2	TLB exception (load)
TLBS	3	0x3	TLB exception (store)
AdEL	4	0x04	Address error exception(load)
AdES	5	0x05	Address error exception(store)
RI	10	0x0a	Reserved Instruction exception
CpU	11	0x0b	Coprocessor Unusable exception
MFPE	16	0x10	MXU floating point exception

3 MXU3 Instruction Set

The MXU3 ISA is implemented via Coprocessor2. All MXU3 instructions are encoded in the COP2 or SPECIAL2 major opcode space. This chapter describes MXU3 Instruction in detail, including the following sections:

3.1 Instruction Summary

The MXU3 assembly language coding rule uses the following syntax abbreviations:

Table 3-1 supported syntax abbreviations

Abbreviation	Descrption
fmt	data format, which could be: 1-bit(1BI), 2-bit(2BI), 4-bit(4BI), a 8-bit byte(B), 16-bit halfword(H), 32-bit Word(W), 64-bit Double-word(D), 128-bit Quadruple-word(Q), 256-bit Octuple-word(O), or 512-bit heXadeca-word(X)
vrd	destination vector register
vrs,vrp	source vector register
rs, rd	general purpose register
fs, fd	floating point register
mcsrs, mcsrd	MXU control and status register
imm	immediate value
offset	should be sign-extended and added to the contents of the base register to form an effective address
base	the contents of GPR base
index	the contents of GPR index
vrs[n], vrd[m]	element <n|m> in vector register

Table 3-2 Valid Element Index Values

Data Format	Valid Element Index for SMID256	Valid Element Index for SMID512	Valid Element Index for SMID1024
1BI	n=0,…,255	n=0,…,511	n=0,…,1023
2BI	n=0,…,127	n=0,…,255	n=0,…,511
4BI	n=0,…,63	n=0,…,127	n=0,…,255
B	n=0,…,31	n=0,…,63	n=0,…,127
H	n=0,…,15	n=0,…,31	n=0,…,63
W	n=0,…,7	n=0,…,15	n=0,…,31
D	n=0,…,3	n=0,…,7	n=0,…,15
Q	n=0,1	n=0,…,3	n=0,…,7
O	n=0	n=0,1	n=0,…,3
X		n=0	n=0,1

3.2 Definitions for Instruction Description

This section defines some useful self-defined concepts and macro functions that will be frequently referenced in later chapters for convenience of description.

Table 3-3 common abbreviations and macro functions

Name	Descrption
VPR[x]	vector operand register x
MLEN	Maximum bit length of MXU3, it is implementation dependent, valid value may be: 256 for SIMD256, or 512 for SIMD512, or 1024 for SIMD1024
esize	bit size of element, including: 1BI(1), 2BI(2), 4BI(4), B(8), H(16), W(32), D(64), Q(128), O(256), X(512)
X'Baa	X bits, aa is a bit string
{X'Baa,Y'Bbb}	(X+Y)bits, return concatenated bit string
N{X'Baa}	(N*X)bits, return N copies of bit string
op[esize,i]	return op[(i+1)*esize-1, i*esize]
=	Assignment
!=	Logical inequality
==	Logical equality
+ - * /	Arithmetic operators
?:	Conditional select
&&	Logical and
||	Logical or
~	Bitwise negation
&	Bitwise and

|	Bitwise inclusive or
^	Bitwise exclusive or
signal_exception(x)	signal exception x
check_cop2_enable()	if !Config1.C2 signal_exception(RI) elif Config1.C2 && !Status.CU2 Cause.CE=2'B10 signal_exception(CpU)
check_fp_exception()	if (MCSR.enables & MCSR.cause) != 0 signal_exception(MFPE)
signed(x)	x should be regarded as a signed number
unsigned(x)	x should be regarded as an unsigned number
sign_extend32(x)	make a sign-extension for x to get a 32-bit signed number
zero_extend32(x)	make a zero-extension for x to get a 32-bit unsigned number
abs(x)	Get the absolute value of x
fpadd(x,y)	return the result of additive value by the floating point values x and y. The operation is performed according to the IEEE-754-2008.
fpsub(x,y)	return the result of subtracting value by the floating point values x and y. The operation is performed according to the IEEE-754-2008.
fpmul(x,y)	return the result of multiplied value by the floating point values x and y. The operation is performed according to the IEEE-754-2008.
fpabs(x)	return the absolute value of the floating point value x. The operation is performed according to the IEEE-754-2008.
fpmax(x,y)	return the maximum value of the floating point values x and y. The operation is performed according to the IEEE-754-2008.
fpmin(x,y)	return the minimum value of the floating point values x and y. The operation is performed according to the IEEE-754-2008.
fpclass(x)	return the class mask of the floating point value x. The operation is performed according to the IEEE-754-2008.
fpeq(x,y)	Compare x with y, if thay are ordered and equal, return all bits 1, else return all bits 0. The operation is performed according to the IEEE-754-2008.
fple(x,y)	Compare x with y, if thay are ordered and the value of x is less than or equal to the value of y, return all bits 1,else return all bits 0. The operation is performed according to the IEEE-754-2008.
fplt(x,y)	Compare x with y, if thay are ordered and the value of x is less than the value of y, return all bits 1, else return all bits 0. The operation is performed according to the IEEE-754-2008.
fpor(x,y)	if the value of x and y both are ordered, return all bits 1, else return all bits 0. The operation is performed according to the IEEE-754-2008.
sint_to_single(x)	return single-precision format of signed integer.
uint_to_single(x)	return single-precision format of unsigned integer.
single_to_sint(x)	return signed integer format of single-precision.

single_to_uint(x)	return signed integer format of single-precision.
single_rto_sint(x)	return single-precision format of rounded single-precision.
single_tto_sint(x)	return signed integer format of truncated single-precision.
single_tto_uint(x)	return unsigned integer format of truncated single-precision.

3.3 List of MXU3 Instructions

From the Table 3-4 to the Table 3-14 provide a list of instructions group by category. The Individual instruction descriptions in following tables.

Table 3-4 Branch Instructions

Mnemonic	Assembler Format
BNEZB, BNEZH, BNEZW	vrs, offset
BNEZV	vrs, offset
BEQZB, BEQZH, BEQZW	vrs, offset
BEQZV	vrs, offset

Table 3-5 Compare Instructions

Mnemonic	Assembler Format
CEQB, CEQH, CEQW	vrd, vrs, vrp
CEQZB, CEQZH, CEQZW	vrd, vrs
CLESB, CLESH, CLESW	vrd, vrs, vrp
CLEUB, CLEUH, CLEUW	vrd, vrs, vrp
CLEZB, CLEZH, CLEZW	vrd, vrs
CLTSB, CLTSH, CLTSW	vrd, vrs, vrp
CLTUB, CLTUH, CLTUW	vrd, vrs, vrp
CLTZB, CLTZH, CLTZW	vrd, vrs

Table 3-6 Integer Arithmetic Instructions

Mnemonic	Assembler Format
ADDB, ADDH, ADDW	vrd, vrs, vrp
ADDIW	vwrd, vwrp, imm
ADDRW	vwrd, vwrp
SUBB, SUBH, SUBW	vrd, vrs, vrp
WADDSBL, WADDSBH, WADDSHL, WADDSHH	vrd, vrs, vrp
WADDUBL, WADDUBH, WADDUHL, WADDUHH	vrd, vrs, vrp
WSUBSBL, WSUBSBH, WSUBSHL, WSUBSHH	vrd, vrs, vrp
WSUBUBL, WSUBUBH, WSUBUHL, WSUBUHH	vrd, vrs, vrp
SR1SUM2BI, SR2SUM2BI, SR4SUM2BI, SR8SUM2BI, SR16SUM2BI, SR32SUM2BI, SR1SUM4BI, SR2SUM4BI, SR4SUM4BI, SR8SUM4BI, SR16SUM4BI, SR32SUM4BI, SR1SUMUB, SR2SUMUB, SR4SUMUB, SR8SUMUB, SR16SUMUB,	vsd[m], vrs

SR32SUMUB, SR1SUMUH, SR2SUMUH, SR4SUMUH, SR8SUMUH, SR16SUMUH, SR32SUMUH	
SR1SUMSB, SR2SUMSB, SR4SUMSB, SR8SUMSB, SR16SUMSB, SR32SUMSB, SR1SUMSH, SR2SUMSH, SR4SUMSH, SR8SUMSH, SR16SUMSH, SR32SUMSH	vsd[m], vrs
SR1SUMW, SR2SUMW, SR4SUMW, SR8SUMW, SR16SUMW, SR32SUMW	vsd[m], vrs
ABSB, ABSH, ABSW	vrd, vrs
MULB,MULH,MULW	vrd, vrs, vrp
SMULBE, SMULBO, SMULHE, SMULHO	vrd, vrs, vrp
UMULBE UMULBO, UMULHE, UMULHO	vrd, vrs, vrp
WSMULBL, WSMULBH, WSMULHL, WSMULHH	vrd, vrs, vrp
WUMULBL, WUMULBH, WUMULHL, WUMULHH	vrd, vrs, vrp
MLAW	vsd, vrs, vrp
MLSW	vsd, vrs, vrp
SMLAHE, SMLAHO	vsd, vrs, vrp
SMLSHE, SMLSHO	vsd, vrs, vrp
WSMLAHL, WSMLAHH	vsd, vrs, vrp
WSMLSHL, WSMLSHH	vsd, vrs, vrp
SR1MAC2BI, SR2MAC2BI, SR4MAC2BI, SR8MAC2BI, SR16MAC2BI, SR32MAC2BI, SR1MAC4BI, SR2MAC4BI, SR4MAC4BI, SR8MAC4BI, SR16MAC4BI, SR32MAC4BI, SR1MACUUB, SR2MACUUB, SR4MACUUB, SR8MACUUB, SR16MACUUB, SR32MACUUB	vsd[m], vrs, vrp[n]
SR1MACSUB, SR2MACSUB, SR4MACSUB, SR8MACSUB, SR16MACSUB, SR32MACSUB,	vsd[m], vrs, vrp[n]
SR1MACSSB, SR2MACSSB, SR4MACSSB, SR8MACSSB, SR16MACSSB, SR32MACSSB, SR1MACSSH, SR2MACSSH, SR4MACSSH, SR8MACSSH, SR16MACSSH, SR32MACSSH	vsd[m], vrs, vrp[n]
S1MACUUB, S2MACUUB, S4MACUUB, S8MACUUB, S16MACUUB, S32MACUUB	vsd[m], vrs, vrp
S1MACSUB, S2MACSUB, S4MACSUB, S8MACSUB, S16MACSUB, S32MACSUB	vsd[m], vrs, vrp
S1MACSSB, S2MACSSB, S4MACSSB, S8MACSSB, S16MACSSB, S32MACSSB S1MACSSH, S2MACSSH, S4MACSSH, S8MACSSH, S16MACSSH, S32MACSSH	vsd[m], vrs, vrp
MAXAB,MAXAH,MAXAW	vrd, vrs, vrp

MAXSB,MAXSH,MAXSW	vrd, vrs, vrp
MAXU2BI, MAXU4BI, MAXUB,MAXUH,MAXUW	vrd, vrs, vrp
MINAB,MINAH,MINAW	vrd, vrs, vrp
MINSB,MINSH,MINSW	vrd, vrs, vrp
MINU2BI, MINU4BI, MINUB,MINUH,MINUW	vrd, vrs, vrp
SATSSWH, SATSSWB, SATSSHB	vrd, vrs
SATSUWH, SATSUWB, SATSUW4BI, SATSUW2BI, SATSUHB, SATSUH4BI, SATSUH2BI, SATSUB4BI, SATSUB2BI	vrd, vrs
SATUUB4BI, SATUUB2BI SATUUHB, SATUUH4BI, SATUUH2BI SATUUWH, SATUUWB, SATUUW4BI	vrd, vrs
TOCB, TOCH, TOCW	vsd[0], vrs

Table 3-7 Bitwise Instructions

Mnemonic	Assembler Format
ANDV	vrd, vrs, vrp
ANDNV	vrd, vrs, vrp
ANDIB	vrd, vrs, imm
ORV	vrd, vrs, vrp
ORNV	vrd, vrs, vrp
ORIB	vrd, vrs, imm
XORV	vrd, vrs, vrp
XORNV	vrd, vrs, vrp
XORIB	vrd, vrs, imm
BSELV	vrd, vrs, vrp

Table 3-8 Floating Point Arithmetic Instructions

Mnemonic	Assembler Format
FADDW	vrd, vrs, vrp
FSUBW	vrd, vrs, vrp
FMULW	vrd, vrs, vrp
FCMULRW	vrd, vrs, vrp
FCMULIW	vrd, vrs, vrp
FCADDW	vrd, vrs, vrp
FXAS1W, FXAS2W, FXAS4W, FXAS8W, FXAS16W	vrd, vrp
FMAXW	vrd, vrs, vrp
FMAXAW	vrd, vrs, vrp
FMINW	vrd, vrs, vrp
FMINAW	vrd, vrs, vrp
FCLASSW	vrd, vrs

Table 3-9 Floating Point Compare Instructions

Mnemonic	Assembler Format
FCEQW	vrd, vrs, vrp
FCLEW	vrd, vrs, vrp
FCLTW	vrd, vrs, vrp
FCORW	vrd, vrs, vrp

Table 3-10 Floating Point Conversion Instructions

Mnemonic	Assembler Format
FFSIW	vrd, vrs
FFUIW	vrd, vrs
FTSIW	vrd, vrs
FTUIW	vrd, vrs
FRINTW	vrd, vrs
FTRUNCSW	vrd, vrs
FTRUNCUW	vrd, vrs

Table 3-11 Shift Instructions

Mnemonic	Assembler Format
SLLB,SLLH,SLLW	vrd, vrs, vrp
SLLIB,SLLIH,SLLIW	vrd, vrs, imm
SRAB,SRAH,SRAW	vrd, vrs, vrp
SRAIB,SRAIH,SRAIW	vrd, vrs, imm
SRARB,SRARH,SRARW	vrd, vrs, vrp
SRARIB,SRARIH,SRARIW	vrd, vrs, imm
SRLB,SRLH,SRLW	vrd, vrs, vrp
SRLIB,SRLIH,SRLIW	vrd, vrs, imm
SRLRB,SRLRH,SRLRW	vrd, vrs, vrp
SRLRIB,SRLRIH,SRLRIW	vrd, vrs, imm

Table 3-12 Element Shuffle/Permute Instructions

Mnemonic	Assembler Format
GT1BI, GT2BI, GT4BI, GTB, GTH	vrd, vrp
GT2W, GT4W, GT8W, GT16W, GT2D, GT4D, GT8D, GT2Q, GT4Q, GT2O	vrd[m], vrp, pos
EXTUWLL, EXTUWLH, EXTUWHL, EXTUWHH, EXTUDLL, EXTUDLH, EXTUDHL, EXTUDHH, EXTUQLL, EXTUQLH, EXTUQHL, EXTUQHH, EXTUOLL, EXTUOLH, EXTUOHL, EXTUOHH, EXTUXLL, EXTUXLH, EXTUXHL, EXTUXHH	vrd, vrp, w
EXTU1BIL, EXTU2BIL, EXTU4BIL, EXTUBL, EXTUHL, EXTU1BIH,EXTU2BIH, EXTU4BIH, EXTUBH, EXTUHH	vrd, vrp
EXTS1BIL, EXTS2BIL, EXTS4BIL, EXTSBL, EXTSHL, EXTS1BIH, EXTS2BIH, EXTS4BIH, EXTSBH, EXTSHH	vrd, vrp
EXTU3BW	vrd, vrp

REPIB,REPIH,REPIW,REPID,REPIQ,REPIO,REPIX	vrd, vrp[n]
MOVW, MOVD, MOVQ, MOVO, MOVX	vrd[m], vrp[n]
SHUFW2, SHUFW4, SHUFW8, SHUFW16, SHUFD2, SHUFD4, SHUFD8, SHUFQ2, SHUFQ4, SHUFO2	vrd, vrp
GSHUFW	vrd, vrs, vrp, imm
GSHUFWB	vrd, vrs, vrp
GSHUFB	vrd, vrs, vrp
GSHUFWH	vrd, vrs, vrp
GSHUFH	vrd, vrs, vrp
ILVE2BI, ILVE4BI, ILVEB, ILVEH, ILVEW, ILVED, ILVEQ, ILVEO, ILVEX	vrd, vrs, vrp
ILVO2BI, ILVO4BI, ILVOB, ILVOH, ILVOW, ILVOD, ILVOQ, ILVOO, ILVOX	vrd, vrs, vrp
BSHLI	vrd, vrp, imm
BSHRI	vrd, vrp, imm
BSHL	vrd, vrs, vrp
BSHR	vrd, vrs, vrp

Table 3-13 Register Load and misc Instructions

Mnemonic	Assembler Format
MTCPUW	rd, vwrp
MFCPUW	vwrd, rs
MTFPUW	fd, vwrp
MFFPUW	vwrd, fs
CTCMXU	mcsrd, rs
CFCMXU	rd, mcsrs
SUMZ	vsd
MFSUM	vrd, vss
MFSUMZ	vrd, vsd
MTSUM	vsd,vrs
MXSUM	vrd, vrs, vsd
LIH	vrd, imm
LIW	vrd, imm
LIWH	vrd, imm
LIWR	vwrd, imm
CMVW	vrd, vrs, vrp, imm
PMAPH, PMAPW	vrd, vrs, vrp

Table 3-14 Memory Load/Store Instructions

Mnemonic	Assembler Format

LUW, LUD, LUQ, LUO	vrd[n], base
LAW, LAD, LAQ, LAO	vrd[n], offset(base)
SUW, SUD, SUQ, SUO	vrp[n], base
SAW, SAD, SAQ, SAO	vrp[n], offset(base)
LUO2B, LUO2H, LUO2W, LUO2D, LUO2Q, LUO4B, LUO4H, LUO4W, LUO4D, LUO4Q	vrd[n], p, base
LAO2B, LAO2H, LAO2W, LAO2D, LAO2Q, LAO4B, LAO4H, LAO4W, LAO4D, LAO4Q	vrd[n], p, base
SUO2W, SUO2D, SUO2Q, SUO4W, SUO4D, SUO4Q	vrp[n], p, base
SAO2W, SAO2D, SAO2Q, SAO4W, SAO4D, SAO4Q	vrp[n], p, base
LUW2B, LUW2H, LUW4B, LUW4H, LUW8B, LUW8H, LUW16B, LUW16H, LUW32B, LUW32H	vrd[n], p, base
LAW2B, LAW2H, LAW4B, LAW4H, LAW8B, LAW8H, LAW16B, LAW16H, LAW32B, LAW32H	vrd[n], p, base
LUD2B, LUD2H, LUD2W, LUD4B, LUD4H, LUD4W, LUD8B, LUD8H, LUD8W, LUD16B, LUD16H, LUD16W	vrd[n], p, base
LAD2B, LAD2H, LAD2W, LAD4B, LAD4H, LAD4W, LAD8B, LAD8H, LAD8W, LAD16B, LAD16H, LAD16W	vrd[n], p, base
SUD2W, SUD4W, SUD8W, SUD16W	vrp[n], p, base
SAD2W, SAD4W, SAD8W, SAD16W	vrp[n], p, base
LUQ2B, LUQ2H, LUQ2W, LUQ2D, LUQ4B, LUQ4H, LUQ4W, LUQ4D, LUQ8B, LUQ8H, LUQ8W, LUQ8D	vrd[n], p, base
LAQ2B, LAQ2H, LAQ2W, LAQ2D, LAQ4B, LAQ4H, LAQ4W, LAQ4D, LAQ8B, LAQ8H, LAQ8W, LAQ8D	vrd[n], p, base
SUQ2W, SUQ2D, SUQ4W, SUQ4D, SUQ8W, SUQ8D	vrp[n], p, base
SAQ2W, SAQ2D, SAQ4W, SAQ4D, SAQ8W, SAQ8D	vrp[n], p, base

Table 3-15 NNA Instructions

Mnemonic	Assembler Format
NNRWR	vwrp, imm
NNRRD	vrd, imm
NNDWR	vrp, imm
NNDRD	vrd, imm
NNCMD	imm
NNMAC	vwrp, imm

3.4 Encoding Category of MXU3 Instructions

COP2

Bit	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
	COP2						Minor opcode					vrp					vrs					vrd					funct					

SPECIAL2

Bit	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
	SPECIAL2						rs					rt					rd					funct					Minor opcode					

3.5 Branch

3.5.1 BNEZ<fmt>

Branch if all elements are not equal to zero **COP2**

Bit	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
	010010								10011				vrs				offset										0101				fmt	

Syntax:

BNEZB	vrs, offset	//fmt=2'B00
BNEZH	vrs, offset	//fmt=2'B01
BNEZW	vrs, offset	//fmt=2'B10

Description：

PC-relative branch if all elements in the vrs are not equal to zero. The branch instruction has a delay slot. Offset is a signed count of 32-bit instructions based from the PC of the delay slot. A branch should not be placed in the delay slot of the instruction.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [31:0] imm={20{offset[9]},offset[9:0],2'B00}}
bit cond=1
for i in MLEN/esize
    cond    = cond && (op1[esize,i] != 0)
if cond:
    I:
    I+1:pc  = pc + imm
```

Exceptions:

RI, CpU

3.5.2 BNEZV

Branch if vector value is not equal to zero **COP2**

Bit	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
	010010	10011	vrs	offset	0101	11

Syntax:

BNEZV vrs, offset

Description：

PC-relative branch if at least one bit in the vrs is not zero, The branch instruction has a delay slot. Offset is a signed count of 32-bit instructions based from the PC of the delay slot. A branch should not be placed in the delay slot of the instruction.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [31:0] imm={20{offset[9]},offset[9:0],2'B00}
bit cond=(op1 != 0)
if cond:
    I:
    I+1:pc    = pc + imm
```

Exceptions:

RI, CpU

3.5.3 BEQZ<fmt>

Branch if at least one element is equal to zero **COP2**

Bit	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
	010010	10011	vrs	offset	0100	fmt

Syntax:

BEQZB	vrs, offset	//fmt=2'B00
BEQZH	vrs, offset	//fmt=2'B01
BEQZW	vrs, offset	//fmt=2'B10

Description：

PC-relative branch if at least one element in the vrs is zero,. The branch instruction has a delay slot. Offset is a signed count of 32-bit instructions based from the PC of the delay slot. A branch should not be placed in the delay slot of the instruction.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [31:0] imm={20{offset[9]},offset[9:0],2'B00}
bit cond=0
for i in MLEN/esize
    cond    = cond || (op1[esize,i] == 0)
if cond:
    I:
    I+1:pc  = pc + imm
```

Exceptions:

RI, CpU

3.5.4 BEQZV

Branch if vector value is equal to zero **COP2**

Bit	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
	010010	10011	vrs	offset	0100	11

Syntax:

BEQZV vrs, offset

Description：

PC-relative branch if all bits in the vrs are zero. The branch instruction has a delay slot. Offset is a signed count of 32-bit instructions based from the PC of the delay slot. A branch should not be placed in the delay slot of the instruction.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [31:0] imm={20{offset[9]},offset[9:0],2'B00}
bit cond=(op1 == 0)
if cond:
    I:
    I+1:pc    = pc + imm
```

Exceptions:

RI, CpU

3.6 Compare

3.6.1 CEQ<fmt>

Compare Equal **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1010	fmt

Syntax:

CEQB	vrd, vrs, vrp	//fmt=2'B00
CEQH	vrd, vrs, vrp	//fmt=2'B01
CEQW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs, and compare it with the corresponding one in the vrp. If they are equal, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{op1[esize,i] == op2[esize,i]}
```

Exceptions:

RI, CpU

3.6.2 CEQZ<fmt>

Compare Equal to Zero **COP2**

Bit	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
	010010	10000	vrs	00000	vrd	1000	fmt

Syntax:

CEQZB	vrd, vrs	//fmt=2'B00
CEQZH	vrd, vrs	//fmt=2'B01
CEQZW	vrd, vrs	//fmt=2'B10

Description：

Take each element in the vrs, and compare it with zero. If the compared result is true, then the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{op1[esize,i] == 0}
```

Exceptions:

RI, CpU

3.6.3 CLES<fmt>

Compare Less-Than-or-Equal Signed **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1111	fmt

Syntax:

CLESB	vrd, vrs, vrp	//fmt=2'B00
CLESH	vrd, vrs, vrp	//fmt=2'B01
CLESW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs and the corresponding one in the vrp for signed comparison. If the element in the vrs is less than or equal to the corresponding one in the vrp, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{signed(op1[esize,i]) <= signed(op2[esize,i])}
```

Exceptions:

RI, CpU

3.6.4 CLEU<fmt>

Compare Less-Than-or-Equal Unsigned **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1110	fmt

Syntax:

CLEUB	vrd, vrs, vrp	//fmt=2'B00
CLEUH	vrd, vrs, vrp	//fmt=2'B01
CLEUW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs and the corresponding one in the vrp for unsignd comparison. If the element in the vrs is less than or equal to the corresponding one in the vrp, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{unsigned(op1[esize,i]) <= unsigned(op2[esize,i])}
```

Exceptions:

RI, CpU

3.6.5 CLEZ<fmt>

Compare Less-Than-or-Equal to Zero **COP2**

Bit	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
	010010	10000	vrs	00000	vrd	1011	fmt

Syntax:

CLEZB	vrd, vrs	//fmt=2'B00
CLEZH	vrd, vrs	//fmt=2'B01
CLEZW	vrd, vrs	//fmt=2'B10

Description：

Take each element in the vrs to cmpare with zero for signed comparison. If the element in the vrs is less than or equal to zero, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{signed(op1[esize,i]) <= 0}
```

Exceptions:

RI, CpU

3.6.6 CLTS<fmt>

Compare Less-Than Signed **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1101	fmt

Syntax:

CLTSB	vrd, vrs, vrp	//fmt=2'B00
CLTSH	vrd, vrs, vrp	//fmt=2'B01
CLTSW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs and the corresponding one in the vrp for signed comparison. If the element in the vrs is less than the corresponding one in the vrp, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{signed(op1[esize,i]) < signed(op2[esize,i])}
```

Exceptions:

RI, CpU

3.6.7 CLTU<fmt>

Compare Less-Than Unsigned **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1100	fmt

Syntax:

CLTUB	vrd, vrs, vrp	//fmt=2'B00
CLTUH	vrd, vrs, vrp	//fmt=2'B01
CLTUW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs and the corresponding one in the vrp for unsigned comparison. If the element in the vrs is less than the corresponding one in the vrp, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{unsigned(op1[esize,i]) < unsigned(op2[esize,i])}
```

Exceptions:

RI, CpU

3.6.8 CLTZ<fmt>

Compare Less-Than to Zero **COP2**

Bit	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
	010010	10000	vrs	00000	vrd	1001	fmt

Syntax:

CLTZB	vrd, vrs	//fmt=2'B00
CLTZH	vrd, vrs	//fmt=2'B01
CLTZW	vrd, vrs	//fmt=2'B10

Description：

Take each element in the vrs to compare with zero for signed comparison. If the element in the vrs is less than zero, the corresponding element in the vrd is set to all one, otherwise it is set to all zero.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/esize
    VPR[vrd][esize,i]   = esize{signed(op1[esize,i]) < 0}
```

Exceptions:

RI, CpU

3.7 Integer Arithmetic

3.7.1 ADD<fmt>

Add **COP2**

Bit	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
	010010	10100	vrs	vrp	vrd	0000	fmt

Syntax:

ADDB	vrd, vrs, vrp	//fmt=2'B00
ADDH	vrd, vrs, vrp	//fmt=2'B01
ADDW	vrd, vrs, vrp	//fmt=2'B10

Description：

Each element in the vrp is added with the corresponding one in the vrs, the add result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = op1[esize,i] + op2[esize,i]
```

Exceptions:

RI, CpU

3.7.2 ADDIW

VWR Immediate ADD **COP2**

Bit	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
	010010	11011	imm[7:3]	vwrp	vwrd	100	imm[2:0]

Syntax:

ADDIW vwrd, vwrp, imm

Description：

The immediate value from the <imm> is signed extended to 32-bit then added with the value of vwrp, the add result updates the vwrd.

Operation：

check_cop2_enable()

bit [31:0] op1=VWR[vwrp]

VWR[vwrd]= op1 + sign_extend32(imm)

Exceptions:

RI, CpU

3.7.3 ADDRW

VWR Registers ADD **COP2**

Bit	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
	010010	11011	00000	vwrp	vwrd	110100

Syntax:

ADDRW vwrd, vwrp

Description：

The value of vwrp is added with the value of vwrd, the result updates the vwrd.

Operation：

check_cop2_enable()

bit [31:0] op1=VWR[vwrp]

bit [31:0] op2=VWR[vwrd]

VWR[vwrd]= op1 + op2

Exceptions:

RI, CpU

3.7.4 SUB<fmt>

Sub						COP2
Bit 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		
010010	10100	vrs	vrp	vrd	0010	fmt

Syntax:

SUBB	vrd, vrs, vrp	//fmt=2'B00
SUBH	vrd, vrs, vrp	//fmt=2'B01
SUBW	vrd, vrs, vrp	//fmt=2'B10

Description：

Each element in the vrp is subtracted from the corresponding one in the vrs, the subtraction result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = op1[esize,i] - op2[esize,i]
```

Exceptions:

RI, CpU

3.7.5 WADDS<fmt><p>

Add Signed Widen **COP2**

Bit	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
	010010						10100				vrs					vrp					vrd					1001				p	fmt

Syntax:

WADDSBL	vrd, vrs, vrp	//p=0, fmt=0
WADDSBH	vrd, vrs, vrp	//p=1, fmt=0
WADDSHL	vrd, vrs, vrp	//p=0, fmt=1
WADDSHH	vrd, vrs, vrp	//p=1, fmt=1

Description：

The low(<p>= 0) or high(<p>= 1) half part of elements in the vrp are signed extended to twice of their original size, then each widened element is added with the corresponding twice-size one in the vrs, the add result updates the corresponding twice-size element in the vrd.

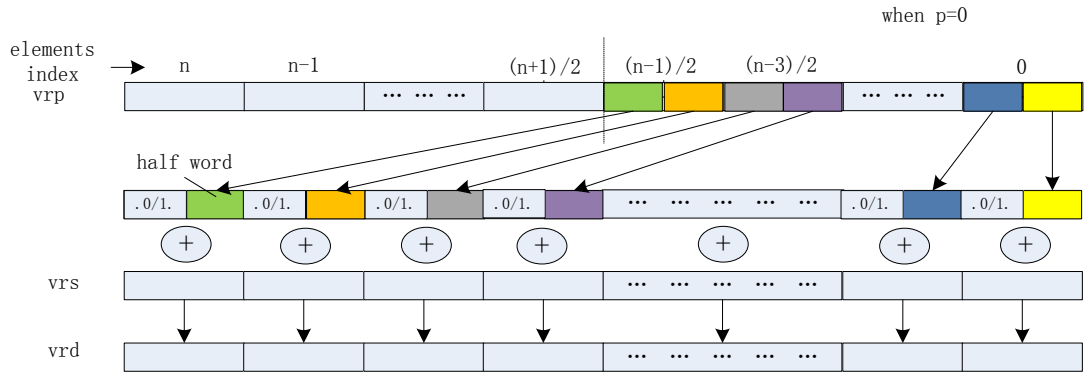
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] tmp
bit sign
N=MLEN/(2*esize)
for i in N
    tmp                 = op2[esize,i+p*N]
    sign                = tmp[esize-1]
    VPR[vrd][2*esize,i]= op1[2*esize,i] + {esize{sign}, tmp}
```

Exceptions:

RI, CpU

WADDS <fmt> <p>



3.7.6 WADDU<fmt><p>

Add Unsigned Widen **COP2**

Bit	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
	010010	10100	vrs	vrp	vrd	1000	p	fmt

Syntax:

WADDUBL	vrd, vrs, vrp	//p=0, fmt=0
WADDUBH	vrd, vrs, vrp	//p=1, fmt=0
WADDUHL	vrd, vrs, vrp	//p=0, fmt=1
WADDUHH	vrd, vrs, vrp	//p=1, fmt=1

Description：

The low(<p>= 0) or high(<p>= 1) half part of elements in the vrp are unsigned extended to twice of their original size, then each widened element is added with the corresponding twice-size one in the vrs, the add result updates the corresponding twice-size element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] tmp
N=MLEN/(2*esize)
for i in N
    tmp                    = op2[esize,i+p*N]
    VPR[vrd][2*esize,i]= op1[2*esize,i] + {esize{0}, tmp}
```

Exceptions:

RI, CpU

3.7.7 WSUBS<fmt><p>

Sub Signed Widen **COP2**

Bit	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
	010010	10100	vrs	vrp	vrd	1011	p	fmt

Syntax:

WSUBSBL	vrd, vrs, vrp	//p=0, fmt=0
WSUBSBH	vrd, vrs, vrp	//p=1, fmt=0
WSUBSHL	vrd, vrs, vrp	//p=0, fmt=1
WSUBSHH	vrd, vrs, vrp	//p=1, fmt=1

Description：

Similar with WADDS<fmt><p>, except that the operation is substraction.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] tmp
bit sign
N=MLEN/(2*esize)
for i in N
    tmp                 = op2[esize,i+p*N]
    sign                = tmp[esize-1]
    VPR[vrd][2*esize,i]= op1[2*esize,i] - {esize{sign}, tmp}
```

Exceptions:

RI, CpU

3.7.8 WSUBU<fmt><p>

Sub Unsigned Widen **COP2**

Bit	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
	010010	10100	vrs	vrp	vrd	1010	p	fmt

Syntax:

WSUBUBL	vrd, vrs, vrp	//p=0, fmt=0
WSUBUBH	vrd, vrs, vrp	//p=1, fmt=0
WSUBUHL	vrd, vrs, vrp	//p=0, fmt=1
WSUBUHH	vrd, vrs, vrp	//p=1, fmt=1

Description：

Similar with WADDU<fmt><p>, except that the operation is subtraction.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] tmp
N=MLEN/(2*esize)
for i in N
    tmp                  = op2[esize,i+p*N]
    VPR[vrd][2*esize,i]= op1[2*esize,i] - {esize{0}, tmp}}
```

Exceptions:

RI, CpU

3.7.9 SR<Lseg>SUM<fmt>

Segmented Replicate Sum Unsigned **COP2**

Bit	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
	010010	10111	vrs	00	le	0	fmt	vsd	0_m

Syntax:

SR1SUM2BI	Vsd[m], vrs	//le=3'B000, fmt=2'B00, 0_m=000000:	Lseg=1, m=0
SR2SUM2BI	vsd[m], vrs	//le=3'B001, fmt=2'B00, 0_m=00000m:	Lseg=2, m=0~1
SR4SUM2BI	vsd[m], vrs	//le=3'B010, fmt=2'B00, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUM2BI	vsd[m], vrs	//le=3'B011, fmt=2'B00, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUM2BI	vsd[m], vrs	//le=3'B100, fmt=2'B00, 0_m=00mmmm:	Lseg=16, m=0~15,
		//valid for SIMD1024 & SIMD512	
SR32SUM2BI	vsd[m], vrs	//le=3'B101, fmt=2'B00, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024
SR1SUM4BI	vsd[m], vrs	//le=3'B000, fmt=2'B01, 0_m=000000:	Lseg=1, m=0
SR2SUM4BI	vsd[m], vrs	//le=3'B001, fmt=2'B01, 0_m=00000m:	Lseg=2, m=0~1
SR4SUM4BI	vsd[m], vrs	//le=3'B010, fmt=2'B01, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUM4BI	vsd[m], vrs	//le=3'B011, fmt=2'B01, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUM4BI	vsd[m], vrs	//le=3'B100, fmt=2'B01, 0_m=00mmmm:	Lseg=16, m=0~15,
		//valid for SIMD1024 & SIMD512	
SR32SUM4BI	vsd[m], vrs	//le=3'B101, fmt=2'B01, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024
SR1SUMUB	vsd[m], vrs	//le=3'B000, fmt=2'B10, 0_m=000000:	Lseg=1, m=0
SR2SUMUB	vsd[m], vrs	//le=3'B001, fmt=2'B10, 0_m=00000m:	Lseg=2, m=0~1
SR4SUMUB	vsd[m], vrs	//le=3'B010, fmt=2'B10, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUMUB	vsd[m], vrs	//le=3'B011, fmt=2'B10, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUMUB	vsd[m], vrs	//le=3'B100, fmt=2'B10, 0_m=00mmmm:	Lseg=16, m=0~15,
		//valid for SIMD1024 & SIMD512	
SR32SUMUB	vsd[m], vrs	//le=3'B101, fmt=2'B10, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024
SR1SUMUH	vsd[m], vrs	//le=3'B000, fmt=2'B11, 0_m=000000:	Lseg=1, m=0
SR2SUMUH	vsd[m], vrs	//le=3'B001, fmt=2'B11, 0_m=00000m:	Lseg=2, m=0~1
SR4SUMUH	vsd[m], vrs	//le=3'B010, fmt=2'B11, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUMUH	vsd[m], vrs	//le=3'B011, fmt=2'B11, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUMUH	vsd[m], vrs	//le=3'B100, fmt=2'B11, 0_m=00mmmm:	Lseg=16, m=0~15,
		//valid for SIMD1024 & SIMD512	
SR32SUMUH	vsd[m], vrs	//le=3'B101, fmt=2'B11, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024

Description：

First, MLEN bits of the vrs need be divided equally into N segments, the segment here has the specific length of <Lseg> words. Then for each segment, its included elements need be unsigned extended and added together. Finally the sum result of each segment need be added to the <m>th word of the correspoding segment in the vsd.

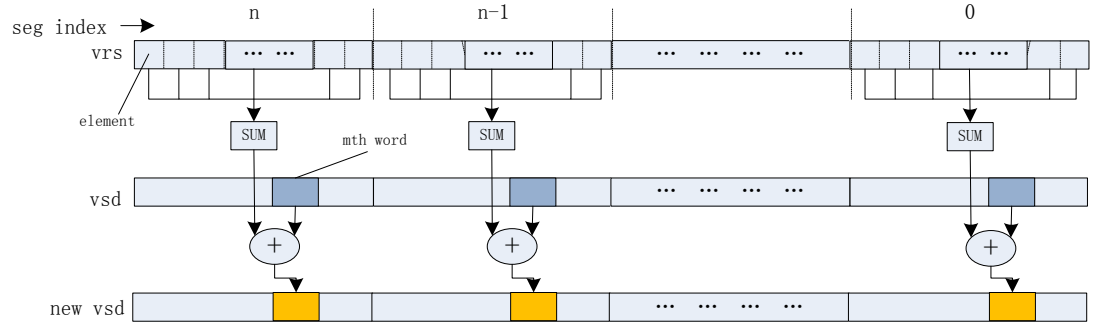
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op=VPR[vrs]
bit [31:0] sum
Nseg=MLEN/(Lseg*32)
Nelem_seg=(Lseg*32)/esize
for j in Nseg
    sum = 0
    for i in Nelem_seg
        sum = sum + {(32-esize){0},op[esize,i+j*Nelem_seg]}
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

S<Lseg>SUM<fmt>



3.7.10 SR<Lseg>SUMS<fmt>

Segmented Replicate Sum Signed **COP2**

Bit	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
	010010	10111	vrs	00	le	1	fmt	vsd	0_m

Syntax:

SR1SUMSB	vsd[m], vrs	//le=3'B000, fmt=2'B00, 0_m=000000:	Lseg=1, m=0
SR2SUMSB	vsd[m], vrs	//le=3'B001, fmt=2'B00, 0_m=00000m:	Lseg=2, m=0~1
SR4SUMSB	vsd[m], vrs	//le=3'B010, fmt=2'B00, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUMSB	vsd[m], vrs	//le=3'B011, fmt=2'B00, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUMSB	vsd[m], vrs	//le=3'B100, fmt=2'B00, 0_m=00mmmm:	Lseg=16, m=0~15,
		//valid for SIMD1024 & SIMD512	
SR32SUMSB	vsd[m], vrs	//le=3'B101, fmt=2'B00, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024
SR1SUMSH	vsd[m], vrs	//le=3'B000, fmt=2'B01, 0_m=000000:	Lseg=1, m=0
SR2SUMSH	vsd[m], vrs	//le=3'B001, fmt=2'B01, 0_m=00000m:	Lseg=2, m=0~1
SR4SUMSH	vsd[m], vrs	//le=3'B010, fmt=2'B01, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUMSH	vsd[m], vrs	//le=3'B011, fmt=2'B01, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUMSH	vsd[m], vrs	//le=3'B100, fmt=2'B01, 0_m=00mmmm:	Lseg=16, m=0~15,
		//valid for SIMD1024 & SIMD512	
SR32SUMSH	vsd[m], vrs	//le=3'B101, fmt=2'B01, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024

Description：

First, MLEN bits of the vrs need be divided equally into N segments, the segment here has the specific length of <Lseg> words. Then for each segment, its included elements need be signed extended and added together. Finally the sum result of each segment need be added to the <m>th word of the correspoding segment in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op=VPR[vrs]
bit [31:0] sum
bit [esize-1:0] tmp
bit sign
Nelem_seg=(Lseg*32)/esize
for j in MLEN/(Lseg*32)
    sum = 0
    for i in Nelem_seg
        tmp  = op[esize,i+j*Nelem_seg]
        sign = tmp[esize-1]
        sum = sum + {(32-esize){sign}, tmp}
    VSR[vsd][32,m+j*Lseg]= VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

3.7.11 SR<Lseg>SUMW

Segmented Replicate Sum Int **COP2**

Bit	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
	010010	10111	vrs	00	le	1	1	0	vsd	0_m

Syntax:

SR1SUMW	vsd[m], vrs	//le=3'B000, 0_m=000000:	Lseg=1, m=0
SR2SUMW	vsd[m], vrs	//le=3'B001, 0_m=00000m:	Lseg=2, m=0~1
SR4SUMW	vsd[m], vrs	//le=3'B010, 0_m=0000mm:	Lseg=4, m=0~3
SR8SUMW	vsd[m], vrs	//le=3'B011, 0_m=000mmm:	Lseg=8, m=0~7
SR16SUMW	vsd[m], vrs	//le=3'B100, 0_m=00mmmm:	Lseg=16, m=0~15, valid for SIMD1024 & SIMD512
SR32SUMW	vsd[m], vrs	//le=3'B101, 0_m=0mmmmm:	Lseg=32, m=0~31, valid for SIMD1024

Description：

First, MLEN bits of the vrs need be divided equally into N segments, the segment here has the specific length of <Lseg> words. Then for each segment, its included 32-bit integer elements need be added together. Finally the sum result of each segment need be added to the <m>th word of the correspoding segment in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op=VPR[vrs]
bit [31:0] sum
Nelem_seg=(Lseg*32)/esize
for j in MLEN/(Lseg*32)
    sum = 0
    for i in Nelem_seg
        sum = sum + op[esize,i+j*Nelem_seg]
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

3.7.12 ABS<fmt>

Absolute **COP2**

Bit	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
	010010	10100	vrs	00000	vrd	0110	fmt

Syntax:

ABSB	vrd, vrs	//fmt=2'B00
ABSH	vrd, vrs	//fmt=2'B01
ABSW	vrd, vrs	//fmt=2'B10

Description：

Calculate the absolute value of each element in the vrs, the result updates the coresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op=VPR[vrs]
for i in MLEN/esize
    VPR[vrd][esize,i] = abs(op[esize,i])
```

Exceptions:

RI, CpU

3.7.13 MUL<fmt>

Multiply **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	1000	fmt

Syntax:

MULB	vrd, vrs, vrp	//fmt=2'B00
MULH	vrd, vrs, vrp	//fmt=2'B01
MULW	vrd, vrs, vrp	//fmt=2'B10

Description：

The integer elements in the vrp are multiplied by corresponding ones in the vrs. Each multiplication result updates the corresponding element in the vrd. Note that each most significant half of the multiplication result is discarded.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]   = op1[esize,i] * op2[esize,i]
```

Exceptions:

RI, CpU

3.7.14 SMUL<fmt><p>

Sign Extended Multiply **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	1011	p	fmt

Syntax:

SMULBE	vrd, vrs, vrp	//p=0, fmt=0
SMULBO	vrd, vrs, vrp	//p=1, fmt=0
SMULHE	vrd, vrs, vrp	//p=0, fmt=1
SMULHO	vrd, vrs, vrp	//p=1, fmt=1

Description：

Each even(<p>= 0) or odd(<p>= 1) signed integer element in the vrp is multiplied by the corresponding signed one in the vrs, the multiplication result updates the corresponding twice-size element in the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

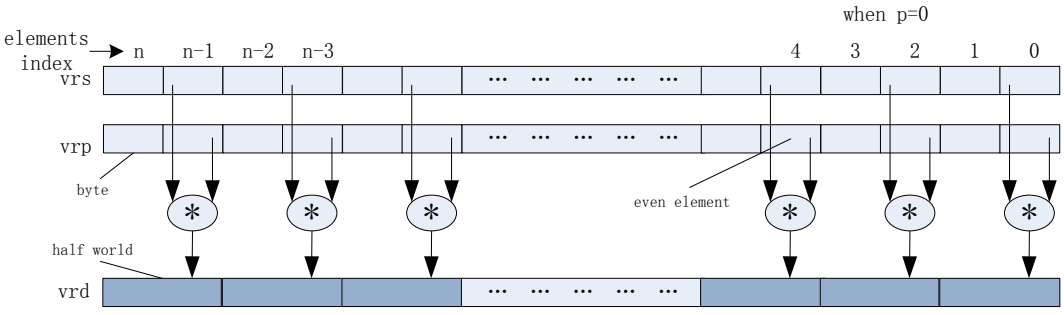
for i in MLEN/(2*esize)

VPR[vrd][2*esize,i]= signed(op1[esize,2*i+p]) * signed(op2[esize,2*i+p])

Exceptions:

RI, CpU

SMUL <fmt><p>



3.7.15 UMUL<fmt><p>

Unsign Extended Multiply **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	1010	p	fmt

Syntax:

UMULBE	vrd, vrs, vrp	//p=0, fmt=0
UMULBO	vrd, vrs, vrp	//p=1, fmt=0
UMULHE	vrd, vrs, vrp	//p=0, fmt=1
UMULHO	vrd, vrs, vrp	//p=1, fmt=1

Description：

Each even(<p>= 0) or odd(<p>= 1) unsigned integer element in the vrp is multiplied by the corresponding unsigned one in the vrs, the multiplication result updates the corresponding twice-size element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*esize)
    VPR[vrd][2*esize,i]= unsigned(op1[esize,2*i+p]) * unsigned(op2[esize,2*i+p])
```

Exceptions:

RI, CpU

3.7.16 WSMUL<fmt><p>

Sign Widen Multiply **COP2**

Bit	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
	010010	11011	vrs	vrp	vrd	1011	p	fmt

Syntax:

WSMULBL	vrd, vrs, vrp	//p=0, fmt=0
WSMULBH	vrd, vrs, vrp	//p=1, fmt=0
WSMULHL	vrd, vrs, vrp	//p=0, fmt=1
WSMULHH	vrd, vrs, vrp	//p=1, fmt=1

Description：

The low(<p>= 0) or high(<p>= 1) half part of integer elements in the vrp are signed extended to twice of their original size, then multiplied by the corresponding twice-size integer elements in the vrs. Each multiplication result updates the corresponding twice-size element in the vrd. Note that each most significant half of the multiplication result is discarded.

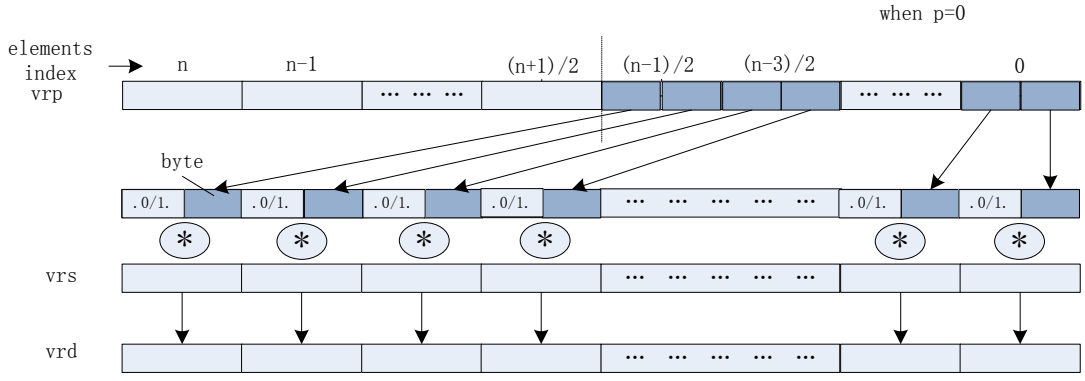
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] tmp
bit sign
N=MLEN/(2*esize)
for i in N
    tmp             = op2[esize,i+p*N]
    sign            = tmp[esize-1]
    VPR[vrd][2*esize,i]= op1[2*esize,i] * {esize{sign}, tmp}
```

Exceptions:

RI, CpU

WSMUL <fmt> <p>



3.7.17 WUMUL<fmt><p>

Unsign Widen Multiply **COP2**

Bit	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
	010010	11011	vrs	vrp	vrd	1010	p	fmt

Syntax:

WUMULBL	vrd, vrs, vrp	//p=0, fmt=0
WUMULBH	vrd, vrs, vrp	//p=1, fmt=0
WUMULHL	vrd, vrs, vrp	//p=0, fmt=1
WUMULHH	vrd, vrs, vrp	//p=1, fmt=1

Description：

The low(<p>= 0) or high(<p>= 1) half part of integer elements in the vrp are unsigned extended to twice of their original size, then multiplied by the corresponding twice-size integer elements in the vrs. Each multiplication result updates the corresponding twice-size element in the vrd. Note that each most significant half of the multiplication result is discarded.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] tmp
N=MLEN/(2*esize)
for i in N
    tmp                  = op2[esize,i+p*N]
    VPR[vrd][2*esize,i]= op1[2*esize,i] * {esize{0}, tmp}
```

Exceptions:

RI, CpU

3.7.18 MLAW

Multiply-Add to Accumulator **COP2**

Bit	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
	010010	10011	vrs	vrp	000	vsd	111000

Syntax:

MLAW vsd, vrs, vrp

Description：

Each 32-bit signed integer element in the vrp is multiplied by the corresponding signed one in the vrs, the multiplication result is added to the corresponding one in the vsd. Note that each most significant half of the multiplication result is discarded.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/32
    VSR[vsd][32,i]    = VSR[vsd][32,i] + op1[32,i] * op2[32,i]
```

Exceptions:

RI, CpU

3.7.19 MLSW

Multiply-Subtract from Accumulator **COP2**

Bit	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
	010010	10011	vrs	vrp	000	vsd	111100

Syntax:

MLSW vsd, vrs, vrp

Description：

Each 32-bit signed integer element in the vrp is multiplied by the corresponding signed one in the vrs, the multiplication result is subtracted from the corresponding one in the vsd. Note that each most significant half of the multiplication result is discarded.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/32
    VSR[vsd][32,i]    = VSR[vsd][32,i] - op1[32,i] * op2[32,i]
```

Exceptions:

RI, CpU

3.7.20 SMLAH<p>

Sign Extend Multiply-Add to Accumulator **COP2**

Bit	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
	010010	10011	vrs	vrp	000	vsd	1110	p	1

Syntax:

SMLAHE	vsd, vrs, vrp	//p=0
SMLAHO	vsd, vrs, vrp	//p=1

Description：

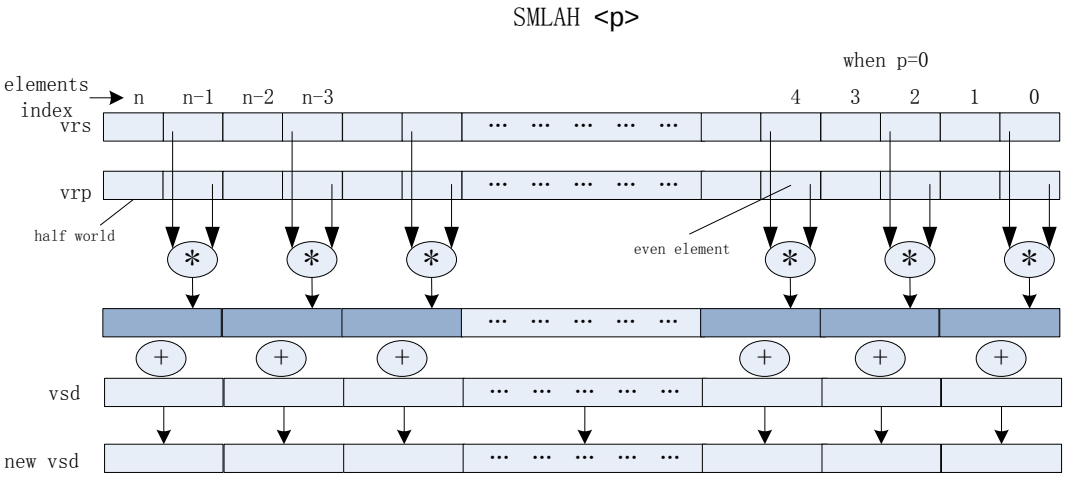
Each even(<p>=0) or odd(<p>=1) signed integer element in the vrp is multiplied by the corresponding signed one in the vrs, the multiplication result is added to the corresponding twice-size element in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*16)
    VSR[vsd][32,i]    = VSR[vsd][32,i] + signed(op1[16,2*i+p]) * signed(op2[16,2*i+p])
```

Exceptions:

RI, CpU



3.7.21 SMLSH<p>

Sign Extend Multiply-Subtract from Accumulator **COP2**

Bit	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
	010010	10011	vrs	vrp	000	vsd	1111	p	1

Syntax:

SMLSHE	vsd, vrs, vrp	//p=0
SMLSHO	vsd, vrs, vrp	//p=1

Description：

Each even(<p>=0) or odd(<p>=1) signed integer element in the vrp is multiplied by the corresponding signed one in the vrs, the multiplication result is added to the corresponding twice-size element in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*16)
    VSR[vsd][32,i]      = VSR[vsd][32,i] - signed(op1[16,2*i+p]) * signed(op2[16,2*i+p])
```

Exceptions:

RI, CpU

3.7.22 WSMLAH<p>

Sign Widen Multiply-Add to Accumulator **COP2**

Bit	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
	010010	11011	vrs	vrp	000	vsd	1110	p	1

Syntax:

WSMLAHL	vsd, vrs, vrp	//p=0
WSMLAHH	vsd, vrs, vrp	//p=1

Description：

The low(<p>=0) or high(<p>=1) half part of integer elements in the vrp are signed extended to twice of their original size, then multiplied by the corresponding twice-size integer elements in the vrs. Each multiplication result is added to the corresponding twice-size integer element in the vsd. Note that each most significant half of the multiplication result is discarded.

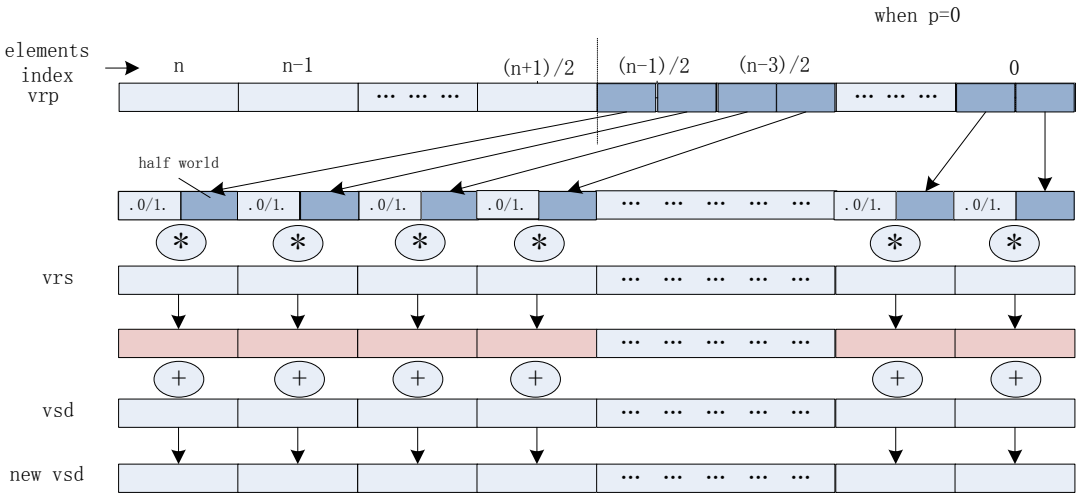
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [16-1:0] tmp
bit sign
N=MLEN/(2*16)
for i in N
    tmp             = op2[16,i+p*N]
    sign            = tmp[16-1]
    VSR[vsd][32,i]      = VSR[vsd][32,i] + op1[2*16,i] * {16{sign}, tmp}
```

Exceptions:

RI, CpU

WSMLAH 〈p〉



3.7.23 WSMLSH<p>

Sign Widen Multiply-Subtract from Accumulator **COP2**

Bit	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
	010010	11011	vrs	vrp	000	vsd	1111	p	1

Syntax:

```
WSMLSHL     vsd, vrs, vrp     //p=0
WSMLSHH     vsd, vrs, vrp     //p=1
```

Description：

The low(<p>=0) or high(<p>=1) half part of integer elements in the vrp are signed extended to twice of their original size, then multiplied by the corresponding twice-size signed integer elements in the vrs. Each multiplication result is subtracted from the corresponding twice-size integer element in the vsd. Note that each most significant half of the multiplication result is discarded.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [16-1:0] tmp
bit sign
N=MLEN/(2*16)
for i in N
    tmp             = op2[16,i+p*N]
    sign            = tmp[16-1]
    VSR[vsd][32,i]      = VSR[vsd][32,i] - op1[2*16,i] * {16{sign}, tmp}
```

Exceptions:

RI, CpU

3.7.24 SR<Lseg>MAC<fmt>

Segmented Replicate MAC Unsigned-Unsigned **COP2**

Bit	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
	010010									111		le2_1		vrs				vrp				0		fmt		vsd		le0		m_n		

Syntax:

SR1MAC2BI	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=0, fmt=2'B00, m_n=nnnnn:	Lseg=1, m=0
		// n=0~31(SIMD1024), 0~15(SIMD512), 0~7(SIMD256)	
SR2MAC2BI	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=1, fmt=2'B00, m_n=mnnnn:	Lseg=2, m=0~1
		// n=0~15(SIMD1024), 0~7(SIMD512), 0~3(SIMD256)	
SR4MAC2BI	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=0, fmt=2'B00, m_n=mmnnn:	Lseg=4, m=0~3
		// n=0~7(SIMD1024), 0~3(SIMD512), 0~1(SIMD256)	
SR8MAC2BI	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=1, fmt=2'B00, m_n=mmmnn:	Lseg=8, m=0~7
		// n=0~3(SIMD1024), 0~1(SIMD512),.0(SIMD256)	
SR16MAC2BI	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=0, fmt=2'B00, m_n=mmmmn:	Lseg=16, m=0~15
		// n=0~1(SIMD1024),.0(SIMD512).	
SR32MAC2BI	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=1, fmt=2'B00, m_n=mmmmm:	Lseg=32, m=0~31
		//n=0(SIMD1024)	
SR1MAC4BI	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=0, fmt=2'B01, m_n=nnnnn:	Lseg=1, m=0
		// n=0~31(SIMD1024), 0~15(SIMD512), 0~7(SIMD256)	
SR2MAC4BI	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=1, fmt=2'B01, m_n=mnnnn:	Lseg=2, m=0~1
		// n=0~15(SIMD1024), 0~7(SIMD512), 0~3(SIMD256)	
SR4MAC4BI	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=0, fmt=2'B01, m_n=mmnnn:	Lseg=4, m=0~3
		// n=0~7(SIMD1024), 0~3(SIMD512), 0~1(SIMD256)	
SR8MAC4BI	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=1, fmt=2'B01, m_n=mmmnn:	Lseg=8, m=0~7
		// n=0~3(SIMD1024), 0~1(SIMD512),.0(SIMD256)	
SR16MAC4BI	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=0, fmt=2'B01, m_n=mmmmn:	Lseg=16, m=0~15
		// n=0~1(SIMD1024),.0(SIMD512).	
SR32MAC4BI	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=1, fmt=2'B01, m_n=mmmmm:	Lseg=32, m=0~31
		//n=0(SIMD1024)	
SR1MACUUB	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=0, fmt=2'B10, m_n=nnnnn:	Lseg=1, m=0
		// n=0~31(SIMD1024), 0~15(SIMD512), 0~7(SIMD256)	
SR2MACUUB	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=1, fmt=2'B10, m_n=mnnnn:	Lseg=2, m=0~1
		// n=0~15(SIMD1024), 0~7(SIMD512), 0~3(SIMD256)	
SR4MACUUB	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=0, fmt=2'B10, m_n=mmnnn:	Lseg=4, m=0~3
		// n=0~7(SIMD1024), 0~3(SIMD512), 0~1(SIMD256)	
SR8MACUUB	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=1, fmt=2'B10, m_n=mmmnn:	Lseg=8, m=0~7
		// n=0~3(SIMD1024), 0~1(SIMD512),.0(SIMD256)	
SR16MACUUB	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=0, fmt=2'B10, m_n=mmmmn:	Lseg=16, m=0~15
		// n=0~1(SIMD1024),.0(SIMD512).	
SR32MACUUB	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=1, fmt=2'B10, m_n=mmmmm:	Lseg=32, m=0~31

//n=0(SIMD1024)

Description：

First, three vector registers: vrs, vrp and vsd need be divided equally into N segments, the segment here has specific length of <Lseg> words. In addition, the <n>th segment of original vrp need be replicated to form a new vrp. Then each unsigned integer element in the vrs is multiplied by the corresponding unsigned one from the new vrp. For each segment, all the included multiplication results are added together, then the sum result is added to the <m>th integer word of the corresponding segment in the vsd.

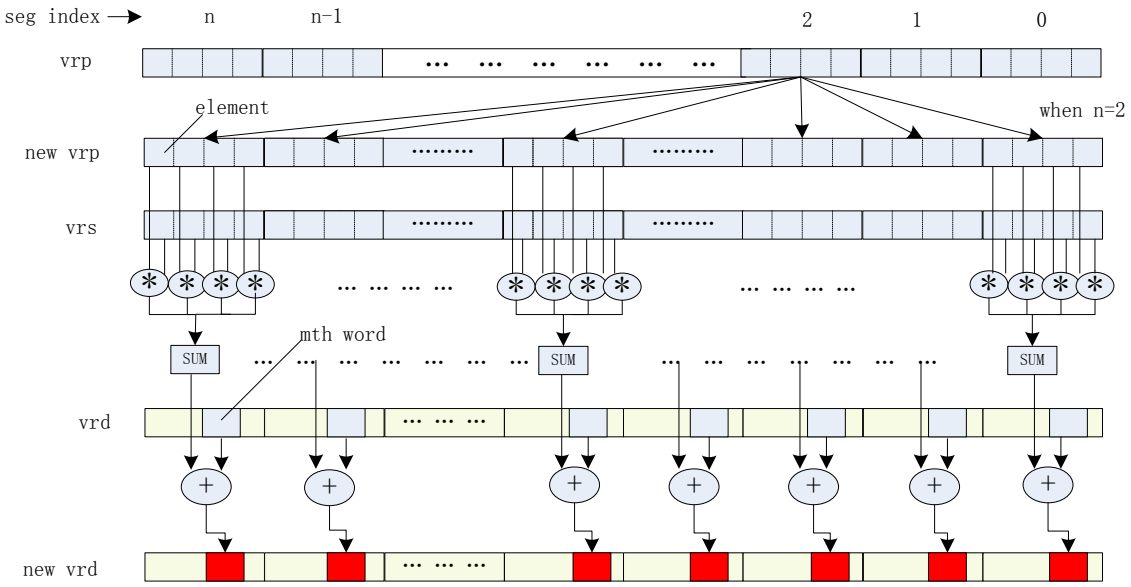
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [Lseg*32-1:0] op=op2[Lseg*32*(n+1)-1:Lseg*32*n]
bit [31:0] sum
M= (Lseg*32)/esize
N= MLEN/(Lseg*32)
for j in N
    sum = 0
    for i in M
        sum = sum + unsigned(op1[esize,i+j*M]) * unsigned(op[esize,i])
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

S<Lseg>MAC<fmt>



3.7.25 SR<Lseg>MACSU<fmt>

Segmented Replicate MAC Signed-Unsigned **COP2**

Bit	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
	010010	111	le2_1	vrs	vrp	1	fmt	vsd	le0	m_n

Syntax:

SR1MACSUB	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=0, fmt=2'B10, m_n=nnnnn:	Lseg=1, m=0
		// n=0~31(SIMD1024), 0~15(SIMD512), 0~7(SIMD256)	
SR2MACSUB	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=1, fmt=2'B10, m_n=mnnnn:	Lseg=2, m=0~1
		// n=0~15(SIMD1024), 0~7(SIMD512), 0~3(SIMD256)	
SR4MACSUB	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=0, fmt=2'B10, m_n=mmnnn:	Lseg=4, m=0~3
		// n=0~7(SIMD1024), 0~3(SIMD512), 0~1(SIMD256)	
SR8MACSUB	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=1, fmt=2'B10, m_n=mmmnn:	Lseg=8, m=0~7
		// n=0~3(SIMD1024), 0~1(SIMD512),.0(SIMD256)	
SR16MACSUB	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=0, fmt=2'B10, m_n=mmmmn:	Lseg=16, m=0~15
		// n=0~1(SIMD1024),.0(SIMD512).	
SR32MACSUB	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=1, fmt=2'B10, m_n=mmmmm:	Lseg=32, m=0~31
		//n=0(SIMD1024)	

Description：

First, three vector registers: vrs, vrp and vsd need be divided equally into N segments, the segment here has specific length of <Lseg> words. In addition, the <n>th segment of original vrp need be replicated to form a new vrp. Then each signed integer element in the vrs is multiplied by the corresponding unsigned one from the new vrp. For each segment, all the included multiplication results are added together, then the sum result is added to the <m>th integer word of the corresponding segment in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [Lseg*32-1:0] op=op2[Lseg*32*(n+1)-1:Lseg*32*n]
bit [31:0] sum
M = (Lseg*32)/esize
N = MLEN/(Lseg*32)
for j in N
    sum = 0
    for i in M
        sum = sum + signed(op1[esize,i+j*M]) * unsigned(op[esize,i])
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

3.7.26 SR<Lseg>MACS<fmt>

Segmented Replicate MAC Signed-Signed **COP2**

Bit	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
	010010	111	le2_1	vrs	vrp	1	fmt	vsd	le0	m_n

Syntax:

SR1MACSSB	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=0, fmt=2'B00, m_n=nnnnn: // n=0~31(SIMD1024), 0~15(SIMD512), 0~7(SIMD256)	Lseg=1, m=0
SR2MACSSB	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=1, fmt=2'B00, m_n=mnnnn: // n=0~15(SIMD1024), 0~7(SIMD512), 0~3(SIMD256)	Lseg=2, m=0~1
SR4MACSSB	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=0, fmt=2'B00, m_n=mmnnn: // n=0~7(SIMD1024), 0~3(SIMD512), 0~1(SIMD256)	Lseg=4, m=0~3
SR8MACSSB	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=1, fmt=2'B00, m_n=mmmnn: // n=0~3(SIMD1024), 0~1(SIMD512),.0(SIMD256)	Lseg=8, m=0~7
SR16MACSSB	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=0, fmt=2'B00, m_n=mmmmn: // n=0~1(SIMD1024),.0(SIMD512).	Lseg=16, m=0~15
SR32MACSSB	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=1, fmt=2'B00, m_n=mmmmm: //n=0(SIMD1024)	Lseg=32, m=0~31
SR1MACSSH	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=0, fmt=2'B01, m_n=nnnnn: // n=0~31(SIMD1024), 0~15(SIMD512), 0~7(SIMD256)	Lseg=1, m=0
SR2MACSSH	vsd[m], vrs, vrp[n]	//le2_1=2'B00, le0=1, fmt=2'B01, m_n=mnnnn: // n=0~15(SIMD1024), 0~7(SIMD512), 0~3(SIMD256)	Lseg=2, m=0~1
SR4MACSSH	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=0, fmt=2'B01, m_n=mmnnn: // n=0~7(SIMD1024), 0~3(SIMD512), 0~1(SIMD256)	Lseg=4, m=0~3
SR8MACSSH	vsd[m], vrs, vrp[n]	//le2_1=2'B01, le0=1, fmt=2'B01, m_n=mmmnn: // n=0~3(SIMD1024), 0~1(SIMD512),.0(SIMD256)	Lseg=8, m=0~7
SR16MACSSH	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=0, fmt=2'B01, m_n=mmmmn: // n=0~1(SIMD1024),.0(SIMD512).	Lseg=16, m=0~15
SR32MACSSH	vsd[m], vrs, vrp[n]	//le2_1=2'B10, le0=1, fmt=2'B01, m_n=mmmmm: //n=0(SIMD1024)	Lseg=32, m=0~31

Description：

First, three vector registers: vrs, vrp and vsd need be divided equally into N segments, the segment here has specific length of <Lseg> words. In addition, the <n>th segment of original vrp need be replicated to form a new vrp. Then each signed integer element in the vrs is multiplied by the corresponding signed one from the new vrp. For each segment, all the included multiplication results are added together, then the sum result is added to the <m>th integer word of the corresponding segment in the vsd.

Operation：

check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]

bit [Lseg*32-1:0] op=op2[Lseg*32*(n+1)-1:Lseg*32*n]
bit [31:0] sum
M = (Lseg*32)/esize
N = MLEN/(Lseg*32)
for j in N
    sum = 0
    for i in M
        sum = sum + signed(op1[esize,i+j*M]) * signed(op[esize,i])
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum

Exceptions:

RI, CpU

3.7.27 S<Lseg>MACUUB

Segmented MAC Unsigned-Unsigned Byte **COP2**

Bit	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
	010010	111	00	vrs	vrp	1	11	vsd	m_le

Syntax:

S1MACUUB	vsd[m], vrs, vrp	//m_le=100000:	Lseg=1, m=0
S2MACUUB	vsd[m], vrs, vrp	//m_le=m10000:	Lseg=2, m=0~1
S4MACUUB	vsd[m], vrs, vrp	//m_le=mm1000:	Lseg=4, m=0~3
S8MACUUB	vsd[m], vrs, vrp	//m_le=mmm100:	Lseg=8, m=0~7
S16MACUUB	vsd[m], vrs, vrp	//m_le=mmmm10:	Lseg=16, m=0~15
S32MACUUB	vsd[m], vrs, vrp	//m_le=mmmmm1:	Lseg=32, m=0~31, valid for SIMD1024

Description：

First, three vector registers: vrs, vrp and vsd need be divided equally into N segments, the segment here has specific length of <Lseg> words. Then each unsigned integer byte element in the vrs is multiplied by the corresponding unsigned one in the vrp. For each segment, all the included multiplication results are added together, then the sum result is added to the <m>th integer word of the corresponding segment in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [31:0] sum
M = (Lseg*32)/8
N = MLEN/(Lseg*32)
for j in N
    sum = 0
    for i in M
        sum = sum + unsigned(op1[8,i+j*M]) * unsigned(op2[8,i+j*M])
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

3.7.28 S<Lseg>MACSUB

Segmented MAC Signed-Unsigned Byte **COP2**

Bit	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
	010010	111	01	vrs	vrp	1	11	vsd	m_le

Syntax:

S1MACSUB	vsd[m], vrs, vrp	//m_le=100000:	Lseg=1, m=0
S2MACSUB	vsd[m], vrs, vrp	//m_le=m10000:	Lseg=2, m=0~1
S4MACSUB	vsd[m], vrs, vrp	//m_le=mm1000:	Lseg=4, m=0~3
S8MACSUB	vsd[m], vrs, vrp	//m_le=mmm100:	Lseg=8, m=0~7
S16MACSUB	vsd[m], vrs, vrp	//m_le=mmmm10:	Lseg=16, m=0~15
S32MACSUB	vsd[m], vrs, vrp	//m_le=mmmmm1:	Lseg=32, m=0~31, valid for SIMD1024

Description：

First, three vector registers: vrs, vrp and vsd need be divided equally into N segments, the segment here has specific length of <Lseg> words. Then each **signed** integer byte element in the vrs is multiplied by the corresponding **unsigned** one in the vrp. For each segment, all the included multiplication results are added together, then the sum result is added to the <m>th integer word of the corresponding segment in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [31:0] sum
M = (Lseg*32)/8
N = MLEN/(Lseg*32)
for j in N
    sum = 0
    for i in M
        sum = sum + signed(op1[8,i+j*M]) * unsigned(op2[8,i+j*M])
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

3.7.29 S<Lseg>MACSS<fmt>

Segmented MAC Signed-Signed **COP2**

Bit	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
	010010						111		fmt		vrs					vrp					1	11		vsd		m_le							

Syntax:

S1MACSSB	vsd[m], vrs, vrp	//fmt=2'B10, m_le=100000:	Lseg=1, m=0
S2MACSSB	vsd[m], vrs, vrp	//fmt=2'B10, m_le=m10000:	Lseg=2, m=0~1
S4MACSSB	vsd[m], vrs, vrp	//fmt=2'B10, m_le=mm1000:	Lseg=4, m=0~3
S8MACSSB	vsd[m], vrs, vrp	//fmt=2'B10, m_le=mmm100:	Lseg=8, m=0~7
S16MACSSB	vsd[m], vrs, vrp	//fmt=2'B10, m_le=mmmm10:	Lseg=16, m=0~15
S32MACSSB	vsd[m], vrs, vrp	//fmt=2'B10, m_le=mmmmm1:	Lseg=32, m=0~31, valid for SIMD1024
S1MACSSH	vsd[m], vrs, vrp	//fmt=2'B11, m_le=100000:	Lseg=1, m=0
S2MACSSH	vsd[m], vrs, vrp	//fmt=2'B11, m_le=m10000:	Lseg=2, m=0~1
S4MACSSH	vsd[m], vrs, vrp	//fmt=2'B11, m_le=mm1000:	Lseg=4, m=0~3
S8MACSSH	vsd[m], vrs, vrp	//fmt=2'B11, m_le=mmm100:	Lseg=8, m=0~7
S16MACSSH	vsd[m], vrs, vrp	//fmt=2'B11, m_le=mmmm10:	Lseg=16, m=0~15
S32MACSSH	vsd[m], vrs, vrp	//fmt=2'B11, m_le=mmmmm1:	Lseg=32, m=0~31, valid for SIMD1024

Description：

First, three vector registers: vrs, vrp and vsd need be divided equally into N segments, the segment here has specific length of <Lseg> words. Then each signed integer element in the vrs is multiplied by the corresponding signed one in the vrp. For each segment, all the included multiplication results are added together, then the sum result is added to the <m>th integer word of the corresponding segment in the vsd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [31:0] sum
M = (Lseg*32)/esize
N = MLEN/(Lseg*32)
for j in N
    sum = 0
    for i in M
        sum = sum + signed(op1[esize,i+j*M]) * signed(op2[esize,i+j*M])
    VSR[vsd][32,m+j*Lseg] = VSR[vsd][32,m+j*Lseg] + sum
```

Exceptions:

RI, CpU

3.7.30 MAXA<fmt>

Maximum of Absolute Values **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	0110	fmt

Syntax:

MAXAB	vrd, vrs, vrp	//fmt=2'B00
MAXAH	vrd, vrs, vrp	//fmt=2'B01
MAXAW	vrd, vrs, vrp	//fmt=2'B10

Description：

Compare corresponding elements' absolute value in the vrs and the vrp. For each comparison, choose the element with a larger absolute value to update the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit cond = 0
for i in MLEN/esize
    cond            = abs(op1[esize,i])>abs(op2[esize,i])
    VPR[vrd][esize,i]  = cond? op1[esize,i] : op2[esize,i]
```

Exceptions:

RI, CpU

3.7.31 MAXS<fmt>

Signed Maximum **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	0111	fmt

Syntax:

MAXSB	vrd, vrs, vrp	//fmt=2'B00
MAXSH	vrd, vrs, vrp	//fmt=2'B01
MAXSW	vrd, vrs, vrp	//fmt=2'B10

Description：

Compare corresponding elements' signed value in the vrs and the vrp. For each comparison, choose the element with a larger signed value to update the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit cond=0
for i in MLEN/esize
    cond            = signed(op1[esize,i])>signed(op2[esize,i])
    VPR[vrd][esize,i]  = cond? op1[esize,i] : op2[esize,i]
```

Exceptions:

RI, CpU

3.7.32 MAXU<fmt>

Unsigned Maximum **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	001	fmt

Syntax:

MAXUB	vrd, vrs, vrp	//fmt=2'B000
MAXUH	vrd, vrs, vrp	//fmt=2'B001
MAXUW	vrd, vrs, vrp	//fmt=2'B010
MAXU2BI	vrd, vrs, vrp	//fmt=2'B100
MAXU4BI	vrd, vrs, vrp	//fmt=2'B101

Description：

Compare corresponding elements' unsigned value in the vrs and the vrp. For each comparison, choose the element with a larger unsigned value to update the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit cond=0
for i in MLEN/esize
    cond            = unsigned(op1[esize,i])>unsigned(op2[esize,i])
    VPR[vrd][esize,i]  = cond? op1[esize,i] : op2[esize,i]
```

Exceptions:

RI, CpU

3.7.33 MINA<fmt>

Minimum of Absolute Vaules **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	0100	fmt

Syntax:

MINAB	vrd, vrs, vrp	//fmt=2'B00
MINAH	vrd, vrs, vrp	//fmt=2'B01
MINAW	vrd, vrs, vrp	//fmt=2'B10

Description：

Compare corresponding elements' absolute value in the vrs and the vrp. For each comparison, choose the element with a smaller absolute value to update the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit cond = 0
for i in MLEN/esize
    cond            = abs(op1[esize,i])<abs(op2[esize,i])
    VPR[vrd][esize,i] = cond? op1[esize,i] : op2[esize,i]
```

Exceptions:

RI, CpU

3.7.34 MINS<fmt>

Signed Minimum **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	0101	fmt

Syntax:

MINSB	vrd, vrs, vrp	//fmt=2'B00
MINSH	vrd, vrs, vrp	//fmt=2'B01
MINSW	vrd, vrs, vrp	//fmt=2'B10

Description：

Compare corresponding elements' signed value in the vrs and the vrp. For each comparison, choose the element with a smaller signed value to update the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit cond=0
for i in MLEN/esize
    cond            = signed(op1[esize,i])<signed(op2[esize,i])
    VPR[vrd][esize,i]  = cond? op1[esize,i] : op2[esize,i]
```

Exceptions:

RI, CpU

3.7.35 MINU<fmt>

Unsigned Minimum **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	000	fmt

Syntax:

MINUB	vrd, vrs, vrp	//fmt=3'B000
MINUH	vrd, vrs, vrp	//fmt=3'B001
MINUW	vrd, vrs, vrp	//fmt=3'B010
MINU2BI	vrd, vrs, vrp	//fmt=3'B100
MINU4BI	vrd, vrs, vrp	//fmt=3'B101

Description：

Compare corresponding elements' unsigned value in the vrs and the vrp. For each comparison, choose the element with a smaller unsigned value to update the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit cond=0
for i in MLEN/esize
    cond            = unsigned(op1[esize,i])<unsigned(op2[esize,i])
    VPR[vrd][esize,i]  = cond? op1[esize,i] : op2[esize,i]
```

Exceptions:

RI, CpU

3.7.36 SATSS<fmt><tgt>

Saturate from Signed to Signed **SPECIAL2**

Bit	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
	011100	0	1	tgt	fmt	vrs	00000	vrd	101111

Syntax:

```
SATSSHB     vrd, vrs;       //tgt=0, fmt=2'B01
SATSSWB     vrd, vrs;       //tgt=0, fmt=2'B10
SATSSWH     vrd, vrs;       //tgt=1, fmt=2'B10
```

Description：

Each signed element of esize in the vrs is saturated to signed value of esize_tgt without changing the data width, the saturated result updates the corresponding element in the vrd.

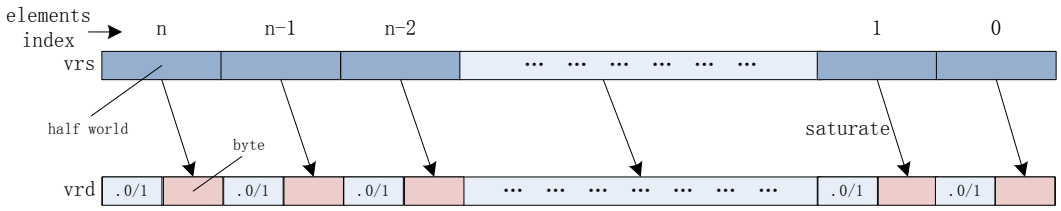
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [esize-1:0] minv={(esize-esize_tgt+1){1},(esize_tgt-1){0}}
bit [esize-1:0] maxv={(esize-esize_tgt+1){0},(esize_tgt-1){1}}
bit [esize-1:0] v
for i in MLEN/esize
    v                   = signed(op1[esize,i])<signed(minv) ? minv : op1[esize,i]
    VPR[vrd][esize,i]   = signed(v)>signed(maxv) ? maxv : v
```

Exceptions:

RI, CpU

SATSS<fmt><tgt>



3.7.37 SATSU<fmt><tgt>

Saturate from Signed to Usigned **SPECIAL2**

Bit	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
	011100	1	tgt	fmt	vrs	00000	vrd	101111

Syntax:

```
SATSUB2BI       vrd, vrs;       //tgt=00, fmt=2'B00
SATSUB4BI       vrd, vrs;       //tgt=01, fmt=2'B00
SATSUH2BI       vrd, vrs;       //tgt=00, fmt=2'B01
SATSUH4BI       vrd, vrs;       //tgt=01, fmt=2'B01
SATSUHB         vrd, vrs;       //tgt=10, fmt=2'B01
SATSUW2BI       vrd, vrs;       //tgt=00, fmt=2'B10
SATSUW4BI       vrd, vrs;       //tgt=01, fmt=2'B10
SATSUWB         vrd, vrs;       //tgt=10, fmt=2'B10
SATSUWH         vrd, vrs;       //tgt=11, fmt=2'B10
```

Description：

Each signed element of esize in the vrs is saturated to unsigned value of esize_tgt without changing the data width, the saturated result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [esize-1:0] minv={esize{0}}
bit [esize-1:0] maxv={(esize-esize_tgt){0},(esize_tgt){1}}
bit [esize-1:0] v
for i in MLEN/esize
    v                   = signed(op1[esize,i])< 0? minv : op1[esize,i]
    VPR[vrd][esize,i]   = signed(v)>signed(maxv) ? maxv : v
```

Exceptions:

RI, CpU

3.7.38 SATUU<fmt><tgt>

Saturate from Unsigned to Unsigned **SPECIAL2**

Bit	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
	011100	00	fmt	vrs	00000	vrd	101111

Syntax:

SATUUB2BI	vrd, vrs;	//fmt=3'B000
SATUUB4BI	vrd, vrs;	//fmt=3'B001
SATUUH2BI	vrd, vrs;	//fmt=3'B010
SATUUH4BI	vrd, vrs;	//fmt=3'B011
SATUUHB	vrd, vrs;	//fmt=3'B100
SATUUW4BI	vrd, vrs;	//fmt=3'B101
SATUUWB	vrd, vrs;	//fmt=3'B110
SATUUWH	vrd, vrs;	//fmt=3'B111

Description：

Each unsigned element of esize in the vrs is saturated to unsigned value of esize_tgt without changing the data width, the saturated result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [esize-1:0] maxv={(esize-esize_tgt){0},(esize_tgt){1}}
for i in MLEN/esize
    VPR[vrd][esize,i]   = unsigned(op1[esize,i])>unsigned(maxv) ? maxv : op1[esize,i]
```

Exceptions:

RI, CpU

3.7.39 TOC<fmt>

Tail One bits Count **COP2**

Bit	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
	010010	10011	vrs	00000	000	vsd	0110	fmt

Syntax:

TOCB	vsd, vrs	//fmt=2'B00
TOCH	vsd, vrs	//fmt=2'B01
TOCW	vsd, vrs	//fmt=2'B10

Description：

Count the number of consecutive elements whose bit0 are value one in the vrs, the counting order is from element 0 to element (MLEN/esize) - 1. The count result is written to word0 in the vsd.

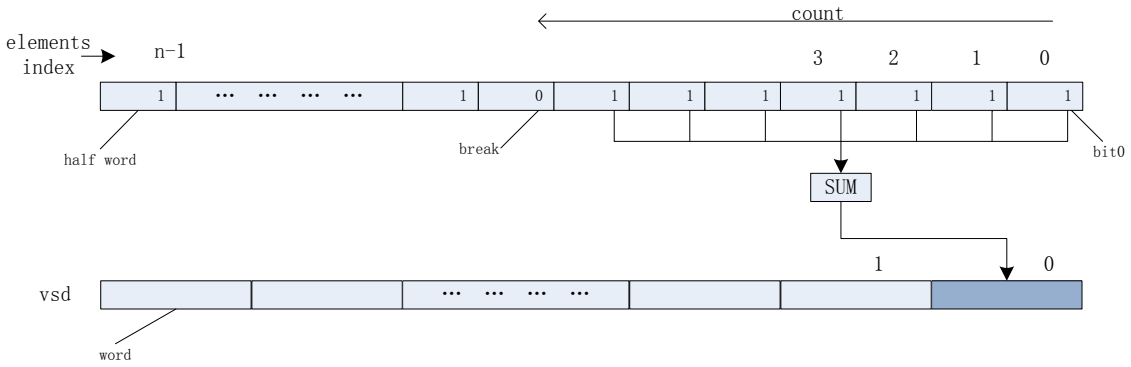
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [esize-1:0] element
bit [31:0] count=0
for (i=0; i<MLEN/esize; i++) {
      element= op1[esize,i];
      if (element[0] != 1) break;
      count++;
}
VSR[vsd][32,0]      = count
```

Exceptions:

RI, CpU

TOC<fmt>



3.8 Bitwise

3.8.1 ANDV

Logical AND **COP2**

Bit	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
	010010						10011					vrs					vrp					vrd					000010					

Syntax:

ANDV vrd, vrs, vrp

Description：

Make a bitwise AND operation between two operands: MLEN bits from the vrs and MLEN bits from the vrp, the result updates the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

VPR[vrd] = op1 & op2

Exceptions:

RI, CpU

3.8.2 ANDNV

Logical AND Negative **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	000011

Syntax:

ANDNV vrd, vrs, vrp

Description：

Make a bitwise AND operation between two operands: MLEN bits from the vrp and MLEN bits being bitwise-negative from the original vrs, the result updates the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

VPR[vrd] = (~op1) & op2

Exceptions:

RI, CpU

3.8.3 ANDIB

Immediate Logical AND **COP2**

Bit	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
	010010	11011	vrs	imm[4:0]	vrd	001	imm[7:5]

Syntax:

ANDIB vrd, vrs, imm

Description：

Take each byte element in the vrs to make a bitwise AND operation with the common 8-bit immediate value from <imm>, the result updates the corresponding byte element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [7:0] imm=imm[7:0]
for i in MLEN/8
    VPR[vrd][8,i]= op1[8,i] & imm
```

Exceptions:

RI, CpU

3.8.4 ORV

Logical OR **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	000100

Syntax:

ORV vrd, vrs, vrp

Description：

Make a bitwise OR operation between two operands: MLEN bits from the vrs and MLEN bits from the vrp, the result updates the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

VPR[vrd] = (op1 | op2)

Exceptions:

RI, CpU

3.8.5 ORNV

Logical OR Negtive **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	000101

Syntax:

ORNV vrd, vrs, vrp

Description：

Make a bitwise OR operation between two operands: MLEN bits from the vrp and MLEN bits being bitwise-negative from the original vrs, the result updates the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

VPR[vrd] = (~op1) | op2

Exceptions:

RI, CpU

3.8.6 ORIB

Immediate Logical OR **COP2**

Bit	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
	010010	11011	vrs	imm[4:0]	vrd	010	imm[7:5]

Syntax:

ORIB vrd, vrs, imm

Description：

Take each byte element in the vrs to make a bitwise OR operation with the common 8-bit immediate value from <imm>, the result updates the corresponding byte element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [7:0] imm=imm[7:0]
for i in MLEN/8
    VPR[vrd][8,i]= (op1[8,i] | imm)
```

Exceptions:

RI, CpU

3.8.7 XORV

Logical XOR **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	000110

Syntax:

XORV vrd, vrs, vrp

Description：

Make a bitwise XOR operation between two operands: MLEN bits from the vrs and MLEN bits from the vrp, the result updates the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

VPR[vrd] = (op1 ^ op2)

Exceptions:

RI, CpU

3.8.8 XORNV

Logical XOR Negtive **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	000111

Syntax:

XORNV vrd, vrs, vrp

Description：

Make a bitwise XOR operation between two operands: MLEN bits from the vrp and MLEN bits being bitwise-negative from the original vrs, the result updates the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

VPR[vrd] = (~op1) ^ op2

Exceptions:

RI, CpU

3.8.9 XORIB

Immediate Logical XOR **COP2**

Bit	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
	010010	11011	vrs	imm[4:0]	vrd	011	imm[7:5]

Syntax:

XORIB vrd, vrs, imm

Description：

Take each byte element in the vrs to make a bitwise XOR operation with the common 8-bit immediate value from <imm>, the result updates the corresponding byte element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [7:0] imm=imm[7:0]
for i in MLEN/8
    VPR[vrd][8,i]= (op1[8,i] ^ imm)
```

Exceptions:

RI, CpU

3.8.10 BSELV

Bit Select **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	001110

Syntax:

BSELV vrd, vrs, vrp

Description：

Take each bit in the vrd, if the bit value is zero then choose the corresponding bit in the vrs, otherwise choose the corresponding one in the vrp. Finally, concatenate all chosen bits together to update the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [MLEN-1:0] op3=VPR[vrd]
for i in MLEN
    VPR[vrd][i]= op3[i] ? op2[i] : op1[i]
```

Exceptions:

RI, CpU

3.9 Floating Point Arithmetic

3.9.1 FADDW

Floating point Addition **COP2**

Bit	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
	010010						10100					vrs					vrp					vrd					0000				11	

Syntax:

FADDW vrd, vrs, vrp

Description：

Add corresponding single-precision floating point elements in the vrs and the vrp. Each add result updates the corresponding element in the vrd.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

bit [MLEN-1:0] op2=VPR[vrp]

for i in MLEN/esize

VPR[vrd][esize,i]= fpadd(op1[esize,i] , op2[esize,i])

Exceptions:

RI, CpU, MFPE

3.9.2 FSUBW

Floating point Subtraction **COP2**

Bit	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
	010010	10100	vrs	vrp	vrd	0010	11

Syntax:

FSUBW vrd, vrs, vrp

Description：

Subtract each single-precision floating point element in the vrp from the corresponding one in the vrs, the subtraction result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpsub(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.9.3 FMULW

Floating point Multiplication **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	1000	11

Syntax:

FMULW vrd, vrs, vrp

Description：

Multiply corresponding single-precision floating point elements in the vrs and the vrp. Each multiplication result updates the corresponding element in the vrd.

The multiplication operation is defined by the IEEE-754-2008.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpmul(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.9.4 FCMULRW

Floating point Complex Multiply for Real **COP2**

Bit	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
	010010	11011	vrs	vrp	vrd	110000

Syntax:

FCMULRW vrd, vrs, vrp

Description：

Each single-precision floating point element-pair in the vrs represents a complex number being composed of odd-imaginary part and even-real part, and the same composition fits for vrp.

For each complex number in the vrs, multiply its odd-imaginary part and even-real part by the corresponding even-real part in the vrp, respectively. The multiplication result-pair updates the corresponding element-pair in the vrd.

The multiply operation is defined by the IEEE-754-2008.The multiplication between an infinity and a zero signals Invalid Operation exception. If the Invalid Operation exception is disabled, the result is the default quiet NaN.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*esize)
    VPR[vrd][esize,2*i]= fpmul(op1[esize,2*i], op2[esize,2*i])
    VPR[vrd][esize,2*i+1]= fpmul(op1[esize,2*i+1], op2[esize,2*i])
```

Exceptions:

RI, CpU, MFPE

3.9.5 FCMULIW

Floating point Complex Multiply for Imaginary **COP2**

Bit	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
	010010						11011				vrs					vrp					vrd					110010							

Syntax:

FCMULIW vrd, vrs, vrp

Description：

Each single-precision floating point element-pair in the vrs represents a complex number being composed of odd-imaginary part and even-real part, and the same composition fits for vrp.

For each complex number in the vrs, multiply its odd-imaginary part and even-real part by the corresponding odd-imaginary part in the vrp, respectively. Note that the product generated by two odd-imaginary parts need do further negative operation. Finally, the result-pair updates the corresponding element-pair in the vrd.

The multiply operation is defined by the IEEE-754-2008.The multiplication between an infinity and a zero signals Invalid Operation exception. If the Invalid Operation exception is disabled, the result is the default quiet NaN.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*esize)
    VPR[vrd][esize,2*i]= fpmul(op1[esize,2*i], op2[esize,2*i+1])
    VPR[vrd][esize,2*i+1]= -(fpmul(op1[esize,2*i+1], op2[esize,2*i+1]))
```

Exceptions:

RI, CpU, MFPE

3.9.6 FCADDW

Floating point Complex Cross Add **COP2**

Bit	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
	010010	10011	vrs	vrp	vrd	111010

Syntax:

FCADDW vrd, vrs, vrp

Description:

Each single-precision floating point element-pair in the vrp represents a complex number being composed of odd-imaginary part and even-real part. However, each single-precision floating point element-pair in the vrs also represents a complex number with reverse order being odd-real part and even-imaginary part.

For each element-pair in the vrs, add the even-imaginary part of vrs with the corresponding odd-imaginary part of vrp, and parallel add the odd-real part of vrs with the even-real part of vrp, respectively. The result-pair updates the corresponding element-pair in the vrd.

Please note that complex-multiply can be implemented by using FCMULRW, FCMULIW and FCADDW together. For example, complex-multiply(vr1, vr2) can be expressed as:

fcmulrw vr0, vr1, vr2

fcmuliw vr3, vr1, vr2

fcaddw vr4, vr3, vr0

The vr4 is the result of complex-multiply.

Operation:

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*esize)
    VPR[vrd][esize,2*i]= fpadd(op1[esize,2*i], op2[esize,2*i+1])
    VPR[vrd][esize,2*i+1]= fpadd(op1[esize,2*i+1], op2[esize,2*i])
```

Exceptions:

RI, CpU, MFPE

3.9.7 FXAS<Lseg>W

Floating point Segmented Cross Add Subtract **COP2**

Bit	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
	010010	10011	00	le	vrp	vrd	111110

Syntax:

FXAS1W	vrd, vrp	//le=3'B000 Lseg=1
FXAS2W	vrd, vrp	//le=3'B001 Lseg=2
FXAS4W	vrd, vrp	//le=3'B010 Lseg=4
FXAS8W	vrd, vrp	//le=3'B011 Lseg=8
FXAS16W	vrd, vrp	//le=3'B100 Lseg=16, Valid for SIMD1024

Description：

First, two vector registers vrp and vrd need be divided equally into N (N must be an even number) segments, the segment here has specific length of <Lseg> words. Then every two segments form a segment-pair.

For each segment-pair of vrp, every single-precision floating point element in the first segment is added by the corresponding one in the second segment, and the former one is subtracted by the latter one in parallel. The addition result updates the corresponding word in the first segment of the segment-pair in the vrd, and the subtraction result updates the corresponding word in the second segment of the segment-pair in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
for n in MLEN/(Lseg*32*2)
   for i in Lseg
      VPR[vrd][32,2*Lseg*n+i] = fpadd(op1[32,2*Lseg*n+i], op1[32,2*Lseg*n+Lseg+i])
      VPR[vrd][32,2*Lseg*n+Lseg+i] = fpsub(op1[32,2*Lseg*n+i], op1[32,2*Lseg*n+Lseg+i])
```

Exceptions:

RI, CpU, MFPE

3.9.8 FMAXW

Floating point Maximum **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	0111	11

Syntax:

FMAXW vrd, vrs, vrp

Description：

Compare corresponding single-precision floating point elements' value in the vrs and the vrp. For each comparison, choose the element with larger value to update the corresponding one in the vrd.

The largest value is defined by the maxNum operation in the IEEE-754-2008.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpmax(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.9.9 FMAXAW

Maximum Based of Absolute Values Comparison **COP2**

Bit	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
	010010						10000					vrp					vrs					vrd					0110				11	

Syntax:

FMAXAW vrd, vrs, vrp

Description：

Compare corresponding single-precision floating point elements' absolute value in the vrs and the vrp. For each comparison, choose the element with larger absolute value to update the corresponding one in the vrd.

The largest absolute value is defined by the maxNumMag operation in the IEEE-754-2008.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpabs(op1[esize,i]) >fpabs(op2[esize,i])? op1[esize,i]:op2[esize,i];
```

Exceptions:

RI, CpU, MFPE

3.9.10 FMINW

Floating point Minimum **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	0101	11

Syntax:

FMINW vrd, vrs, vrp

Description：

Compare corresponding single-precision floating point elements' value in the vrs and the vrp. For each comparison, choose the element with smaller value to update the corresponding one in the vrd.

The smallest value is defined by the minNum operation in the IEEE-754-2008.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpmin(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.9.11 FMINAW

Minimum Based on Absolute Values Comparison **COP2**

Bit	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
	010010						10000					vrs					vrp					vrd					0100				11	

Syntax:

FMINAW vrd, vrs, vrp

Description：

Compare corresponding single-precision floating point elements' absolute value in the vrs and the vrp. For each comparison, choose the element with smaller absolute value to update the corresponding one in the vrd.

The smallest absolute value is defined by the minNumMag operation in the IEEE-754-2008.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpabs(op1[esize,i]) < fpabs(op2[esize,i])? op1[esize,i]:op2[esize,i];
```

Exceptions:

RI, CpU, MFPE

3.9.12 FCLASSW

Floating point Class Mask **COP2**

Bit	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
	010010	01000	vrs	00000	vrd	101110

Syntax:

FCLASSW vrd, vrs,

Description：

Take each single-precision floating point element in the vrs to generate a bit mask reflecting the floating point class which the taken element belonging to, the bit mask then is written to the corresponding element in the vrd.

The mask has 10 bits with different meaning: Bit 0 and 1 indicate NAN values: signaling NaN(bit0) and quiet NaN (bit1).Bit 2,3,4,5 classify negative values: infinity(bit2), normal(bit3), subnormal(bit4) and zero(bit5). Bit 6,7,8,9 classify positive values: infinity (bit6), normal(bit7), subnormal(bit8) and zero (bit9).

The input values and generated bit masks are not affected by the flush-to-zero bit FS in the MCSR register.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpclass(op1[esize,i])
```

Exceptions:

RI, CpU

3.10 Floating Point Compare

3.10.1 FCEQW

Floating point Quiet Compare Equal **COP2**

Bit	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
	010010						10000					vrs					vrp					vrd					1010				11	

Syntax:

FCEQW vrd, vrs, vrp

Description：

Take each single-precision floating point element in the vrs to compare with the corresponding one in the vrp, if they are ordered and equal, the corresponding element in the vrd is set to all one. Otherwise, it is set to all zero.

The quiet compare operation is defined by the IEEE-754-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MCSR register. In case of a floating point exception, the default result has all bits set to 0.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpeq(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.10.2 FCLEW

Floating point Quiet Compare Less or Equal **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1111	11

Syntax:

FCLEW vrd, vrs, vrp

Description：

Take each single-precision floating point element in the vrs to compare with the corresponding one in the vrp, if they are ordered and the element from the vrs is less than or equal to the one from the vrp, the corresponding element in the vrd is set to all one. Otherwise, it is set to all zero.

The quiet compare operation is defined by the IEEE-754-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MCSR register. In case of a floating point exception, the default result has all bits set to 0.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fple(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.10.3 FCLTW

Floating point Quiet Compare Less Than **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1101	11

Syntax:

FCLTW vrd, vrs, vrp

Description：

Take each single-precision floating point element in the vrs to compare with the corresponding one in the vrp, if they are ordered and the element from the vrs is less than the one from the vrp, the corresponding element in the vrd is set to all one. Otherwise, it is set to all zero.

The quiet compare operation is defined by the IEEE-754-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MCSR register. In case of a floating point exception, the default result has all bits set to 0.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fplt(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.10.4 FCORW

Floating point Quiet Compare Ordered **COP2**

Bit	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
	010010	10000	vrs	vrp	vrd	1100	11

Syntax:

FCORW vrd, vrs, vrp

Description：

Take each single-precision floating point element in the vrs to compare with the corresponding one in the vrp, if they are ordered (both floating point elements value are not NaN values), the corresponding element in the vrd is set to all one. Otherwise, it is set to all zero.

The quiet compare operation is defined by the IEEE-754-2008.

The Inexact Exception is not signaled when subnormal input operands are flushed based on the flush-to-zero bit FS in MCSR register. In case of a floating point exception, the default result has all bits set to 0.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/esize
    VPR[vrd][esize,i]= fpor(op1[esize,i] , op2[esize,i])
```

Exceptions:

RI, CpU, MFPE

3.11 Floating Point Conversion

3.11.1 FFSIW

Signed Integer Convert to Single **SPECIAL2**

Bit	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
	011100						00010				vrs					00000					vrd					101110							

Syntax:

FFSIW vrd, vrs

Description：

Each 32-bit signed integer element in the vrs is converted to a single-precision floating point value, the result updates the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32:
    VPR[vrd][32,i]= sint_to_single(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.11.2 FFUIW

Unsigned 32-bit Integer Convert to Single **SPECIAL2**

Bit	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
	011100	00000	vrs	00000	vrd	101110

Syntax:

FFUIW vrd, vrs

Description：

Each 32-bit unsigned integer element in the vrs is converted to a single-precision floating point value, the result updates the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32:
    VPR[vrd][32,i]= uint_to_single(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.11.3 FTSIW

Single Convert to Signed Integer **SPECIAL2**

Bit	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
	011100	00111	vrs	00000	vrd	101110

Syntax:

FTSIW vrd, vrs

Description：

Each single-precision floating point element in the vrs is rounded and converted to a 32-bit signed integer value based on the rounding mode bits RM of the MCSR register, the result updates the correspoding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32:
    VPR[vrd][32,i]= single_to_sint(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.11.4 FTUIW

Single Convert to Unsigned Integer **SPECIAL2**

Bit	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
	011100	00101	vrs	00000	vrd	101110

Syntax:

FTUIW vrd, vrs

Description：

Each single-precision floating point element in the vrs is rounded and converted to an 32-bit unsigned integer value based on the rounding mode bits RM of the MCSR register, the result updates the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32:
    VPR[vrd][32,i]= single_to_uint(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.11.5 FRINTW

Single Rounding and Convert to Integer **SPECIAL2**

Bit	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
	011100	00001	vrs	00000	vrd	101110

Syntax:

FRINTW vrd, vrs

Description：

Each single-precision floating point element in the vrs is rounded to a 32-bit integral value with single-precision floating point data format based on the rounding mode bits RM of the MCSR register, the result updates the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32:
    VPR[vrd][32,i]= single_rto_ sint(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.11.6 FTRUNCSW

Single Truncate to Signed Integer **SPECIAL2**

Bit	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
	011100	00110	vrs	00000	vrd	101110

Syntax:

FTRUNCSW vrd, vrs

Description：

Each single-precision floating point element in the vrs is truncated ,i.e. rounded toward zero, to a 32-bit signed integer value , the result updates the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32:
    VPR[vrd][32,i]= single_tto_ sint(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.11.7 FTRUNCUW

Single Truncate to Unsigned Integer **SPECIAL2**

Bit	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
	011100	00100	vrs	00000	vrd	101110

Syntax:

FTRUNCUW vrd, vrs

Description：

Each single-precision floating point element in the vrs is truncated, i.e. rounded toward zero, to a 32-bit unsigned integer value, the result updates the corresponding one in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
for i in MLEN/32
    VPR[vrd][32,i]= single_tto_ uint(op1[32,i])
```

Exceptions:

RI, CpU, MFPE

3.12 Shift

3.12.1 SLL<fmt>

Shift Left **COP2**

Bit	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
	010010						10001				vrs					vrp					vrd					1000				fmt			

Syntax:

SLLB	vrd, vrs, vrp	//fmt=2'B00
SLLH	vrd, vrs, vrp	//fmt=2'B01
SLLW	vrd, vrs, vrp	//fmt=2'B10

Description：

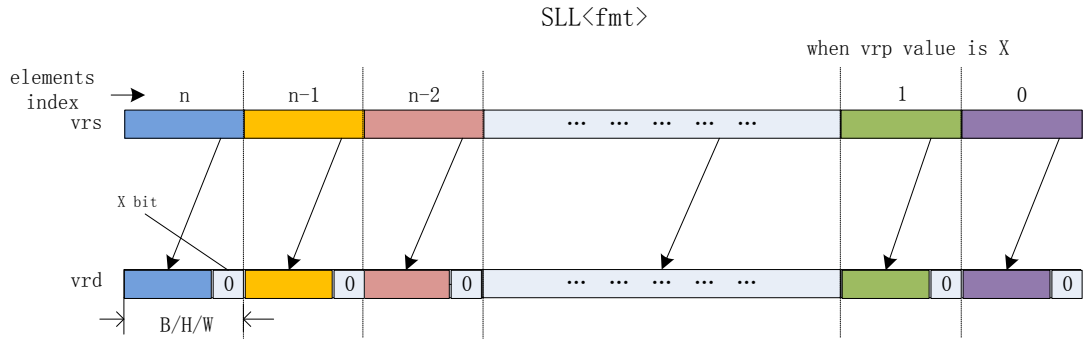
Take each element in the vrs to shift left with a bit-shift amount being specified by the value of the corresponding one in the vrp, inserting zeros to the emptied lower-order bits, the shifted result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [log2(esize)-1:0] shift_amount
bit[esize-1:0] temp
for i in MLEN/esize
    shift_amount      = op2[esize,i][log2(esize)-1:0]
    temp              = op1[esize,i]
    VPR[vrd][esize,i] = {temp[esize-1-shift_amount:0], shift_amount{0}}
```

Exceptions:

RI, CpU



3.12.2 SLLI<fmt>

Immediate Shift Left **COP2**

Bit	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
	010010	10101	vrs	imm	vrd	1000	fmt

Syntax:

SLLIB	vrd, vrs, imm	//fmt=2'B00
SLLIH	vrd, vrs, imm	//fmt=2'B01
SLLIW	vrd, vrs, imm	//fmt=2'B10

Description：

Take each element in the vrs to shift left with a common bit-shift amount being specified by the immediate value from the <imm>, inserting zeros to the emptied lower-order bits, the shifted result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [log2(esize)-1:0] shift_amount=imm[log2(esize)-1:0]
bit[esize-1:0] temp
for i in MLEN/esize
    temp            = op1[esize,i];
    VPR[vrd][esize,i]  = {temp[esize-1-shift_amount:0], shift_amount{0}}
```

Exceptions:

RI, CpU

3.12.3 SRA<fmt>

Arithmetic Shift Right **COP2**

Bit	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
	010010	10001	vrs	vrp	vrd	1110	fmt

Syntax:

SRAB	vrd, vrs, vrp	//fmt=2'B00
SRAH	vrd, vrs, vrp	//fmt=2'B01
SRAW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a bit-shift amount being specified by the value of the corresponding one in the vrp, duplicating sign-bit (the most significant bit of the taken element from vrs) in the emptied higher-order bits, the shifted result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [log2(esize)-1:0] shift_amount
bit[esize-1:0] temp
bit sign
for i in MLEN/esize
    shift_amount      = op2[esize,i] [log2(esize)-1:0]
    temp              = op1[esize,i]
    sign              = temp[esize-1]
    VPR[vrd][esize,i] = {shift_amount{sign}, temp[esize-1:shift_amount]}
```

Exceptions:

RI, CpU

3.12.4 SRAI<fmt>

Arithmetic Shift Right Immediate **COP2**

Bit	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
	010010						10101					vrs					imm					vrd					1110				fmt	

Syntax:

SRAIB	vrd, vrs, imm	//fmt=2'B00
SRAIH	vrd, vrs, imm	//fmt=2'B01
SRAIW	vrd, vrs, imm	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a common bit-shift amount being specified by the immediate value from the <imm>, duplicating sign-bit (the most significant bit of the taken element from vrs) in the emptied higher-order bits, the shifted result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [log2(esize)-1:0] shift_amount=imm[log2(esize)-1:0]
bit[esize-1:0] temp
bit sign
for i in MLEN/esize
    temp              = op1[esize,i]
    sign              = temp[esize-1]
    VPR[vrd][esize,i] = {shift_amount{sign}, temp[esize-1:shift_amount]}
```

Exceptions:

RI, CpU

3.12.5 SRAR<fmt>

Arithmetic Rounding Shift Right **COP2**

Bit	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
	010010	10001	vrs	vrp	vrd	1111	fmt

Syntax:

SRARB	vrd, vrs, vrp	//fmt=2'B00
SRARH	vrd, vrs, vrp	//fmt=2'B01
SRARW	vrd, vrs, vrp	//fmt=2'B10

Description：

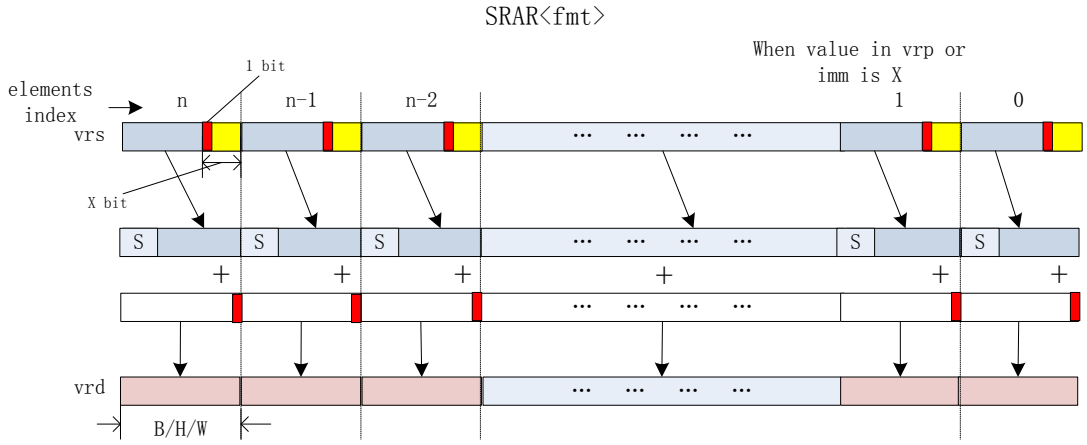
Take each element in the vrs to shift right with a bit-shift amount being specified by the value of the corresponding one in the vrp, duplicating sign-bit (the most significant bit of the taken element from vrs) in the emptied higher-order bits, then add the last bit being shifted away from the element of vrs to form the final rounding result to update the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [log2(esize)-1:0] shift_amount
bit[esize-1:0] temp
bit sign
bit round
for i in MLEN/esize
    shift_amount      = op2[esize,i] [log2(esize)-1:0]
    temp              = op1[esize,i]
    sign              = temp[esize-1]
    round             = temp[shift_amount-1]
    VPR[vrd][esize,i] = {shift_amount{sign} , temp[esize-1:shift_amount]} + round
```

Exceptions:

RI, CpU



3.12.6 SRARI<fmt>

Arithmetic Rounding Shift Right Immediate **COP2**

Bit	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
	010010	10101	vrs	imm	vrd	1111	fmt

Syntax:

SRARIB	vrd, vrs, imm	//fmt=2'B00
SRARIH	vrd, vrs, imm	//fmt=2'B01
SRARIW	vrd, vrs, imm	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a bit-shift amount being specified by the immediate value from the <imm>, duplicating sign-bit (the most significant bit of the taken element from vrs) in the emptied higher-order bits, then add the last bit being shifted away from the element of vrs to form the final rounding result to update the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [log2(esize)-1:0] shift_amount=imm[log2(esize)-1:0]
bit[esize-1:0] temp
bit sign
bit round
for i in MLEN/esize
    temp = op1[esize,i]
    sign = temp[esize-1]
    round =temp[shift_amount-1]
    VPR[vrd][esize,i]={shift_amount{sign} , temp[esize-1:shift_amount]}+round
```

Exceptions:

RI, CpU

3.12.7 SRL<fmt>

Logical Shift Right **COP2**

Bit	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
	010010	10001	vrs	vrp	vrd	1100	fmt

Syntax:

SRLB	vrd, vrs, vrp	//fmt=2'B00
SRLH	vrd, vrs, vrp	//fmt=2'B01
SRLW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a bit-shift amount being specified by the value of the corresponding one in the vrp, inserting zeros to the emptied higher-order bits, the shifted result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [log2(esize)-1:0] shift_amount
bit[esize-1:0] temp
for i in MLEN/esize
    temp=op1[esize,i]
    shift_amount = op2[esize,i] [log2(esize)-1:0]
    VPR[vrd][esize,i]={shift_amount{0} , temp[esize-1:shift_amount]}
```

Exceptions:

RI, CpU

3.12.8 SRLI<fmt>

Logical Shift Right Immediate **COP2**

Bit	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
	010010	10101	vrs	imm	vrd	1100	fmt

Syntax:

SRLIB	vrd, vrs, imm	//fmt=2'B00
SRLIH	vrd, vrs, imm	//fmt=2'B01
SRLIW	vrd, vrs, imm	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a common bit-shift amount being specified by the immediate value from the <imm>, inserting zeros to the emptied higher-order bits, the shifted result updates the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [log2(esize)-1:0] shift_amount=imm[log2(esize)-1:0]
bit[esize-1:0] temp
for i in MLEN/esize
    temp=op1[esize,i]
    VPR[vrd][esize,i]={shift_amount{0} , temp[esize-1:shift_amount]}
```

Exceptions:

RI, CpU

3.12.9 SRLR<fmt>

Logical Rounding Shift Right **COP2**

Bit	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
	010010	10001	vrs	vrp	vrd	1101	fmt

Syntax:

SRLRB	vrd, vrs, vrp	//fmt=2'B00
SRLRH	vrd, vrs, vrp	//fmt=2'B01
SRLRW	vrd, vrs, vrp	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a bit-shift amount being specified by the value of the corresponding one in the vrp, inserting zeros to the emptied higher-order bits, then the shifted result is added with the last bit of the element being shifted away to form the final rounding result to update the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [log2(esize)-1:0] shift_amount
bit[esize-1:0] temp
bit round
for i in MLEN/esize
    shift_amount = op2[esize,i] [log2(esize)-1:0]
    temp=op1[esize,i]
    round= temp[shift_amount-1]
    VPR[vrd][esize,i]= {shift_amount{0} , temp[esize-1:shift_amount]}+round
```

Exceptions:

RI, CpU

3.12.10 SRLRI<fmt>

Logical Rounding Shift Right Immediate **COP2**

Bit	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
	010010	10101	vrs	imm	vrd	1101	fmt

Syntax:

SRLRIB	vrd, vrs, imm	//fmt=2'B00
SRLRIH	vrd, vrs, imm	//fmt=2'B01
SRLRIW	vrd, vrs, imm	//fmt=2'B10

Description：

Take each element in the vrs to shift right with a common bit-shift amount being specified by the immediate value from the <imm>, inserting zeros to the emptied higher-order bits, then the shifted result is added with the last bit of the element being shifted away to form the final rounding result to update the corresponding element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [log2(esize)-1:0] shift_amount=imm[log2(esize)-1:0]
bit[esize-1:0] temp
bit round
for i in MLEN/esize
    temp=op1[esize,i]
    round=temp[shift_amount-1]
    VPR[vrd][esize,i]= {shift_amount{0} , temp[esize-1:shift_amount]}+round
```

Exceptions:

RI, CpU

3.13 Element Shuffle/Permute

3.13.1 GT<fmt>

Gather low & high element from a pair elements, group the low & high together SPECIAL2

Bit	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
	011100	01110	0	0	fmt	vrp	vrd	110100

Syntax:

GT1BI	vrd, vrp;	//fmt=3'B100
GT2BI	vrd, vrp;	//fmt=3'B101
GT4BI	vrd, vrp;	//fmt=3'B110
GTB	vrd, vrp;	//fmt=3'B000
GTH	vrd, vrp;	//fmt=3'B001

Description：

For each word-pair in the vrp, its included odd esize-elements need be gathered to update the odd-word of the corresponding word-pair in the vrd, and its included even esize-elements need be gathered to update the even-word of the corresponding word-pair in the vrd..

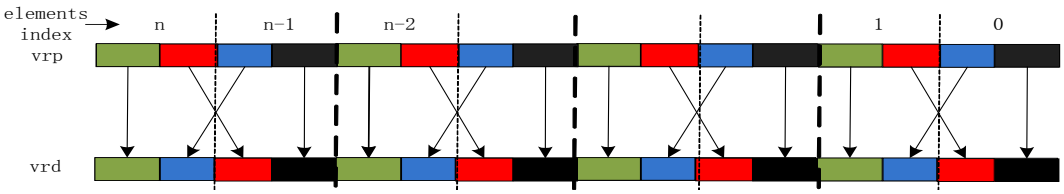
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
bit [31:0] result_low
bit [31:0] result_high
for j in MLEN/(2*32)
    result_low      = 0
    result_high     = 0
    for i in 32/esize
        result_low[esize,i]  = op1[esize,2*i+j*2*32/esize]
        result_high[esize,i]= op1[esize,2*i+1+j*2*32/esize]
    VPR[vrd][64,j]= {result_high, result_low}
```

Exceptions:

RI, CpU

GTH VRd, VRp



3.13.2 GT<n><fmt>

Gather dispersed element together **SPECIAL2**

Bit	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
	011100	fmt_pos	n_m	vrp	vrd	110101

Syntax:

GT2W	vrd[m], vrp, pos;	//n_m=0001m, fmt_pos=0000p: n=2, m=0~1, pos=0~1
GT4W	vrd[m], vrp, pos;	//n_m=001mm, fmt_pos=000pp: n=4, m=0~3, pos=0~3
GT8W	vrd[m], vrp, pos;	//n_m=01mmm, fmt_pos=00ppp: n=8, m=0~7, pos=0~7, valid for SIMD1024 & SIMD512
GT16W	vrd[m], vrp, pos;	//n_m=1mmmm, fmt_pos=0pppp: n=16, m=0~15, pos=0~15, valid for SIMD1024
GT2D	vrd[m], vrp, pos;	//n_m=0001m, fmt_pos=1000p: n=2, m=0~1, pos=0~1
GT4D	vrd[m], vrp, pos;	//n_m=001mm, fmt_pos=100pp: n=4, m=0~3, pos=0~3, valid for SIMD1024 & SIMD512
GT8D	vrd[m], vrp, pos;	//n_m=01mmm, fmt_pos=10ppp: n=8, m=0~7, pos=0~7, valid for SIMD1024
GT2Q	vrd[m], vrp, pos;	//n_m=0001m, fmt_pos=1100p: n=2, m=0~1, pos=0~1, valid for SIMD1024 & SIMD512
GT4Q	vrd[m], vrp, pos;	//n_m=001mm, fmt_pos=110pp: n=4, m=0~3, pos=0~3, valid for SIMD1024
GT2O	vrd[m], vrp, pos;	//n_m=0001m, fmt_pos=1110p: n=2, m=0~1, pos=0~1, valid for SIMD1024

Description：

Pick every <pos>-th element from every <n>-element group in the vrp, concatenate them together to form a data with size of MLEN/n to update the vrd. The location of the vrd to be updated starts from the element position specified by the <m>, and the rest of the vrd keeps unchanged.

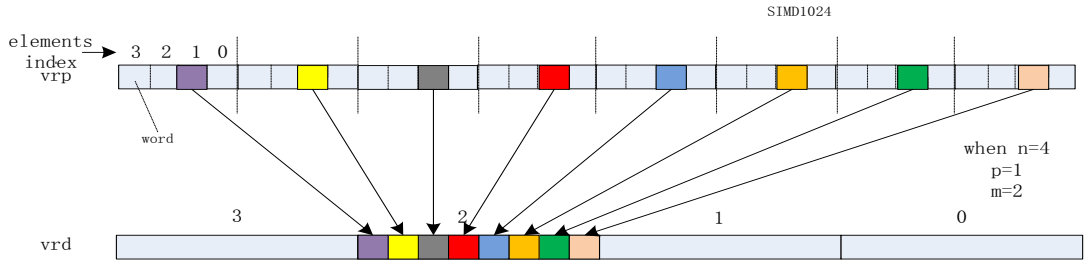
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
rsize= MLEN/n
bit [rsize-1:0] result
for i in MLEN/(n*esize)
    result[esize,i]= op1[esize,i*n+pos]
VPR[vrd][rsize,m]= result
```

Exceptions:

RI, CpU

GTnW VRd[m],VRp,p



3.13.3 EXTU<fmt><p><zp>

Zero Extend Element to Twice Size **SPECIAL2**

Bit	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
	011100	1000	w	p	zp	fmt	vrp	vrd	110001	

Syntax:

EXTUWLL	vrd, vrp,w;	//p=0, zp=0, fmt=3'B000
EXTUWLH	vrd, vrp,w;	//p=0, zp=1, fmt=3'B000
EXTUWHL	vrd, vrp,w;	//p=1, zp=0, fmt=3'B000
EXTUWHH	vrd, vrp,w;	//p=1, zp=1, fmt=3'B000
EXTUDLL	vrd, vrp,w;	//p=0, zp=0, fmt=3'B001
EXTUDLH	vrd, vrp,w;	//p=0, zp=1, fmt=3'B001
EXTUDHL	vrd, vrp,w;	//p=1, zp=0, fmt=3'B001
EXTUDHH	vrd, vrp,w;	//p=1, zp=1, fmt=3'B001
EXTUQLL	vrd, vrp,w;	//p=0, zp=0, fmt=3'B010
EXTUQLH	vrd, vrp,w;	//p=0, zp=1, fmt=3'B010
EXTUQHL	vrd, vrp,w;	//p=1, zp=0, fmt=3'B010
EXTUQHH	vrd, vrp,w;	//p=1, zp=1, fmt=3'B010
EXTUOLL	vrd, vrp,w;	//p=0, zp=0, fmt=3'B011, valid for SIMD1024 & SIMD512
EXTUOLH	vrd, vrp,w;	//p=0, zp=1, fmt=3'B011, valid for SIMD1024 & SIMD512
EXTUOHL	vrd, vrp,w;	//p=1, zp=0, fmt=3'B011, valid for SIMD1024 & SIMD512
EXTUOHH	vrd, vrp,w;	//p=1, zp=1, fmt=3'B011, valid for SIMD1024 & SIMD512
EXTUXLL	vrd, vrp,w;	//p=0, zp=0, fmt=3'B100, valid for SIMD1024
EXTUXLH	vrd, vrp,w;	//p=0, zp=1, fmt=3'B100, valid for SIMD1024
EXTUXHL	vrd, vrp,w;	//p=1, zp=0, fmt=3'B100, valid for SIMD1024
EXTUXHH	vrd, vrp,w;	//p=1, zp=1, fmt=3'B100, valid for SIMD1024

Description：

The low(<p> =0) or high(<p> =1) half part of elements in the vrp are zero-extended to twice of their original size, then for each result with pattern-{element, 0} (<zp>=0) or pattern-{0, element} (<zp>=1), if <w>=1, the entire result updates the corresponding twice-esize element in the vrd, otherwise, only the element part of the result updates the corresponding esize element in the vrd.

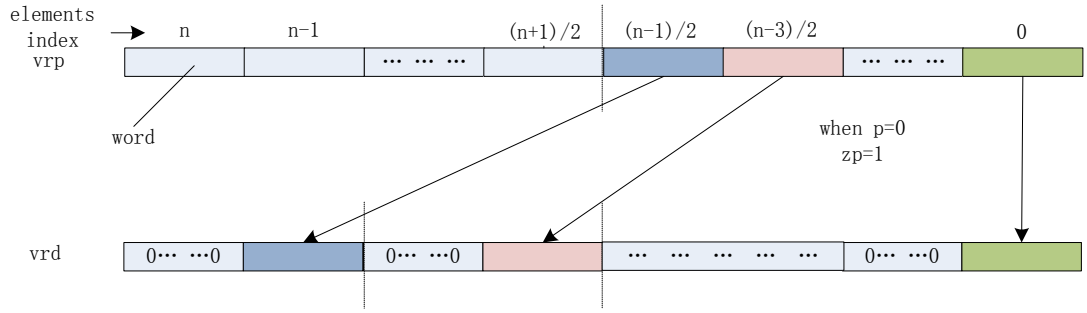
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
for i in MLEN/(2*esize)
    if (w == 1) VPR[vrd][esize,2*i+zp]= {esize{0}}
    VPR[vrd][esize,2*i+1-zp]= op1[esize,i+ p*MLEN/(2*esize)]
```

Exceptions:

RI, CpU

EXTU <fmt> <p> <zp>



3.13.4 EXT<s><fmt><p>

Extend Element to Twice Size **SPECIAL2**

Bit	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
	011100	01101	s	p	fmt	vrp	vrd	110001

Syntax:

```
EXTU1BIL    vrd, vrp;    //s=0, p=0, fmt=3'B100
EXTU2BIL    vrd, vrp;    //s=0, p=0, fmt=3'B101
EXTU4BIL    vrd, vrp;    //s=0, p=0, fmt=3'B110
EXTUBL      vrd, vrp;    //s=0, p=0, fmt=3'B000
EXTUHL      vrd, vrp;    //s=0, p=0, fmt=3'B001
EXTU1BIH    vrd, vrp;    //s=0, p=1, fmt=3'B100
EXTU2BIH    vrd, vrp;    //s=0, p=1, fmt=3'B101
EXTU4BIH    vrd, vrp;    //s=0, p=1, fmt=3'B110
EXTUBH      vrd, vrp;    //s=0, p=1, fmt=3'B000
EXTUHH      vrd, vrp;    //s=0, p=1, fmt=3'B001
EXTS1BIL    vrd, vrp;    //s=1, p=0, fmt=3'B100
EXTS2BIL    vrd, vrp;    //s=1, p=0, fmt=3'B101
EXTS4BIL    vrd, vrp;    //s=1, p=0, fmt=3'B110
EXTSBL      vrd, vrp;    //s=1, p=0, fmt=3'B000
EXTSHL      vrd, vrp;    //s=1, p=0, fmt=3'B001
EXTS1BIH    vrd, vrp;    //s=1, p=1, fmt=3'B100
EXTS2BIH    vrd, vrp;    //s=1, p=1, fmt=3'B101
EXTS4BIH    vrd, vrp;    //s=1, p=1, fmt=3'B110
EXTSBH      vrd, vrp;    //s=1, p=1, fmt=3'B000
EXTSHH      vrd, vrp;    //s=1, p=1, fmt=3'B001
```

Description：

The low(<p> =0) or high(<p> =1) half part of elements in the vrp are sign-extended (<s>=1) or unsign-extended (<s>=0) to twice of their original size, then each result with pattern {extension, element} updates the corresponding twice-esize element in the vrd.

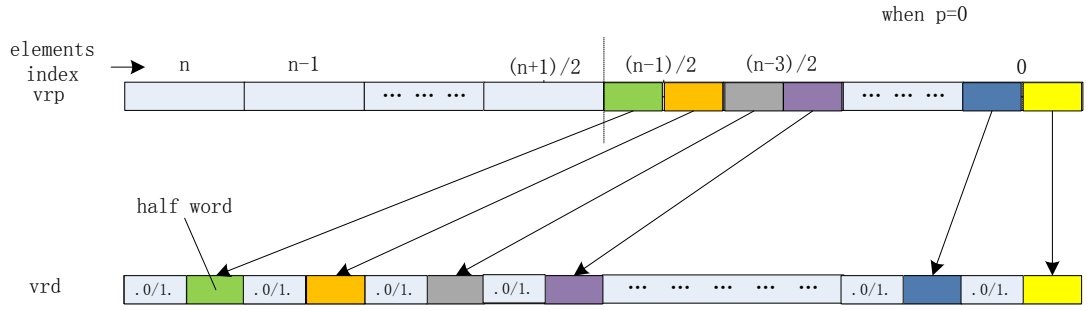
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
bit [esize-1:0] element
bit sign
for i in MLEN/(2*esize)
    element= op1[esize,i+ p*MLEN/(2*esize)]
    sign=element[esize-1] & s
    VPR[vrd][2*esize,i]= {esize{sign},element}
```

Exceptions:

RI, CpU

EXTS <fmt> <p>



3.13.5 EXTU3BW

Zero Extend every 3-Byte to a Word **SPECIAL2**

Bit	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
	011100	01100	00000	vrp	vrd	110001

Syntax:

EXTU3BW vrd, vrp;

Description：

For each 3-byte element in the vrp, cancatenate a leading zero byte to form a new word data to update the corresponding word element in the vrd.

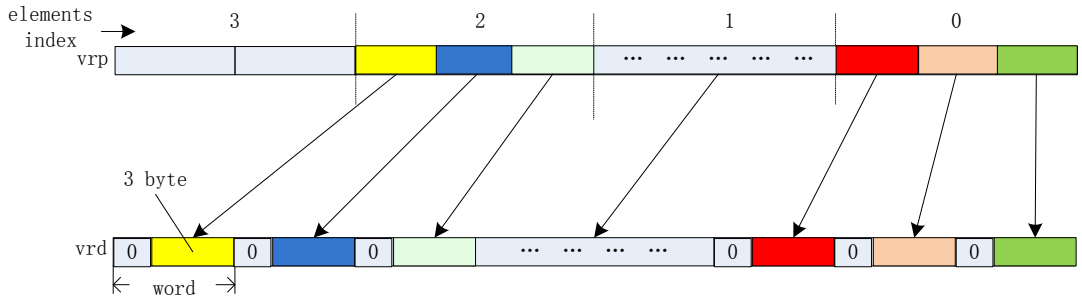
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
for i in MLEN/32
    VPR[vrd][32,i]= {8{0}, op1[24,i]}
```

Exceptions:

RI, CpU

EXT U 3B W



3.13.6 REPI<fmt_n>

Replicate Vector Element by Immediate **2R8I**

Bit	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
	011100						00		fmt_n								vrp						vrd						110000			

Syntax:

REPIB	vrd, vrp[n]	//n=0~127, fmt_n=0nnnnnnn	*1
REPIH	vrd, vrp[n]	//n=0~63, fmt_n=10nnnnnn	
REPIW	vrd, vrp[n]	//n=0~31, fmt_n=110nnnnn	
REPID	vrd, vrp[n]	//n=0~15, fmt_n=1110nnnn	
REPIQ	vrd, vrp[n]	//n=0~7, fmt_n=11110nnn	
REPIO	vrd, vrp[n]	//n=0~3, fmt_n=111110nn	valid for SIMD1024 & SIMD512
REPIX	vrd, vrp[n]	//n=0~1, fmt_n=1111110n	valid for SIMD1024

*1: for SIMD256, maximum value of n is 31; for SIMD512, maximum value of n is 63.

Description：

Replicate the <n>-th element in the vrp to form a new MLEN-bit vector value to update the vrd.

Operation：

check_cop2_enable()

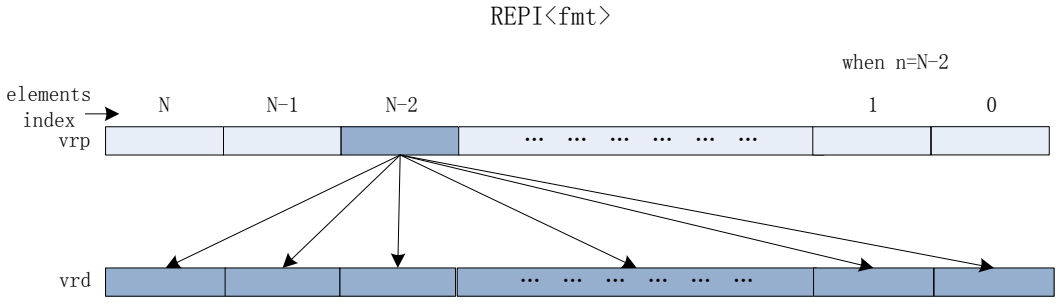
bit [MLEN-1:0] op1=VPR[vrs]

for i in MLEN/esize

VPR[vrd][esize,i]= op1[esize,n]

Exceptions:

RI, CpU



3.13.7 MOVW

Move one Word from <n>th Element to <m>th **SPECIAL2**

Bit	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
	011100	m	n	vrp	vrd	110110

Syntax:

MOVW vrd[m],vrp[n]

Description：

Take the <n>th word element in the vrp to update the <m>th word element in the vrd, all other word elements in the vrd are unchanged.

Operation：

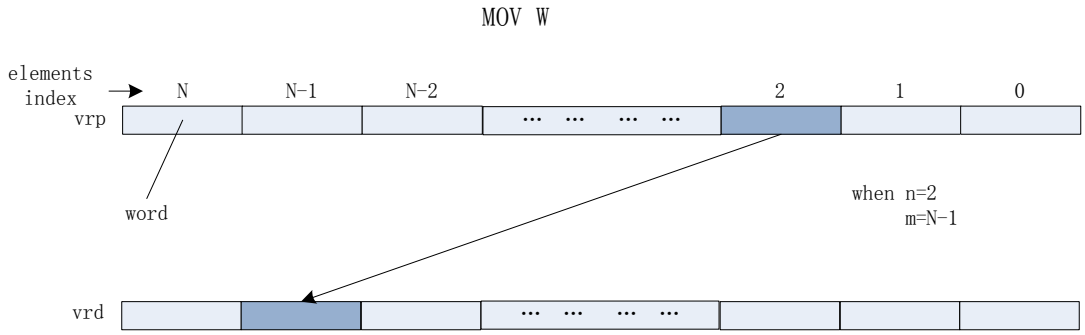
check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

VPR[vrd][32,m]= op1[32,n]

Exceptions:

RI, CpU



3.13.8 MOV<fmt>

Move one Element from <n>th Element to <m>th **SPECIAL2**

Bit	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
	011100						fmt 1		m				fmt 0		n		vrp						vrd						110111			

Syntax:

MOVD	vrd[m],vrp[n]	//fmt_1=0, fmt_0=0	
MOVQ	vrd[m],vrp[n]	//fmt_1=0, fmt_0=1	
MOVO	vrd[m],vrp[n]	//fmt_1=1, fmt_0=0	valid for SIMD1024 & SIMD512
MOVX	vrd[m],vrp[n]	//fmt_1=1, fmt_0=1	valid for SIMD1024

Description：

Take the <n>th element in the vrp to update the <m>th element in the vrd, all other elements in the vrd are unchanged.

Operation：

check_cop2_enable()

bit [MLEN-1:0] op1=VPR[vrs]

VPR[vrd][esize,m]= op1[esize,n]

Exceptions:

RI, CpU

3.13.9 SHUF<fmt><grp>

Shuffle Elements from 2 or More Groups **SPECIAL2**

Bit	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
	011100	0	grp	fmt	00000	vrp	vrd	110011

Syntax:

SHUFW2	vrd, vrp	//fmt=2'B00, grp=2'B00(n=2)	
SHUFW4	vrd, vrp	//fmt=2'B00, grp=2'B01(n=4)	
SHUFW8	vrd, vrp	//fmt=2'B00, grp=2'B10(n=8)	valid for SIMD1024 & SIMD512
SHUFW16	vrd, vrp	//fmt=2'B00, grp=2'B11(n=16)	valid for SIMD1024
SHUFD2	vrd, vrp	//fmt=2'B01, grp=2'B00(n=2)	
SHUFD4	vrd, vrp	//fmt=2'B01, grp=2'B01(n=4)	valid for SIMD1024 & SIMD512
SHUFD8	vrd, vrp	//fmt=2'B01, grp=2'B10(n=8)	valid for SIMD1024
SHUFQ2	vrd, vrp	//fmt=2'B10, grp=2'B00(n=2)	valid for SIMD1024 & SIMD512
SHUFQ4	vrd, vrp	//fmt=2'B10, grp=2'B01(n=4)	valid for SIMD1024
SHUFO2	vrd, vrp	//fmt=2'B11, grp=2'B00(n=2)	valid for SIMD1024

Description：

Divide the vrp into equivalent <n> groups first. Then pick the position-0 element from every group and gather them to form a new data with pattern {element0 of group$_{n-1}$,..., element0 of group$_0$}, repeat the process for each included element belonging to divided groups. Finally concatenate all new data that has the same data width of esize*n to update the vrd.

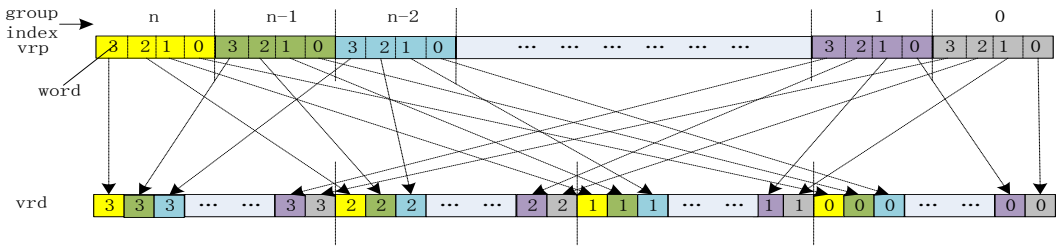
Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
for j in MLEN/(group_n*esize)
    for i in group_n
        VPR[vrd][esize,i+j*group_n]= op1[esize,j+i* MLEN/(group_n*esize)]
```

Exceptions:

RI, CpU

SHUF{W|D|Q|O}{2|4|8|16} VRd, VRp



3.13.10 GSHUFW

Generic Word Shuffle **SPECIAL2**

Bit	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
	011100	100	imm	vrs	vrp	vrd	111011

Syntax:

GSHUFW vrd, vrs,vrp,imm

Description：

For each word element in the vrs, use <imm> as index to choose a byte from the word, and then use this byte element as index to choose a word element from the vrp to update the corresponding word element in the vrd.

In addition, if the index value is 0x80, then just write 0 to the corresponding word element in the vrd, and if the index value is 0xc0, the corresponding word element in the vrd remains unchanged.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] element
bit [7:0] w_sel, mask
if      (MLEN == 1024) mask = 0x1f
else if (MLEN ==  512) mask = 0x0f
else if (MLEN ==  256) mask = 0x07
for i in MLEN/esize
    element= op1[esize,i]
    w_sel= element[8,imm]
    if      (w_sel==0x80)   VPR[vrd][esize,i]={esize{0}}
    else if(w_sel==0xc0)    VPR[vrd][esize,i]=VPR[vrd][esize,i]
    else
        w_sel = w_sel & mask
        VPR[vrd][esize,i]=op2[esize,w_sel]
```

Exceptions:

RI, CpU

3.13.11 GSHUFWB

Generic Word Shuffle + Byte Shuffle within Word **SPECIAL2**

Bit	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
	011100	11000	vrs	vrp	vrd	111011

Syntax:

GSHUFWB vrd, vrs, vrp

Description:

For each word element in the vrs, use its bit7~0 as index to choose a word element from the vrp, use its bit9~8 to further choose a byte from the chosen word for destination byte0, similarly, bit11~10 for destination byte1, bit13~12 for destination byte2 and bit15~14 for destination byte3, finally the destination byte collection {byte3, byte2, byte1, byte0} is used to update the corresponding word element in the vrd.

In addition, if the index (bit7~0) value is 0x80, just write 0 to the corresponding word element in the vrd, and if the index (bit7~0) value is 0xc0, the corresponding word element in the vrd remains unchanged.

Operation:

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] element, temp
bit [7:0] w_sel, mask
bit [1:0] b_sel0, b_sel1, b_sel2, b_sel3
if      (MLEN == 1024) mask = 0x1f
else if (MLEN ==  512) mask = 0x0f
else if (MLEN ==  256) mask = 0x07
for i in MLEN/esize
    element= op1[esize,i]
    w_sel= element[7:0]
    if      (w_sel==0x80)   VPR[vrd][esize,i]={esize{0}}
    else if(w_sel==0xc0)    VPR[vrd][esize,i]= VPR[vrd][esize,i]
    else
        w_sel = w_sel & mask
        temp=op2[esize, w_sel]
        b_sel0= element[9:8]
        b_sel1= element[11:10]
        b_sel2= element[13:12]
        b_sel3= element[15:14]
        VPR[vrd][esize,i]={temp[8,b_sel3], temp[8,b_sel2], temp[8,b_sel1], temp[8,b_sel0]}
```

Exceptions:

RI, CpU

3.13.12 GSHUFB

Generic Byte Shuffle within Word **SPECIAL2**

Bit	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
	011100	00001	vrs	vrp	vrd	101011

Syntax:

GSHUFB vrd, vrs, vrp

Description：

For each word element in the vrs, use its lower 8-bit to choose 4 bytes from the corresponding word element in the vrp, in detail, bit1~0 as index for byte0, bit3~2 as index for byte1, bit5~4 as index for byte2 and bit7~6 as index for byte3, finally, the byte collection {byte3, byte2, byte1, byte0} is used to update the corresponding word element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] element, temp
bit [1:0] b_sel0, b_sel1, b_sel2, b_sel3
for i in MLEN/esize
    element= op1[esize,i]
    temp= op2[esize,i]
    b_sel0= element[1:0]
    b_sel1= element[3:2]
    b_sel2= element[5:4]
    b_sel3= element[7:6]
    VPR[vrd][esize,i]={temp[8,b_sel3], temp[8,b_sel2], temp[8,b_sel1], temp[8,b_sel0]}
```

Exceptions:

RI, CpU

3.13.13 GSHUFWH

Generic Word Shuffle + Halfword Shuffle within Word **SPECIAL2**

Bit	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
	011100	10000	vrs	vrp	vrd	101110

Syntax:

GSHUFWH vrd, vrs, vrp

Description：

For each word element in the vrs, use its bit7~0 as index to choose a word element from the vrp, use its bit8 to further choose a halfword from the chosen word for destination halfword0, similarly, bit9 for destination halfword1, finally the destination halfword collection {halfword1, halfword0} is used to update the corresponding word element in the vrd.

In addition, if the index (bit7~0) value is 0x80, just write 0 to the corresponding word element in the vrd, and if the index (bit7~0) value is 0xc0, the corresponding word element in the vrd remains unchanged.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] element, temp
bit [7:0] w_sel, mask
bit [0:0] h_sel0, h_sel1
if      (MLEN == 1024) mask = 0x1f
else if (MLEN ==  512) mask = 0x0f
else if (MLEN ==  256) mask = 0x07
for i in MLEN/esize
    element= op1[esize,i]
    w_sel= element[7:0]
    if     (w_sel==0x80)   VPR[vrd][esize,i]={esize{0}}
    else if(w_sel==0xc0)   VPR[vrd][esize,i]=VPR[vrd][esize,i]
    else
        w_sel = w_sel & mask
        temp=op2[esize,w_sel]
        h_sel0= element[8]
        h_sel1= element[9]
        VPR[vrd][esize,i]={temp[16,h_sel1], temp[16,h_sel0]}
```

Exceptions:

RI, CpU

3.13.14 GSHUFH

Generic Halfword Shuffle within Word **SPECIAL2**

Bit	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
	011100	10100	vrs	vrp	vrd	101110

Syntax:

GSHUFH vrd, vrs, vrp

Description：

For each word element in the vrs, use its lower 2-bit to choose 2 halfwords from the corresponding word element in the vrp, in detail, bit0 as index for halfword0, bit1 as index for halfword1, finally, the halfword collection {halfword1, halfword0} is used to update the corresponding word element in the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
bit [esize-1:0] element, temp
bit [0:0] h_sel0, h_sel1
for i in MLEN/esize
    element= op1[esize,i]
    temp= op2[esize,i]
    h_sel0= element[0]
    h_sel1= element[1]
    VPR[vrd][esize,i]={temp[16,h_sel1], temp[16,h_sel0]}
```

Exceptions:

RI, CpU

3.13.15 ILVE<fmt>

Interleave Even Elements **SPECIAL2**

Bit	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
	011100	0	fmt	vrs	vrp	vrd	111000

Syntax:

ILVE2BI	vrd, vrs, vrp	//fmt=4'B0000
ILVE4BI	vrd, vrs, vrp	//fmt=4'B0001
ILVEB	vrd, vrs, vrp	//fmt=4'B0010
ILVEH	vrd, vrs, vrp	//fmt=4'B0011
ILVEW	vrd, vrs, vrp	//fmt=4'B0100
ILVED	vrd, vrs, vrp	//fmt=4'B0101
ILVEQ	vrd, vrs, vrp	//fmt=4'B0110
ILVEO	vrd, vrs, vrp	//fmt=4'B0111, valid for SIMD1024 & SIMD512
ILVEX	vrd, vrs, vrp	//fmt=4'B1000, valid for SIMD1024

Description：

Pick all even elements in the vrs and the vrp, shuffle them to form a new sequence in the way of the elements from vrs always locating at the even place, the new sequence is used to update the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*esize)
    VPR[vrd][esize,i*2]= op1[esize,i*2]
    VPR[vrd][esize,i*2+1]= op2[esize,i*2]
```

Exceptions:

RI, CpU

3.13.16 ILVO<fmt>

Interleave Odd Elements **SPECIAL2**

Bit	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
	011100	1	fmt	vrs	vrp	vrd	111000

Syntax:

ILVO2BI	vrd, vrs, vrp	//fmt=4'B0000
ILVO4BI	vrd, vrs, vrp	//fmt=4'B0001
ILVOB	vrd, vrs, vrp	//fmt=4'B0010
ILVOH	vrd, vrs, vrp	//fmt=4'B0011
ILVOW	vrd, vrs, vrp	//fmt=4'B0100
ILVOD	vrd, vrs, vrp	//fmt=4'B0101
ILVOQ	vrd, vrs, vrp	//fmt=4'B0110
ILVOO	vrd, vrs, vrp	//fmt=4'B0111, valid for SIMD1024 & SIMD512
ILVOX	vrd, vrs, vrp	//fmt=4'B1000, valid for SIMD1024

Description：

Pick all odd elements in the vrs and the vrp, shuffle them to form a new sequence in the way of the elements from vrs always locating at the even place, finally, the new sequence updates the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
for i in MLEN/(2*esize)
    VPR[vrd][esize,i*2+1]= op1[esize,i*2+1]
    VPR[vrd][esize,i*2]= op2[esize,i*2+1]
```

Exceptions:

RI, CpU

3.13.17 BSHLI

Byte Shift Left Immediate **SPECIAL2**

Bit	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
	011100	000	imm	vrp	vrd	111001

Syntax:

BSHLI vrd, vrp, imm //imm[6:0] for SIMD1024, imm[5:0] for SIMD512, imm[4:0] for SIMD256

Description：

Left shift the vrp, the byte-shift amount is specified by the immediate value from the <imm>, the emptied lower order bytes are set to all zero, the final shifted result updates the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
for i in MLEN/8
    VPR[vrd][8,i]= i < imm ? 0 : op1[8,i-imm]
```

Exceptions:

RI, CpU

3.13.18 BSHRI

Byte Shift Right Immediate **SPECIAL2**

Bit	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
	011100	010	imm	vrp	vrd	111001

Syntax:

```
BSHRI        vrd, vrp, imm        //imm[6:0] for SIMD1024, imm[5:0] for SIMD512, imm[4:0] for SIMD256
```

Description：

Right shift the vrp, the byte-shift amount is specified by the immediate value from the <imm>, the emptied higher order bytes are set to all zero, the final shifted result updates the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrp]
for i in MLEN/8
    VPR[vrd][8,i]= i >= (MLEN/8 - imm) ? 0 : op1[8,i+imm]
```

Exceptions:

RI, CpU

3.13.19 BSHL

Byte Shift Left **SPECIAL2**

Bit	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
	011100	00100	vrs	vrp	vrd	111001

Syntax:

BSHL vrd, vrs, vrp

Description：

Left shift the vrp, the byte-shift amount is specified by the lower bits of the vrs, in detail, bit6~0 for SIMD1024, bit5~0 for SIMD512 or bit4~0 for SIMD256. The emptied lower order bytes are set to all zero, the final shifted result updates the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
N=7 for SIMD1024
N=6 for SIMD512
N=5 for SIMD256
bit [N-1:0] imm=op1[N-1:0]
for i in MLEN/8
    VPR[vrd][8,i]= i < imm ? 0 : op2[8,i-imm]
```

Exceptions:

RI, CpU

3.13.20 BSHR

Byte Shift Right **SPECIAL2**

Bit	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
	011100	01100	vrs	vrp	vrd	111001

Syntax:

BSHR vrd, vrs, vrp

Description：

Right shift the vrp, the byte-shift amount is specified by the lower bits of the vrs, in detail, bit6~0 for SIMD1024, bit5~0 for SIMD512 or bit4~0 for SIMD256. The emptied higher order bytes are set to all zero, the final shifted result updates the vrd.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1=VPR[vrs]
bit [MLEN-1:0] op2=VPR[vrp]
N=7 for SIMD1024
N=6 for SIMD512
N=5 for SIMD256
bit [N-1:0] imm=op1[N-1:0]
for i in MLEN/8
    VPR[vrd][8,i]= i >= (MLEN/8 - imm) ? 0 : op2[8,i+imm]
```

Exceptions:

RI, CpU

3.14 Register Load and Misc

3.14.1 MTFPUW

Move VWR to FPR **SPECIAL2**

Bit	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
	011100						11000					00000				vwrp						fd					101110					

Syntax:

MTFPUW fd, vwrp

Description：

The 32-bit value in the VWR vwrp is copied to the FPR fd.

Operation：

check_cop2_enable()

N=4 for SIMD512

N=5 for SIMD1024

bit [31:0] op1=VWR[vwrp[N-1:0]]

FPR[fd]= op1

Exceptions:

RI, CpU

3.14.2 MFFPUW

Move from FPR to VWR **SPECIAL2**

Bit	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
	011100	11001	fs	00000	vwrd	101110

Syntax:

MFFPUW vwrd, fs

Description：

The value in the FPR fs is copied to the VWR vwrd.

Operation：

check_cop2_enable()

N=4 for SIMD512

N=5 for SIMD1024

bit [31:0] op1=FPR[fs]

VWR[vwrd[N-1:0]]= op1

Exceptions:

RI, CpU

3.14.3 CTCMXU

GPR Copy to MXU3 Control Register **SPECIAL2**

Bit	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
	011100	rs	10001	mcsrd	00000	000011

Syntax:

CTCMXU mcsrd, rs

Description：

The content of the GPR rs is copied to the MXU3 control register mcd.

Operation：

check_cop2_enable()

bit [31:0] op1=GPR[rs]

MCSR[mcd]= op1

Exceptions:

RI, CpU, MFPE

3.14.4 CFCMXU

GPR Copy from MXU3 Control Register **SPECIAL2**

Bit	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
	011100	00000	10000	mcsrs	rd	000011

Syntax:

CFCMXU rd, mcsrs

Description：

The content of the MXU3 control register mcs is copied to the GPR rd.

Operation：

check_cop2_enable()

bit [31:0] op1=MCSR[mcs]

GPR[rd]= op1

Exceptions:

RI, CpU

3.14.5 SUMZ

Zero a SUM Register **COP2**

Bit	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
	010010	10011	00000	00000	000	vsd	011100

Syntax:

SUMZ vsd

Description：

Clear the vsd.

Operation：

check_cop2_enable()

VSR[vsd]= 0

Exceptions:

RI, CpU

3.14.6 MFSUM

Move a SUM Register to VPR **COP2**

Bit	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
	010010	10011	00000	000	vss	vrd	001111

Syntax:

MFSUM vrd, vss

Description：

The content in the vss is copied to the vrd.

Operation：

check_cop2_enable()

VPR[vrd]= VSR[vss]

Exceptions:

RI, CpU

3.14.7 MFSUMZ

Move a SUM Register to VPR & Zero the SUM one **COP2**

Bit	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
	010010	10011	00000	000	vsd	vrd	011110

Syntax:

MFSUMZ vrd, vsd

Description：

The content in the vsd is moved to the vrd meanwhile clear the vsd.

Operation：

check_cop2_enable()

VPR[vrd]= VSR[vsd]

VSR[vsd]= 0

Exceptions:

RI, CpU

3.14.8 MTSUM

Move a VPR Register to SUM **COP2**

Bit	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
	010010	10011	vrs	00000	000	vsd	011101

Syntax:

MTSUM vsd, vrs

Description：

The content of the vrs is copied to the vsd.

Operation：

check_cop2_enable()

VSR[vsd]= VPR[vrs]

Exceptions:

RI, CpU

3.14.9 MXSUM

Exchange a SUM Register and a VPR Register **COP2**

Bit	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
	010010	10011	vrs	000	vsd	vrd	011111

Syntax:

MXSUM vrd, vrs, vsd

Description：

Move the value in the vsd to update the vrd, then copy the value in the vrs to the vsd.

Operation：

check_cop2_enable()

bit[MLEN-1:0] temp

temp = VSR[vsd]

VSR[vsd]= VPR[vrs]

VPR[vrd]=temp

Exceptions:

RI, CpU

3.14.10 PREFL1C (obsolete)

Prefetch a Block of Data to L1 Cache (DC) **SPECIAL2**

Bit	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
	011100	rs	0000	h	cl_num		000011

~~**Syntax:**~~

~~PREFL1C rs, cl_num, h~~

~~**Description:**~~

~~Prefetch number of cl_num cache lines between address of rs and rs + cl_num*32 to data cache. Cancel the prefetch is TLB miss found. The h specify a L2 cache hint. h=0 means normal operation. h=1 means bypass L2 cache, where if the cache line is not found in L2 cache, fetch it from DDR and not store a copy in L2 cache, if the cache line is found in L2 cache, fetch it from L2 cache and flush the copy in L2 cache.~~

~~**Operation:**~~

```
check_cop2_enable()
bit [31:0] op1=GPR[rs]
bit [26:0] cl_start= op1>>5
bit [26:0] cl_end= cl_start + cl_num
bit [31:0] addr
for i in cl_num
    addr = (cl_start + i) << 5
    prefetch_to_l1c(addr, h)
```

~~**Exceptions:**~~

~~RI, CpU~~

3.14.11 PREFL2C (obsolete)

Prefetch a Block of Data to L2 Cache **SPECIAL2**

Bit	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
	011100	rs	00010	cl_num		000011

~~**Syntax:**~~

~~PREFL2C rs, cl_num~~

~~**Description:**~~

~~Prefetch number of cl_num cache lines between address of rs and rs + cl_num*32 to L2 cache. Cancel the prefech is TLB miss found.~~

~~**Operation:**~~

~~check_cop2_enable()~~

~~bit [31:0] op1=GPR[rs]~~

~~bit [26:0] cl_start= op1>>5~~

~~bit [26:0] cl_end= cl_start + cl_num~~

~~bit [31:0] addr~~

~~for i in cl_num~~

~~addr = (cl_start + i) << 5~~

~~prefetch_to_l2c(addr)~~

~~**Exceptions:**~~

~~RI, CpU~~

3.14.12 LIH

Immediate Load to all H Element **COP2**

Bit	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
	010010	11000	imm[15:11]	imm[4:0]	vrd	imm[10:5]

Syntax:

LIH vrd, imm

Description：

The immediate <imm> is replicated to update all halfword elements in the vrd.

Operation：

check_cop2_enable()

for i in MLEN/16

VPR[vrd][16,i]= imm

Exceptions:

RI, CpU

3.14.13 LIW

Immediate Load to all Word Element **COP2**

Bit	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
	010010	11001	imm[15:11]	imm[4:0]	vrd	imm[10:5]

Syntax:

LIW vrd, imm

Description：

The immediate <imm> is sign-extended to 32-bit then replicated to update all word elements in the vrd.

Operation：

```
check_cop2_enable()
for i in MLEN/32
    VPR[vrd][32,i]= {16{imm[15]},imm}
```

Exceptions:

RI, CpU

3.14.14 LIWH

Immediate Load to all Word Element High Half **COP2**

Bit	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
	010010	11010	imm[15:11]	imm[4:0]	vrd	imm[10:5]

Syntax:

LIWH vrd, imm

Description：

The immediate <imm> is replicated to update all odd halfword elements in the vrd, the even ones in the vrd are left unchanged.

Operation：

```
check_cop2_enable()
for i in MLEN/32
    VPR[vrd][16,i*2+1]= imm
```

Exceptions:

RI, CpU

3.14.15 LIWR

Immediate Load to VWR **COP2**

Bit	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
	010010	10011	imm	vwrd	000000

Syntax:

LIWR vwrd, imm

Description：

The immediate <imm> is sign-extended to 32-bit to update the vwrd.

Operation：

check_cop2_enable()

VWR[vwrd]= sign_extend32(imm)

Exceptions:

RI, CpU

3.14.16 CMVW

Move a Word on Condition **SPECIAL2**

Bit	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
	011100	0	imm	vrs	vrp	vrd	110010

Syntax:

CMVW vrd, vrs, vrp, imm

Description：

For each word element in the vrp, use its lower 4 bits (bit3~0) to compare with the <imm>, if they are equal, take the corresponding word element in the vrs to update the correspoding one in the vrd, otherwise, the corresponding word in the vrd is unchanged.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1= VPR[vrs]
bit [MLEN-1:0] op2= VPR[vrp]
bit [31:0] cond
for i in MLEN/32
    cond= op2[32,i]
    if (cond[3:0]==imm)
        VPR[vrd][32,i]= op1[32,i]
```

Exceptions:

RI, CpU

3.14.17 PMAP<fmt>

Piecewise Mapping **COP2**

Bit	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
	010010						10011					vrs					vrp					vrd					0010				fmt	

Syntax:

PMAPH	vrd, vrs, vrp	//fmt=2'B01
PMAPW	vrd, vrs, vrp	//fmt=2'B10

Description：

Compare each signed integer element in the vrs with the corresponding one in the vrp, update the corresponding element in the vrd with different constant value according to different compare cases.

Operation：

```
check_cop2_enable()
bit [MLEN-1:0] op1= VPR[vrp]
bit [MLEN-1:0] op2= VPR[vrs]
for i in MLEN/esize
    if (op2[esize,i]<0)
        VPR[vrd][esize,i]=3
    else if (op1[esize,i] >= op2[esize,i])
        VPR[vrd][esize,i]= 3
    else if ((op1[esize,i] >= 0) && (op1[esize,i] < op2[esize,i]))
        VPR[vrd][esize,i]= 2
    else if ((op1[esize,i] < 0) && (op1[esize,i] >= (-op2[esize,i])))
        VPR[vrd][esize,i]= 1
    else if (op1[esize,i] < (-op2[esize,i]))
        VPR[vrd][esize,i]= 0
```

Exceptions:

RI, CpU

3.15 Load/Store

3.15.1 LU<fmt_n>

Load 1 Element and Update base **SPECIAL2**

Bit	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
	011100	base	00000	fmt_n	vrd	01001	fmt

Syntax:

LUW	vrd[n], base	//n=0~31(SIMD1024), 0~15(SIMD512), 0~07(SIMD256), fmt_n=nnnnn, fmt=0
LUD	vrd[n], base	//n=0~15(SIMD1024), 0~07(SIMD512), 0~03(SIMD256), fmt_n=nnnn0, fmt=1
LUQ	vrd[n], base	//n=0~07(SIMD1024), 0~03(SIMD512), 0~01(SIMD256), fmt_n=nnn01, fmt=1
LUO	vrd[n], base	//n=0~03(SIMD1024), 0~01(SIMD512), fmt_n=nn011, fmt=1

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous (esize/8)-byte data from the memory location indexed by the address to update the <n>-th esize element in the vrd, the other parts of the vrd are unchanged. After the load operation, the base itself need be updated by adding the esize/8 to its original value.

Operation：

check_cop2_enable()

bit [31:0] addr= GPR[base]

VPR[vrd][esize,n]= mem[esize:addr]

GPR[base]= GPR[base] + esize/8

Exceptions:

RI, CpU, AdEL, TLBL

3.15.2 LA<fmt_n>

Load 1 Element **SPECIAL2**

Bit	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
	011100	base	offset	fmt_n	vrd	01000	fmt

Syntax:

LAW	vrd[n], sft_off(base)	//n=0~31(SIMD1024), 0~15(SIMD512), 0~07(SIMD256), fmt_n=nnnnn, fmt=0
		//sft_off=offset<<2;
LAD	vrd[n], sft_off(base)	//n=0~15(SIMD1024), 0~07(SIMD512), 0~03(SIMD256), fmt_n=nnnn0, fmt=1
		//sft_off=offset<<3;
LAQ	vrd[n], sft_off(base)	//n=0~07(SIMD1024), 0~03(SIMD512), 0~01(SIMD256), fmt_n=nnn01, fmt=1
		//sft_off=offset<<4;
LAO	vrd[n], sft_off(base)	//n=0~03(SIMD1024), 0~01(SIMD512), fmt_n=nn011, fmt=1
		//sft_off=offset<<5;

Description：

Add the GPR value of the base and the shifted result of the original bit-field of offset: sft_off (shift number is determined by the esize) to form an effective address with no alignment restriction, load continuous (esize/8)-byte data from the memory location indexed by the address to update the <n>-th esize element in the vrd, the other parts of the vrd are unchanged.

Operation：

check_cop2_enable()

bit [31:0] addr= GPR[base] + sign_extend32(sft_off)

VPR[vrd][esize,n]= mem[esize:addr]

Exceptions:

RI, CpU, AdEL, TLBL

3.15.3 SU<fmt_n>

Store 1 Element and Update base **SPECIAL2**

Bit	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
	011100	base	00000	vrp	fmt_n	01011	fmt

Syntax:

SUW	vrp[n], base	//n=0~31(SIMD1024), 0~15(SIMD512), 0~07(SIMD256), fmt_n=nnnnn, fmt=0
SUD	vrp[n], base	//n=0~15(SIMD1024), 0~07(SIMD512), 0~03(SIMD256), fmt_n=nnnn0, fmt=1
SUQ	vrp[n], base	//n=0~07(SIMD1024), 0~03(SIMD512), 0~01(SIMD256), fmt_n=nnn01, fmt=1
SUO	vrp[n], base	//n=0~03(SIMD1024), 0~01(SIMD512), fmt_n=nn011, fmt=1

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, store the <n>-th esize element in the vrp to the memory location indexed by the address. After the store operation, the base itself need be updated by adding the esize/8 to its original value.

Operation：

check_cop2_enable()

bit [31:0] addr= GPR[base]

mem[esize:addr] =VPR[vrp][esize,n]

GPR[base]= GPR[base] + esize/8

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.4 SA<fmt_n>

Store 1 Element **SPECIAL2**

Bit	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
	011100	base	offset	vrp	fmt_n	01010	fmt

Syntax:

SAW	vrp[n], sft_off(base)	//n=0~31(SIMD1024), 0~15(SIMD512), 0~07(SIMD256), fmt_n=nnnnn, fmt=0
		//sft_off=offset<<2;
SAD	vrp[n], sft_off(base)	//n=0~15(SIMD1024), 0~07(SIMD512), 0~03(SIMD256), fmt_n=nnnn0, fmt=1
		//sft_off=offset<<3;
SAQ	vrp[n], sft_off(base)	//n=0~07(SIMD1024), 0~03(SIMD512), 0~01(SIMD256), fmt_n=nnn01, fmt=1
		//sft_off=offset<<4;
SAO	vrp[n], sft_off(base)	//n=0~03(SIMD1024), 0~01(SIMD512), fmt_n=nn011, fmt=1
		//sft_off=offset<<5;

Description：

Add the GPR value of the base and the shifted result of the original bit-field of offset: sft_off (shift number is determined by the esize) to form an effective address with no alignment restriction, store the <n>-th esize element in the vrp to the memory location indexed by the address.

Operation：

check_cop2_enable()

bit [31:0] addr= GPR[base] + sign_extend32(sft_off)

mem[esize:addr] =VPR[vrp][esize,n]

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.5 LUO<x><fmt>

Load-scatter Octuple-word and Update base **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	0111	vrd	011011

Syntax:

LUO2B	vrd[n], p, base	//fmt=3'B000, xe=0(x=2), <n>=0~1, <p>=0~1
LUO2H	vrd[n], p, base	//fmt=3'B001, xe=0(x=2), <n>=0~1, <p>=0~1
LUO2W	vrd[n], p, base	//fmt=3'B010, xe=0(x=2), <n>=0~1, <p>=0~1
LUO2D	vrd[n], p, base	//fmt=3'B011, xe=0(x=2), <n>=0~1, <p>=0~1
LUO2Q	vrd[n], p, base	//fmt=3'B100, xe=0(x=2), <n>=0~1, <p>=0~1
LUO4B	vrd[n], p, base	//fmt=3'B000, xe=1(x=4), <n>=0, <p>=0~3
LUO4H	vrd[n], p, base	//fmt=3'B001, xe=1(x=4), <n>=0, <p>=0~3
LUO4W	vrd[n], p, base	//fmt=3'B010, xe=1(x=4), <n>=0, <p>=0~3
LUO4D	vrd[n], p, base	//fmt=3'B011, xe=1(x=4), <n>=0, <p>=0~3
LUO4Q	vrd[n], p, base	//fmt=3'B100, xe=1(x=4), <n>=0, <p>=0~3

NOTE:

<n> and <p> are determined by not only the bit field <n_p> but also the bit field <xe>, that is, when <xe>=0, <n> is equal to <n_p>.bit1 and <p> is equal to <n_p>.bit0; when <xe>=1, <n> is fixed zero and <p> is equal to <n_p>.

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 32-byte data from the memory location indexed by the address, the 32-byte data then is divided equally to 256/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged, and after the load operation, the base itself need be updated by adding the 256/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [255:0] data= mem[256:addr]
for i in 256/esize
    VPR[vrd][esize, p + i*x + x*n*(256/esize)]= data[esize,i]
GPR[base]= GPR[base] + 256/8
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.6 LAO<x><fmt>

Load-scatter Octuple-word **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	0111	vrd	011001

Syntax:

LAO2B	vrd[n], p, base	//fmt=3'B000, xe=0(x=2), <n>=0~1, <p>=0~1
LAO2H	vrd[n], p, base	//fmt=3'B001, xe=0(x=2), <n>=0~1, <p>=0~1
LAO2W	vrd[n], p, base	//fmt=3'B010, xe=0(x=2), <n>=0~1, <p>=0~1
LAO2D	vrd[n], p, base	//fmt=3'B011, xe=0(x=2), <n>=0~1, <p>=0~1
LAO2Q	vrd[n], p, base	//fmt=3'B100, xe=0(x=2), <n>=0~1, <p>=0~1
LAO4B	vrd[n], p, base	//fmt=3'B000, xe=1(x=4), <n>=0, <p>=0~3
LAO4H	vrd[n], p, base	//fmt=3'B001, xe=1(x=4), <n>=0, <p>=0~3
LAO4W	vrd[n], p, base	//fmt=3'B010, xe=1(x=4), <n>=0, <p>=0~3
LAO4D	vrd[n], p, base	//fmt=3'B011, xe=1(x=4), <n>=0, <p>=0~3
LAO4Q	vrd[n], p, base	//fmt=3'B100, xe=1(x=4), <n>=0, <p>=0~3

NOTE:

<n> and <p> are determined by not only the bit field <n_p> but also the bit field <xe>, that is, when <xe>=0, <n> is equal to <n_p>.bit1 and <p> is equal to <n_p>.bit0; when <xe>=1, <n> is fixed zero and <p> is equal to <n_p>.

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 32-byte data from the memory location indexed by the address, the 32-byte data then is divided equally to 256/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [255:0] data= mem[256:addr]
for i in 256/esize
    VPR[vrd][esize, p + i*x + x*n*(256/esize)]= data[esize,i]
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.7 SUO<x><fmt>

Store-gather Octuple-word and Update base **SPECIAL2**

Bit	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
	011100	base	fmt	xe	00	vrp	n_p	011	011111

Syntax:

SUO2W	vrp[n], p, base	//fmt=2'B00, xe=0(x=2), <n>=0~1, <p>=0~1
SUO2D	vrp[n], p, base	//fmt=2'B01, xe=0(x=2), <n>=0~1, <p>=0~1
SUO2Q	vrp[n], p, base	//fmt=2'B10, xe=0(x=2), <n>=0~1, <p>=0~1
SUO4W	vrp[n], p, base	//fmt=2'B00, xe=1(x=4), <n>=0, <p>=0~3
SUO4D	vrp[n], p, base	//fmt=2'B01, xe=1(x=4), <n>=0, <p>=0~3
SUO4Q	vrp[n], p, base	//fmt=2'B10, xe=1(x=4), <n>=0, <p>=0~3

NOTE:

<n> and <p> are determined by not only the bit field <n_p> but also the bit field <xe>, that is, when <xe>=0, <n> is equal to <n_p>.bit1 and <p> is equal to <n_p>.bit0; when <xe>=1, <n> is fixed zero and <p> is equal to <n_p>.

Description：

Pick every esize element with a corresponding specific location in the vrp, gather them with total amount of 256/esize to form a 32-byte data. Please refer to the following operation for the calculation rule of the location. In addtion, use the GPR value of the base to form an effective address with no alignment restriction, then store the 32-byte data to the memory location indexed by the address. After the store operation, the base itself need be updated by adding the 256/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [255:0] data
for i in 256/esize
    data[esize,i]= VPR[vrp][esize, p + i*x + x*n*(256/esize)]
mem[256:addr]= data
GPR[base]= GPR[base] + 256/8
```

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.8 SAO<x><fmt>

Store-gather Octuple-word **SPECIAL2**

Bit	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
	011100	base	fmt	xe	00	vrp	n_p	011	011101

Syntax:

SAO2W	vrp[n], p, base	//fmt=2'B00, xe=0(x=2), <n>=0~1, <p>=0~1
SAO2D	vrp[n], p, base	//fmt=2'B01, xe=0(x=2), <n>=0~1, <p>=0~1
SAO2Q	vrp[n], p, base	//fmt=2'B10, xe=0(x=2), <n>=0~1, <p>=0~1
SAO4W	vrp[n], p, base	//fmt=2'B00, xe=1(x=4), <n>=0, <p>=0~3
SAO4D	vrp[n], p, base	//fmt=2'B01, xe=1(x=4), <n>=0, <p>=0~3
SAO4Q	vrp[n], p, base	//fmt=2'B10, xe=1(x=4), <n>=0, <p>=0~3

NOTE:

<n> and <p> are determined by not only the bit field <n_p> but also the bit field <xe>, that is, when <xe>=0, <n> is equal to <n_p>.bit1 and <p> is equal to <n_p>.bit0; when <xe>=1, <n> is fixed zero and <p> is equal to <n_p>.

Description：

Pick every esize element with a corresponding specific location in the vrp, gather them with total amount of 256/esize to form a 32-byte data. Please refer to the following operation for the calculation rule of the location. In addtion, use the GPR value of the base to form an effective address with no alignment restriction, then store the 32-byte data to the memory location indexed by the address.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [255:0] data
for i in 256/esize
    data[esize,i]= VPR[vrp][esize, p + i*x + x*n*(256/esize)]
mem[256:addr]= data
```

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.9 LUW<x><fmt>

Load-scatter a Word and Update base **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	0 vrd	011011

Syntax:

LUW2B	vrd[n], p, base	//fmt=0, xe= 3'B000(x=2), n=0~15, p=0~1, n_p=nnnnp
LUW2H	vrd[n], p, base	//fmt=1, xe= 3'B000(x=2), n=0~15, p=0~1, n_p=nnnnp
LUW4B	vrd[n], p, base	//fmt=0, xe= 3'B001(x=4), n=0~7, p=0~3, n_p=nnnpp
LUW4H	vrd[n], p, base	//fmt=1, xe= 3'B001(x=4), n=0~7, p=0~3, n_p=nnnpp
LUW8B	vrd[n], p, base	//fmt=0, xe= 3'B010(x=8), n=0~3, p=0~7, n_p=nnppp
LUW8H	vrd[n], p, base	//fmt=1, xe= 3'B010(x=8), n=0~3, p=0~7, n_p=nnppp
LUW16B	vrd[n], p, base	//fmt=0, xe= 3'B011(x=16), n=0~1, p=0~15, n_p=npppp
LUW16H	vrd[n], p, base	//fmt=1, xe= 3'B011(x=16), n=0~1, p=0~15, n_p=npppp
LUW32B	vrd[n], p, base	//fmt=0, xe= 3'B100(x=32), n=0, p=0~31, n_p=ppppp
LUW32H	vrd[n], p, base	//fmt=1, xe= 3'B100(x=32), n=0, p=0~31, n_p=ppppp

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 4-byte data from the memory location indexed by the address, the 4-byte data then is divided equally to 32/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged, and after the load operation, the base itself need be updated by adding the 32/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [31:0] data= mem[32:addr]
for i in 32/esize
    VPR[vrd][esize, p + i*x + x*n*(32/esize)]= data[esize,i]
GPR[base]= GPR[base] + 32/8
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.10 LAW<x><fmt>

Load-scatter a Word **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	0	vrd	011001

Syntax:

LAW2B	vrd[n], p, base	//fmt=0, xe= 3'B000(x=2), n=0~15, p=0~1, n_p=nnnnp
LAW2H	vrd[n], p, base	//fmt=1, xe= 3'B000(x=2), n=0~15, p=0~1, n_p=nnnnp
LAW4B	vrd[n], p, base	//fmt=0, xe= 3'B001(x=4), n=0~7, p=0~3, n_p=nnnpp
LAW4H	vrd[n], p, base	//fmt=1, xe= 3'B001(x=4), n=0~7, p=0~3, n_p=nnnpp
LAW8B	vrd[n], p, base	//fmt=0, xe= 3'B010(x=8), n=0~3, p=0~7, n_p=nnppp
LAW8H	vrd[n], p, base	//fmt=1, xe= 3'B010(x=8), n=0~3, p=0~7, n_p=nnppp
LAW16B	vrd[n], p, base	//fmt=0, xe= 3'B011(x=16), n=0~1, p=0~15, n_p=npppp
LAW16H	vrd[n], p, base	//fmt=1, xe= 3'B011(x=16), n=0~1, p=0~15, n_p=npppp
LAW32B	vrd[n], p, base	//fmt=0, xe= 3'B100(x=32), n=0, p=0~31, n_p=ppppp
LAW32H	vrd[n], p, base	//fmt=1, xe= 3'B100(x=32), n=0, p=0~31, n_p=ppppp

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 4-byte data from the memory location indexed by the address, the 4-byte data then is divided equally to 32/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [31:0] data= mem[32:addr]
for i in 32/esize
    VPR[vrd][esize, p + i*x + x*n*(32/esize)]= data[esize,i]
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.11 LUD<x><fmt>

Load-scatter Double-word and Update base **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	01	vrd	011011

Syntax:

LUD2B	vrd[n], p, base	//fmt=2'B00, xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
LUD2H	vrd[n], p, base	//fmt=2'B01, xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
LUD2W	vrd[n], p, base	//fmt=2'B10, xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
LUD4B	vrd[n], p, base	//fmt=2'B00, xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
LUD4H	vrd[n], p, base	//fmt=2'B01, xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
LUD4W	vrd[n], p, base	//fmt=2'B10, xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
LUD8B	vrd[n], p, base	//fmt=2'B00, xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
LUD8H	vrd[n], p, base	//fmt=2'B01, xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
LUD8W	vrd[n], p, base	//fmt=2'B10, xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
LUD16B	vrd[n], p, base	//fmt=2'B00, xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp
LUD16H	vrd[n], p, base	//fmt=2'B01, xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp
LUD16W	vrd[n], p, base	//fmt=2'B10, xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 8-byte data from the memory location indexed by the address, the 8-byte data then is divided equally to 64/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged, and after the load operation, the base itself need be updated by adding the 64/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [63:0] data= mem[64:addr]
for i in 64/esize
    VPR[vrd][esize, p + i*x + x*n*(64/esize)]= data[esize,i]
GPR[base]= GPR[base] + 64/8
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.12 LAD<x><fmt>

Load-scatter Double-word **SPECIAL2**

Bit	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
	011100	base	fmt xe	n_p	01	vrd	011001

Syntax:

LAD2B	vrd[n], p, base	//fmt=2'B00, xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
LAD2H	vrd[n], p, base	//fmt=2'B01, xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
LAD2W	vrd[n], p, base	//fmt=2'B10, xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
LAD4B	vrd[n], p, base	//fmt=2'B00, xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
LAD4H	vrd[n], p, base	//fmt=2'B01, xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
LAD4W	vrd[n], p, base	//fmt=2'B10, xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
LAD8B	vrd[n], p, base	//fmt=2'B00, xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
LAD8H	vrd[n], p, base	//fmt=2'B01, xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
LAD8W	vrd[n], p, base	//fmt=2'B10, xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
LAD16B	vrd[n], p, base	//fmt=2'B00, xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp
LAD16H	vrd[n], p, base	//fmt=2'B01, xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp
LAD16W	vrd[n], p, base	//fmt=2'B10, xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 8-byte data from the memory location indexed by the address, the 8-byte data then is divided equally to 64/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [63:0] data= mem[64:addr]
for i in 64/esize
    VPR[vrd][esize, p + i*x + x*n*(64/esize)]= data[esize,i]
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.13 SUD<x><fmt>

Store-gather Double-word and Update base **SPECIAL2**

Bit	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
	011100	base	xe	000	vrp	n_p		0	011111

Syntax:

SUD2W	vrp[n], p, base	//xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
SUD4W	vrp[n], p, base	//xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
SUD8W	vrp[n], p, base	//xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
SUD16W	vrp[n], p, base	//xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp

Description：

Pick every esize element with a corresponding specific location in the vrp, gather them with total amount of 64/esize to form a 8-byte data. Please refer to the following operation for the calculation rule of the location. In addtion, use the GPR value of the base to form an effective address with no alignment restriction, then store the 8-byte data to the memory location indexed by the address. After the store operation, the base itself need be updated by adding the 64/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [63:0] data
for i in 64/esize
    data[esize,i]= VPR[vrp][esize, p + i*x + x*n*(64/esize)]
mem[64:addr]= data
GPR[base]= GPR[base] + 64/8
```

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.14 SAD<x><fmt>

Store-gather Double-word **SPECIAL2**

Bit	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
	011100	base	xe	000	vrp	n_p	0	011101

Syntax:

SAD2W	vrp[n], p, base	//xe= 2'B00(x=2), n=0~7, p=0~1, n_p=nnnp
SAD4W	vrp[n], p, base	//xe= 2'B01(x=4), n=0~3, p=0~3, n_p=nnpp
SAD8W	vrp[n], p, base	//xe= 2'B10(x=8), n=0~1, p=0~7, n_p=nppp
SAD16W	vrp[n], p, base	//xe= 2'B11(x=16), n=0, p=0~15, n_p=pppp

Description：

Pick every esize element with a corresponding specific location in the vrp, gather them with total amount of 64/esize to form a 8-byte data. Please refer to the following operation for the calculation rule of the location. In addtion, use the GPR value of the base to form an effective address with no alignment restriction, then store the 8-byte data to the memory location indexed by the address.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [63:0] data
for i in 64/esize
    data[esize,i]= VPR[vrp][esize, p + i*x + x*n*(64/esize)]
mem[64:addr]= data
```

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.15 LUQ<x><fmt>

Load-scatter Quadruple-word and Update base **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	011	vrd	011011

Syntax:

LUQ2B	vrd[n], p, base	//fmt=2'B00, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LUQ2H	vrd[n], p, base	//fmt=2'B01, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LUQ2W	vrd[n], p, base	//fmt=2'B10, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LUQ2D	vrd[n], p, base	//fmt=2'B11, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LUQ4B	vrd[n], p, base	//fmt=2'B00, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LUQ4H	vrd[n], p, base	//fmt=2'B01, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LUQ4W	vrd[n], p, base	//fmt=2'B10, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LUQ4D	vrd[n], p, base	//fmt=2'B11, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LUQ8B	vrd[n], p, base	//fmt=2'B00, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
LUQ8H	vrd[n], p, base	//fmt=2'B01, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
LUQ8W	vrd[n], p, base	//fmt=2'B10, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
LUQ8D	vrd[n], p, base	//fmt=2'B11, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 16-byte data from the memory location indexed by the address, the 16-byte data then is divided equally to 128/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged, and after the load operation, the base itself need be updated by adding the 128/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [127:0] data= mem[128:addr]
for i in 128/esize
    VPR[vrd][esize, p + i*x + x*n*(128/esize)]= data[esize,i]
GPR[base]= GPR[base] + 128/8
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.16 LAQ<x><fmt>

Load-scatter Quadruple-word **SPECIAL2**

Bit	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
	011100	base	fmt	xe	n_p	011	vrd	011001

Syntax:

LAQ2B	vrd[n], p, base	//fmt=2'B00, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LAQ2H	vrd[n], p, base	//fmt=2'B01, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LAQ2W	vrd[n], p, base	//fmt=2'B10, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LAQ2D	vrd[n], p, base	//fmt=2'B11, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
LAQ4B	vrd[n], p, base	//fmt=2'B00, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LAQ4H	vrd[n], p, base	//fmt=2'B01, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LAQ4W	vrd[n], p, base	//fmt=2'B10, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LAQ4D	vrd[n], p, base	//fmt=2'B11, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
LAQ8B	vrd[n], p, base	//fmt=2'B00, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
LAQ8H	vrd[n], p, base	//fmt=2'B01, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
LAQ8W	vrd[n], p, base	//fmt=2'B10, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
LAQ8D	vrd[n], p, base	//fmt=2'B11, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp

Description：

Use the GPR value of the base to form an effective address with no alignment restriction, load continuous 16-byte data from the memory location indexed by the address, the 16-byte data then is divided equally to 128/esize elements. For each esize element generated by equipartition, there is a corresponding specific location in the vrd that the element should be updated to. Please refer to the following operation for the calculation rule of the location. Moreover, the other parts of the vrd keep unchanged.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [127:0] data= mem[128:addr]
for i in 128/esize
    VPR[vrd][esize, p + i*x + x*n*(128/esize)]= data[esize,i]
```

Exceptions:

RI, CpU, AdEL, TLBL

3.15.17 SUQ<x><fmt>

Store-gather Quadruple-word and Update base **SPECIAL2**

Bit	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
	011100	base	fmt	xe	00	vrp	n_p	01	011111

Syntax:

SUQ2W	vrp[n], p, base	//fmt=1'B0, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
SUQ2D	vrp[n], p, base	//fmt=1'B1, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
SUQ4W	vrp[n], p, base	//fmt=1'B0, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
SUQ4D	vrp[n], p, base	//fmt=1'B1, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
SUQ8W	vrp[n], p, base	//fmt=1'B0, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
SUQ8D	vrp[n], p, base	//fmt=1'B1, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp

Description：

Pick every esize element with a corresponding specific location in the vrp, gather them with total amount of 128/esize to form a 16-byte data. Please refer to the following operation for the calculation rule of the location. In addtion, use the GPR value of the base to form an effective address with no alignment restriction, then store the 16-byte data to the memory location indexed by the address. After the store operation, the base itself need be updated by adding the 128/8 to its original value.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [127:0] data
for i in 128/esize
    data[esize,i]= VPR[vrp][esize, p + i*x + x*n*(128/esize)]
mem[128:addr]= data
GPR[base]= GPR[base] + 128/8
```

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.15.18 SAQ<x><fmt>

Store-gather Quadruple-word **SPECIAL2**

Bit	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
	011100	base	fmt	xe	00	vrp	n_p	01	011101

Syntax:

SAQ2W	vrp[n], p, base	//fmt=1'B0, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
SAQ2D	vrp[n], p, base	//fmt=1'B1, xe= 2'B00(x=2), n=0~3, p=0~1, n_p=nnp
SAQ4W	vrp[n], p, base	//fmt=1'B0, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
SAQ4D	vrp[n], p, base	//fmt=1'B1, xe= 2'B01(x=4), n=0~1, p=0~3, n_p=npp
SAQ8W	vrp[n], p, base	//fmt=1'B0, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp
SAQ8D	vrp[n], p, base	//fmt=1'B1, xe= 2'B10(x=8), n=0, p=0~7, n_p=ppp

Description：

Pick every esize element with a corresponding specific location in the vrp, gather them with total amount of 128/esize to form a 16-byte data. Please refer to the following operation for the calculation rule of the location. In addtion, use the GPR value of the base to form an effective address with no alignment restriction, then store the 16-byte data to the memory location indexed by the address.

Operation：

```
check_cop2_enable()
bit [31:0] addr= GPR[base]
bit [127:0] data
for i in 128/esize
    data[esize,i]= VPR[vrp][esize, p + i*x + x*n*(128/esize)]
mem[128:addr]= data
```

Exceptions:

RI, CpU, Modify, AdES, TLBS

3.16 Neural Networks Accelerate

NNA is a neural networks accelerate unit, it is belonging to MXU3. Please refer to the "NNA manual" for more details about NNA.

3.16.1 NNRWR

NNA Registers Write **SPECIAL2**

Bit	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
	011100						00000					imm[9:5]					vwrp					imm[4:0]						111010			

Syntax:

NNRWR vwrp,imm

Description：

NNA registrs write.

Operation：

check_cop2_enable()

NNA_registers_write(VWR[vwrp], imm)

Exceptions:

RI, CpU

3.16.2 NNRRD

NNA Registers Read **SPECIAL2**

Bit	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
	011100	00001	imm	vrd	111010

Syntax:

NNRRD vrd, imm

Description：

NNA registrs read, the data will be written to VPR.

Operation：

check_cop2_enable()

bit [MLEN-1:0] nna_reg

nna_reg = NNA_registers_read(imm)

VPR[vrd] = nna_reg

Exceptions:

RI, CpU

3.16.3 NNDWR

NNA Data Write **SPECIAL2**

Bit	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
	011100	00010	imm[9:5]	vrp	imm[4:0]	111010

Syntax:

NNDWR vrp, imm

Description：

NNA data write.

Operation：

check_cop2_enable()

NNA_data_write(VPR[vrp], imm)

Exceptions:

RI, CpU

3.16.4 NNDRD

NNA Data Read **SPECIAL2**

Bit	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
	011100	00011	imm	vrd	111010

Syntax:

NNDRD vrd,imm

Description：

NNA data read.

Operation：

check_cop2_enable()

bit [MLEN-1:0] nna_data

nna_data = NNA_data_read(imm)

VPR[vrd] = nna_data

Exceptions:

RI, CpU

3.16.5 NNCMD

NNA Command **SPECIAL2**

Bit	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
	011100	00101	imm	111010

Syntax:

NNCMD imm

Description：

Send command to NNA.

Operation：

check_cop2_enable()

NNA_cmd(imm)

Exceptions:

RI, CpU

3.16.6 NNMAC

NNA Multiply-accumulate **SPECIAL2**

Bit	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
	011100	00100	imm[9:5]	vwrp	imm[4:0]	111010

Syntax:

NNMAC vwrp,imm

Description：

NNA multiply-accmulate.

Operation：

check_cop2_enable()

NNA_mac(VWR[vwrp], imm)

Exceptions:

RI, CpU

Appendix A

A.1 Instruction Formats

- COP2 format

COP2

Bit	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
	COP2						Minor opcode					vrp					vrs					vrd					funct					

- SPECIAL2 format

SPECIAL2

Bit	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
	SPECIAL2						rs					rt					rd					funct					Minor opcode					

A.2 Instruction Bit Encoding

This chapter describe the bit encoding tables used for MXU3. These tables only list the instruction encoding for the MXU3 instruction.

A instruction's encoding is found at the below tables.

- MXU3 Encoding of the Opcode Field

opcode	***bits[28:26]***							
bits[31:29]	000	001	010	011	100	101	110	111
000								
001								
010			COP2					
011					SPECILA2			
100								
101								
110								
111								

- Encoding of COP2 Field for MXU3 Instruction Formats

COP2	*bits[23:21]*							
bits[25:24]	000	001	010	011	100	101	110	111
00								
01								
10	ABS,MM,CMP	SH	LIWR	*,BW,EQ,Reg,Branch,FCADD,FXAS	+-	SH		SRSUM
11	LIH	LIW	LIWH	*,BW,ADDIW,ADDRW,FCMUL	SRMAC,SMAC	SRMAC,SMAC	SRMAC,SMAC	SRMAC,SMAC

- Encoding of SPECIAL2 Field for MXU3 Instruction Formats

fmt,funct0	*bits[2:0]*							
bits[5:3]	000	001	010	011	100	101	110	111
000				Reg,Pref				
001					PMAPx	PMAPx	PMAPx	PMAPx
010	LAx	LAx	LUx	LUx	SAx	SAx	SUx	SUx
011		LAx-S		LUx-S		SAx-G		SUx-G
100								
101				GSHUFB			Reg,fp,cvt	SAT
110	REPI	EXTx	CMVW	SHUF	GT-bit	GT-W+	MOVW	MOVx
111	ILV{E|O}	BSHxx	NNA	GSHUFx				

Revision History

Revision	Date/Author	Description
01.00	January,1,2019 Jin	● No technical content changes ● Change the name MXU into MXU3
01.10	Febrary,13,2019 General	● Remove PREFL1C and PREFL2C, adjust some description for better understanding
01.20	April,15,2019 tyliu	● Adjust some Load/Store instructions according to compiler team's proposal
01.30	October,11,2019 tyliu	● Add new integer arithmetic and float-pointing instructions ● Modify float-pointing instructions so that no longer use VSR