Ingenic®

SIMD/DSP Instruction Set

Revision: 1.0
Date: Apr. 2006

北京君正集成电路有限公司
Ingenic Semiconductor Co. Ltd

Ingenic SIMD/DSP Instruction Set

Release history

Date	Revision	Change
Apr. 2006	1.0	Release-1 (Jz4740 only supports this version)
Oct.2007	1.1	Release-2

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 Ingenicfor the current documentation and errata.

Ingenic Semiconductor Co. Ltd

22th Floor, Building A, Cyber Tower, No.2, Zhong Guan Cun South Avenue
Haidian District, Beijing 100086, China
Tel: 86-10-82511297
Fax: 86-10-82511589
Http: //www.ingenic.cn

Content

This document depicts the JZ SIMD/DSP instructions which serving for MXU – Media Extension Unit. Except Jz4730 MXU is a basic component in Jz47xx series belonging to Xburst uA.

1.1 Extended registers for SIMD/DSP Instruction Set

1.1.1 General purpose registers

In MXU, a dedicated register file named XRF comprises sixteen 32-bit general purpose registers – XR0~XR15. XR0 is a special one, which always is read as zero. Moreover, XR16 is an alias of MXU_CR described below and it can only be accessed by S32I2M/S32M2I instructions.

1.1.2 MXU control register (MXU_CR)

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
Name	LC	RC																												BIAS	RD_EN	MXUEN
RST	0	0																												0	0	0
R/W																																
Note		Reserved								Read/Write								Read Only								Write Only						

- MXUEN: Flag of MXU enable. 1 – enable. 0 - disable
- RD_EN: Flag of round enable, 1 – round enable, 0 – round disable. Effective for **D16MULF, D16MUL, D16MACF** and **D16MACE** instructions.
- BIAS: Biased rounding and un-biased rounding toggle. 0 – un-bias, 1 – bias.
- RC: Carry out of right lane's adder
- LC: Carry out of left lane's adder

Note that only when MXU is enabled (default is disable), executing SIMD/DSP instructions is valid, otherwise, the result is unpredictable. However, **S32M2I/S32I2M** can be executed without any constraint.

Moreover, when MXUEN is toggled from disable to enable by executing **S32I2M**, there are at least three non-SIMD/DSP instructions necessary to serve as isolation between the **S32I2M** instruction and following available SIMD/DSP instructions. However, when MXUEN is disabled, following SIMD/DSP instructions' being executed will be nullified immediately.

1.1.3 Rounding Operation

Rounding result is an option for 16x16 multiply and accumulation operations. The upper portion (bits 31~16) of the accumulator is rounded according to the contents of the lower portion (bits 15~0) of the accumulator. MXU supports two rounding modes: biased rounding and un-biased rounding.

- **Un-biased rounding (convergent rounding)**

Un-biased rounding (also called round-to-nearest even number) is the default-rounding mode. The traditional rounding method rounds up any lower portion value greater than or equal to the midway point (0x8000) and rounds down any value less than the point. Thus the final results are eventually biased to greater-than-zero direction. Convergent rounding solves the pitfall by rounding down if the number (bit 16) is even (= 0) and rounding up if the number (bit 16) is odd (= 1) for midway point.

- **Biased rounding**

When biased rounding is selected by setting BIAS bit field in the MXU_CR, all lower portion values greater than or equal to the midway point are rounded up and all values less than the midway point are rounded down. Therefore, a small positive bias is introduced.

1.1.4 Fraction multiplication

When execute the **D16MULF/D16MULE** or **D16MACF/D16MACE**, MXU takes it for granted that the multiplier operates in fraction mode (Q15 format). Before updating the destination registers, the rounding operation will be done or not indicated by MXU_CR.RD_EN.

1.2 Instruction format

1.2.1 Instruction assembly mnemonic

The SIMD/DSP instruction assembly mnemonic format is as follow

<Operation parallel level><Operand size><Operation abbr.>[<Suffix>]

Operation parallel level: S (single), D (dual) and Q (Quad)
Operand size (bit): 31, 16 and 8
Operation abbr.: LDD, LDI, STD, SDI, MUL, MAC, MAD, ADD, ACC, CPS, AVG, ABD, SLT, SAD, SLL, SLR, SAR, MAX, MIN, M2I, I2M, SAT, ALN, SFL
Suffix (optional): V, L, F, R, E and W

1.2.1.1 Operation abbreviation

Abbr	Meaning
ADD	Add or subtract
ADDC	Add with carry-in
ACC	Add and accumulate
ASUM	Sum together to accumulate (add or subtract) *1
ASUMC	Sum with carry-in together to accumulate (add or subtract) *1
AVG	Average between 2 operands
ABD	Absolute difference
ALN	Align data
AND	a & b *1
CPS	Copy sign, used to get absolute value
EXTR	Extract specific bits from MSB position *1
I2M	Move from IU to MXU
LDD	Load data from memory to XRF
LDI	Load data from memory to XRF and increase the address base register
LUI	Load unsigned immediate *1
MUL	Multiply
MULU	Unsigned multiply *1
MADD	64-bit operand add 32x32 product *1
MSUB	64-bit operand subtract 32x32 product *1
MAC	Multiply and accumulate (add or subtract)
MAD	Multiply and add or subtract
MAX	Maximum between 2 operands
MIN	Minimum between 2 operands
M2I	Move from MXU to IU
MOVZ	Move if zero *1
MOVN	Move if non-zero *1
NOR	~ (a | b) *1

OR	a | b [*1]
STD	Store data from XRF to memory
SDI	Store data from XRF to memory and increase the address base register
SLT	Set of less than comparison
SAD	Sum of absolute differences
SLL	Shift logic left
SLR	Shift logic right
SAR	Shift arithmetic right
SAT	Saturation
SFL	Shuffle
SCOP	Calculate x's scope (-1: means x<0, 0: means x==0, 1: means x>0) [*1]
XOR	a ^ b [*1]

[*1] : These operations belong to SIMD/DSP release-2.

1.2.1.2 Suffix abbreviation

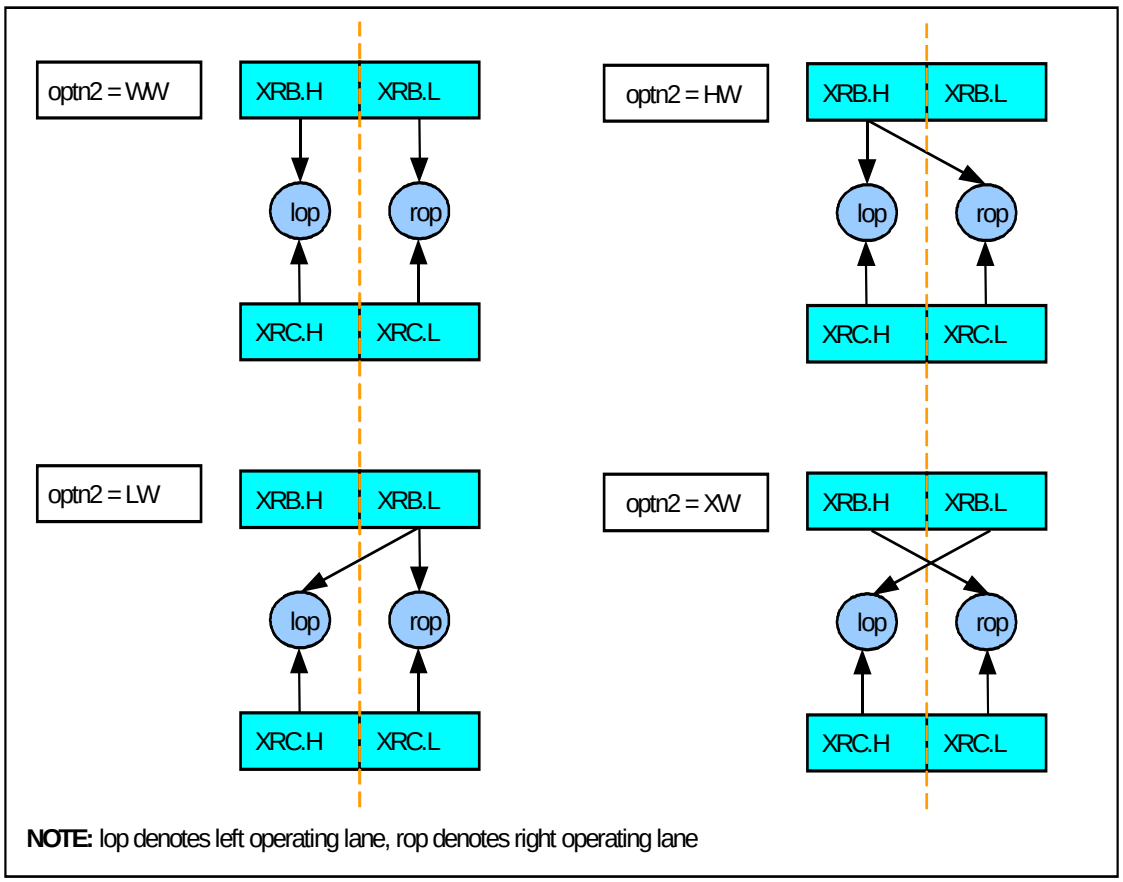
Abbr	Meaning
V	Variable instead of immediate
L	Low part result
W	Combination of L and V
F	Fixed point multiplication
R	Rounding
E	Expanded results

1.2.2 Instruction operands

1.2.2.1 Some immediate number abbreviation

Abbr	Meaning
STRD2	Stride of the address index register
OPTN1,OPTN2	Operand getting pattern
APTN1, APTN2	Accumulate pattern, how to do the accumulate, add or subtract
EPTN2	Execute pattern, how to do the execution, add or subtract

1.2.3 Pattern of 16bit Add/Multiply



Above diagram shows the four cases of **pattern_MUL** or **pattern_ADD**.

1.3 Instruction summary

Class	Mnemonic	Operands
Load & Store (29)	S32LDD	xra, rb, s12
	S32STD	xra, rb, s12
	S32LDDV	xra, rb, rc, strd2
	S32STDV	xra, rb, rc, strd2
	S32LDI	xra, rb, s12
	S32SDI	xra, rb, s12
	S32LDIV	xra, rb, rc, strd2
	S32SDIV	xra, rb, rc, strd2
	S32LDDR	xra, rb, s12 *1
	S32STDR	xra, rb, s12 *1
	S32LDDVR	xra, rb, rc, strd2 *1
	S32STDVR	xra, rb, rc, strd2 *1
	S32LDIR	xra, rb, s12 *1
	S32SDIR	xra, rb, s12 *1
	S32LDIVR	xra, rb, rc, strd2 *1
	S32SDIVR	xra, rb, rc, strd2 *1
	S16LDD	xra, rb, s10, eptn2 *1
	S16STD	xra, rb, s10, eptn2 *1
	S16LDI	xra, rb, s10, eptn2 *1
	S16SDI	xra, rb, s10, eptn2 *1
	S8LDD	xra, rb, s8, eptn3 *1
	S8STD	xra, rb, s8, eptn3 *1
	S8LDI	xra, rb, s8, eptn3 *1
	S8SDI	xra, rb, s8, eptn3 *1
	LXW	rd, rs, rt, strd2 *1
	LXH	rd, rs, rt, strd2 *1
	LXHU	rd, rs, rt, strd2 *1
	LXB	rd, rs, rt, strd2 *1
	LXBU	rd, rs, rt, strd2 *1
MUL & MAC (19)	S32MADD	xra, xrd, rs, rt *1
	S32MADDU	xra, xrd, rs, rt *1
	S32SUB	xra, xrd, rs, rt *1
	S32SUBU	xra, xrd, rs, rt *1
	S32MUL	xra, xrd, rs, rt *1
	S32MULU	xra, xrd, rs, rt *1
	D16MUL	xra, xrb, xrc, xrd, optn2

	D16MULE	xra, xrb, xrc, optn2 *1
	D16MULF	xra, xrb, xrc, optn2
	D16MAC	xra, xrb, xrc, xrd, aptn2, optn2
	D16MACE	xra, xrb, xrc, xrd, aptn2, optn2 *1
	D16MACF	xra, xrb, xrc, xrd, aptn2, optn2
	D16MADL	xra, xrb, xrc, xrd, aptn2, optn2
	S16MAD	xra, xrb, xrc, xrd, aptn1, optn1
	Q8MUL	xra, xrb, xrc, xrd
	Q8MULSU	xra, xrb, xrc, xrd *1
	Q8MAC	xra, xrb, xrc, xrd, aptn2
	Q8MACSU	xra, xrb, xrc, xrd, aptn2 *1
	Q8MADL	xra, xrb, xrc, xrd, aptn2
Sum & Difference (22)	D32ADD	xra, xrb, xrc, xrd, eptn2
	D32ADDC	xra, xrb, xrc, xrd *1
	D32ACC	xra, xrb, xrc, xrd, eptn2
	D32ACCM	xra, xrb, xrc, xrd, eptn2 *1
	D32ASUM	xra, xrb, xrc, xrd, eptn2 *1
	S32CPS	xra, xrb, xrc
	Q16ADD	xra, xrb, xrc, xrd, eptn2, optn2
	Q16ACC	xra, xrb, xrc, xrd, eptn2
	Q16ACCM	xra, xrb, xrc, xrd, eptn2 *1
	D16ASUM	xra, xrb, xrc, xrd, eptn2 *1
	D16CPS	xra, xrb, xrc
	D16AVG	xra, xrb, xrc
	D16AVGR	xra, xrb, xrc
	Q8ADD	xra, xrb, xrc, eptn2
	Q8ADDE	xra, xrb, xrc, xrd, eptn2
	Q8ACCE	xra, xrb, xrc, xrd, eptn2
	Q8ABD	xra, xrb, xrc
	Q8SAD	xra, xrb, xrc, xrd
	Q8AVG	xra, xrb, xrc
	Q8AVGR	xra, xrb, xrc
	D8SUM	xra, xrb, xrc, xrd *1
	D8SUMC	xra, xrb, xrc, xrd *1
Shift (14)	D32SLL	xra, xrb, xrc, xrd, sft4
	D32SLR	xra, xrb, xrc, xrd, sft4
	D32SAR	xra, xrb, xrc, xrd, sft4
	D32SARL	xra, xrb, xrc, sft4
	D32SLLV	xra, xrb, rb
	D32SLRV	xra, xrb, rb

	D32SARV	xra, xrb, rb
	D32SARW	xra, xrb, xrc, rb
	Q16SLL	xra, xrb, xrc, xrd, sft4
	Q16SLR	xra, xrb, xrc, xrd, sft4
	Q16SAR	xra, xrb, xrc, xrd, sft4
	Q16SLLV	xra, xrb, rb
	Q16SLRV	xra, xrb, rb
	Q16SARV	xra, xrb, rb
Compare (16)	S32MAX	xra, xrb, xrc
	S32MIN	xra, xrb, xrc
	S32SLT	xra, xrb, xrc *1
	S32MOVZ	xra, xrb, xrc *1
	S32MOVN	xra, xrb, xrc *1
	D16MAX	xra, xrb, xrc
	D16MIN	xra, xrb, xrc
	D16SLT	xra, xrb, xrc *1
	D16MOVZ	xra, xrb, xrc *1
	D16MOVN	xra, xrb, xrc *1
	Q8MAX	xra, xrb, xrc
	Q8MIN	xra, xrb, xrc
	Q8SLT	xra, xrb, xrc
	Q8SLTU	xra, xrb, xrc *1
	Q8MOVZ	xra, xrb, xrc *1
	Q8MOVN	xra, xrb, xrc *1
Bitwise (4)	S32NOR	xra, xrb, xrc *1
	S32AND	xra, xrb, xrc *1
	S32XOR	xra, xrb, xrc *1
	S32OR	xra, xrb, xrc *1
Move (2)	S32M2I	xra, rb
	S32I2M	xra, rb
Misc (8)	S32SFL	xra, xrb, xrc, xrd, optn2
	S32ALN	xra, xrb, xrc, rb
	S32ALNI	xra, xrb, xrc, s3 *1
	S32LUI	xra, s8, optn3 *1
	S32EXTR	xra, xrb, rb, bits5 *1
	S32EXTRV	xra, xrb, rs, rt *1

	Q16SCOP	xra, xrb, xrc, xrd [*1]
	Q16SAT	xra, xrb, xrc

NOTE: Total 114 instructions, and [*1] belong to SIMD/DSP release-2.

1.4 Macro functions for convenience of description

This section defines some useful macro functions that will be frequently referenced in later chapters for instruction description.

- **sign_ext32(x)**
  Sign extend x to a 32-bit signed value.
- **zero_ext32(x)**
  Zero extend x to a 32-bit unsigned value.
- **sign_ext16(x)**
  Sign extend x to a 16-bit signed value.
- **zero_ext16(x)**
  Zero extend x to a 16-bit unsigned value.
- **zero_ext10(x)**
  Zero extend x to a 10-bit unsigned value.
- **zero_ext9(x)**
  Zero extend x to a 9-bit unsigned value.
- **signed(x)**
  x represents a signed value.
- **unsign(x)**
  x represents an unsigned value
- **trunc_8(x)**
  Truncate bits of x except the lower 8 bits.
- **trunc_32(x)**
  Truncate bits of x except the lower 32 bits.
- **usat_8(x)**
  saturate x to value 0 to 255. That is, if x < 0, then the result is 0; if x > 255, then result is 255. Others, result = x.
- **abs(x)**
  Get the absolution of x.
- **(>>L)**
  Logic right shift.
- **(>>A)**
  Arithmetic right shift.

Note that for all 16bit calculations (S16, D16, Q16), signed operation is default.

1.5 Load/Store

1.5.1 S32LDDR/S32STDR

Syntax:

```
S32LDDR XRa, rb, S12
S32STDR XRa, rb, S12
```

Operation:

```
S32LDDR:
tmp32= (Mem32 [rb + S12]);
XRa= {tmp32[7:0], tmp32[15:8], tmp32[23:16], tmp32[31:24]};

S32STDR:
(Mem32 [rb + S12])
= {XRa[7:0], XRa[15:8], XRa[23:16], XRa[31:24]};
```

Description:

S32LDDR: Load a word data from the memory address formed by adding rb with S12, reverse the word's byte sequence and subsequently update XRa with the reversed result.

S32STDR: Store a word to the memory address formed by adding rb with S12, the byte sequence of the word must be reversed first.

NOTE:

1. S12 should be the multiple of 4.
2. The offset range represented by S12 is –2048 ~ 2044.
3. To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDDR    XR1, $1, -4
S32STDR    XR15, v0, 0
```

1.5.2 S32LDD/S32STD

Syntax:

```
S32LDD XRa, rb, S12
S32STD XRa, rb, S12
```

Operation:

```
S32LDD:
XRa= (Mem32 [rb + S12]);

S32STD:
(Mem32 [rb + S12]) = XRa;
```

Description:

S32LDD: Load a word from the address formed by adding rb with S12.

S32STD: Store a word to the address formed by adding rb with S12.

NOTE:

1. S12 should be the multiple of 4.
2. The offset range represented by S12 is –2048 ~ 2044.
3. To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDD  XR11, $5, -4
S32STD  XR3, a1, 2044
```

1.5.3 S32LDDVR/S32STDVR

Syntax:

```
S32LDDVR    XRa, rb, rc, STRD2
S32STDVR    XRa, rb, rc, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
S32LDDVR:
tmp32 = (Mem32 [rb + (rc << STRD2)]);
XRa = {tmp32[7:0], tmp32[15:8], tmp32[23:16], tmp32[31:24]};

S32STDVR:
(Mem32 [rb + (rc << STRD2)])
= {XRa[7:0], XRa[15:8], XRa[23:16], XRa[31:24]};
```

Description:

S32LDDVR: Load a word from the memory address formed by adding rb with rc<<STRD2, reverse the word's byte sequence and subsequently update XRa with the reversed result.

S32STDVR: Store a word to the memory address formed by adding rb with rc<<STRD2, reverse the the word's byte sequence first.

NOTE:

To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDDVR    XR11, $5, v0, 1
S32STDVR    XR3, a1, a2, 0
```

1.5.4 S32LDDV/S32STDV

Syntax:

```
S32LDDV XRa, rb, rc, STRD2
S32STDV XRa, rb, rc, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
S32LDDV:
XRa = (Mem32 [rb + (rc << STRD2)]);

S32STDV:
(Mem32 [rb + (rc << STRD2)]) = XRa;
```

Description:

S32LDDV: Load a word from the memory address formed by adding rb with rc<<STRD2.

S32STDV: Store a word to the memory address formed by adding rb with rc<<STRD2.

NOTE:

To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDDV    XR11, $5, v0, 1
S32STDV    XR3, a1, a2, 0
```

1.5.5 S32LDIR/S32SDIR

Syntax:

```
S32LDIR XRa, rb, S12
S32SDIR XRa, rb, S12
```

Operation:

```
S32LDIR:
tmp32 = (Mem32 [rb + S12]);
XRa = {tmp32[7:0], tmp32[15:8], tmp32[23:16], tmp32[31:24]};
rb = rb + S12;

S32SDIR:
(Mem32 [rb + S12])
= {XRa[7:0], XRa[15:8], XRa[23:16], XRa[31:24]};
rb = rb + S12;
```

Description:

S32LDIR: same as S32LDDR in addition to rb is updated by the address formed.
S32SDIR: same as S32STDR in addition to rb is updated by the address formed.

NOTE:

1. S12 should be the multiple of 4.
2. The offset range represented by S12 is –2048 ~ 2044.
3. To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDIR    XR11, v0, 0x10
S32SDIR    XR3, k1, -0x50
```

1.5.6 S32LDI/S32SDI

Syntax:

```
S32LDI XRa, rb, S12
S32SDI XRa, rb, S12
```

Operation:

```
S32LDI:
XRa = (Mem32 [rb + S12]);
rb = rb + S12;

S32SDI:
(Mem32 [rb + S12]) = XRa;
rb = rb + S12;
```

Description:

S32LDI: same as S32LDD in addition to rb is updated by the address formed.
S32SDI: same as S32STD in addition to rb is updated by the address formed.

NOTE:

1. S12 should be the multiple of 4.
2. The offset range represented by S12 is –2048 ~ 2044.
3. To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDI  XR11, v0, 0x10
S32SDI  XR3, k1, -0x50
```

1.5.7 S32LDIVR/S32SDIVR

Syntax:

```
S32LDIVR   XRa, rb, rc, STRD2
S32SDIVR   XRa, rb, rc, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
S32LDIVR:
tmp32= (Mem32 [rb + (rc << STRD2)]);
XRa = {tmp32[7:0], tmp32[15:8], tmp32[23:16], tmp32[31:24]};
rb = rb + (rc << STRD2)

S32SDIVR:
(Mem32 [rb + (rc << STRD2)])
= {XRa[7:0], XRa[15:8], XRa[23:16], XRa[31:24]};
rb = rb + (rc << STRD2)
```

Description:

S32LDIVR: same as S32LDDVR in addition to rb is updated by the address formed.
S32SDIVR: same as S32STDVR in addition to rb is updated by the address formed.

NOTE:

To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDIVR   XR11, $5, v0, 1
S32SDIVR   XR3, a1, a2, 0
```

1.5.8 S32LDIV/S32SDIV

Syntax:

```
S32LDIV    XRa, rb, rc, STRD2
S32SDIV    XRa, rb, rc, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
S32LDIV:
XRa= (Mem32 [rb + (rc << STRD2)]);
rb = rb + (rc << STRD2)

S32SDIV:
(Mem32 [rb + (rc << STRD2)]) = XRa;
rb = rb + (rc << STRD2)
```

Description:

S32LDIV: same as S32LDDV in addition to rb is updated by the address formed.
S32SDIV: same as S32STDV in addition to rb is updated by the address formed.

NOTE:

To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
S32LDIV    XR11, $5, v0, 1
S32SDIV    XR3, a1, a2, 0
```

1.5.9 LXW

Syntax:

```
LXW    rd, rs, rt, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
rd= (Mem32 [rs + (rt << STRD2)]);
```

Description:

Load a word from the memory address formed by adding rs with rt<<STRD2, update rd with that word

NOTE:

To avoid address error exception, the address formed must be 4-byte aligned.

Example:

```
LXW $1, $5, v0, 1
```

1.5.10 S16LDD

Syntax:

```
S16LDD XRa, rb, S10, OPTN2
```

Parameter:

```
OPTN2: ptn0(000), ptn1(001), ptn2(010), ptn3(011)
```

Operation:

```
tmp16 = (Mem16 [rb + S10]);
switch(OPTN2) {
case 0: XRa[15:00] = tmp16;
case 1: XRa[31:16] = tmp16;
case 2: XRa[31:00] = {{16{sign of tmp16}}, tmp16};
case 3: XRa[31:00] = {tmp16, tmp16};
}
```

Description:

Load a half word from the memory address formed by adding rb with S10, permutate the half word in terms of specific pattern to form a word to update XRa

NOTE:

1. The valid offset range of S10 is –512 ~ 510, S10 must be multiple of 2
2. Only the specific half word position in XRa is updated for pattern 0~1
3. To avoid address error exception, the address formed must be 2-byte aligned

Example:

```
S16LDD  XR11, $5, -512, 3
```

1.5.11 S16LDI

Syntax:

```
S16LDI XRa, rb, S10, OPTN2
```

Parameter:

```
OPTN2: ptn0(00), ptn1(01), ptn2(10), ptn3(11)
```

Operation:

```
tmp16 = (Mem16 [rb + S10]);
switch(OPTN2) {
case 0: XRa[15:00] = tmp16;
case 1: XRa[31:16] = tmp16;
case 2: XRa[31:00] = {{16{sign of tmp16}}, tmp16};
case 3: XRa[31:00] = {tmp16, tmp16};
}
rb = rb + S10;
```

Description:

Load a half word from the memory address formed by adding rb with S10, permutate the half word in terms of specific pattern to form a word to update XRa, meanwile rb is updated by the address formed

NOTE:

1. The valid offset range of S10 is –512 ~ 510, S10 must be multiple of 2
2. Only the specific half word position in XRa is updated for pattern 0~1
3. To avoid address error exception, the address formed must be 2-byte aligned

Example:

```
S16LDI  XR11, $5, 2, 1
```

1.5.12 S16STD

Syntax:

```
S16STD XRa, rb, S10, OPTN2
```

Parameter:

```
OPTN2: ptn0(00), ptn1(01), 10~11 are reserved
```

Operation:

```
switch(OPTN2) {
case 0: tmp16 = XRa[15:00];
case 1: tmp16 = XRa[31:16];
}
(Mem16 [rb + S10]) = tmp16;
```

Description:

Select a half word from XRa in terms of specific pattern, write the half word to the memory address formed by adding rb with S10.

NOTE:

1. The valid offset range of S10 is –512 ~ 510, S10 must be multiple of 2
2. To avoid address error exception, the address formed must be 2-byte aligned.

Example:

```
S16STD  XR3, a1, -10, ptn0
```

1.5.13 S16SDI

Syntax:

```
S16SDI XRa, rb, S10, OPTN2
```

Parameter:

```
OPTN2: ptn0(00), ptn1(01), 10~11 are reserved
```

Operation:

```
switch(OPTN2) {
case 0: tmp16 = XRa[15:00];
case 1: tmp16 = XRa[31:16];
}
(Mem16 [rb + S10]) = tmp16;
rb = rb + S10;
```

Description:

Select a half word from XRa in terms of specific pattern, write the half word to the memory address formed by adding rb with S10, meanwhile rb is updated by the address formed.

NOTE:

1. The valid offset range of S10 is –512 ~ 510, S10 must be multiple of 2
2. To avoid address error exception, the address formed must be 2-byte aligned.

Example:

```
S16SDI  XR3, a1, 14, ptn0
```

1.5.14 LXH/LXHU

Syntax:

```
LXH    rd, rs, rt, STRD2
LXHU   rd, rs, rt, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
LXH:   rd= sign_ext32(Mem16 [rs + (rt << STRD2)]);
LXHU:  rd= zero_ext32(Mem16 [rs + (rt << STRD2)]);
```

Description:

Load a half word from the memory address formed by adding rs with rt<<STRD2, make sign-extension or zero-extension of the half word to update rd.

NOTE:

To avoid address error exception, the address formed must be 2-byte aligned.

Example:

```
LXH  v1, $5, v0, 1
LXHU t0, t1, t2, 0
```

1.5.15 S8LDD

Syntax:

```
S8LDD  XRa, rb, S8, OPTN3
```

Parameter:

```
OPTN3: ptn0(000), ptn1(001), ptn2(010), ptn3(011), ptn4(100),
       ptn5(101), ptn6(110), ptn7(111)
```

Operation:

```
tmp8 = (Mem8 [rb + S8]);
switch(OPTN3) {
case 0: XRa[7:0] = tmp8;
case 1: XRa[15:8] = tmp8;
case 2: XRa[23:16] = tmp8;
case 3: XRa[31:24] = tmp8;
case 4: XRa = {8'b0, tmp8, 8'b0, tmp8};
case 5: XRa = {tmp8, 8'b0, tmp8, 8'b0};
case 6: XRa = {{8{sign of tmp8}}, tmp8, {8{sign of tmp8}}, tmp8};
case 7: XRa = {tmp8, tmp8, tmp8, tmp8};
}
```

Description:

Load a byte from the memory address formed by adding rb with S8, permutate the byte in terms of specific pattern to form a word to update XRa

NOTE:

1. The offset range represented by S8 is –128 ~ 127.
2. Only the specific byte position in XRa is updated for pattern 0~3

Example:

```
S8LDD   XR11, $5, 1, 7
S8LDD   XR3, a1, -17, ptn7
```

1.5.16 S8LDI

Syntax:

```
S8LDI  XRa, rb, S8, OPTN3
```

Parameter:

```
OPTN3: ptn0(000), ptn1(001), ptn2(010), ptn3(011), ptn4(100),
       ptn5(101), ptn6(110), ptn7(111)
```

Operation:

```
tmp8 = (Mem8 [rb + S8]);
switch(OPTN3) {
case 0: XRa[7:0] = tmp8;
case 1: XRa[15:8] = tmp8;
case 2: XRa[23:16] = tmp8;
case 3: XRa[31:24] = tmp8;
case 4: XRa = {8'b0, tmp8, 8'b0, tmp8};
case 5: XRa = {tmp8, 8'b0, tmp8, 8'b0};
case 6: XRa = {{8{sign of tmp8}}, tmp8, {8{sign of tmp8}}, tmp8};
case 7: XRa = {tmp8, tmp8, tmp8, tmp8};
}
rb = rb + S8;
```

Description:

Load a byte from the memory address formed by adding rb with S8, permutate the byte in terms of specific pattern to form a word to update XRa, meanwile rb is updated by the address formed

NOTE:

1. The offset range represented by S8 is –128 ~ 127.
2. Only the specific byte position in XRa is updated for pattern 0~3

Example:

```
S8LDI   XR11, $5, 13, 3
S8LDI   XR3, a1, -1, ptn5
```

1.5.17 S8STD

Syntax:

```
S8STD  XRa, rb, S8, OPTN3
```

Parameter:

```
OPTN3: ptn0(000), ptn1(001), ptn2(010), ptn3(011), 100~111 are
       reserved
```

Operation:

```
switch(OPTN3) {
case 0: tmp8 = XRa[7:0];
case 1: tmp8 = XRa[15:8];
case 2: tmp8 = XRa[23:16];
case 3: tmp8 = XRa[31:24];
}
(Mem8 [rb + S8]) = tmp8;
```

Description:

Select a byte from XRa in terms of specific pattern, write the byte to the memory address formed by adding rb with S8.

NOTE:

The offset range represented by S8 is –128 ~ 127.

Example:

```
S8STD   XR11, $5, 1, 0
S8STD   XR3, a1, -17, ptn7
```

1.5.18 S8SDI

Syntax:

```
S8SDI  XRa, rb, S8, OPTN3
```

Parameter:

```
OPTN3: ptn0(000), ptn1(001), ptn2(010), ptn3(011), 100~111 are
       reserved
```

Operation:

```
switch(OPTN3) {
case 0: tmp8 = XRa[7:0];
case 1: tmp8 = XRa[15:8];
case 2: tmp8 = XRa[23:16];
case 3: tmp8 = XRa[31:24];
}
(Mem8 [rb + S8]) = tmp8;
rb = rb + S8;
```

Description:

Select a byte from XRa in terms of specific pattern, write the byte to the memory address formed by adding rb with S8, meanwile rb is updated by the address formed.

NOTE:

The offset range represented by S8 is –128 ~ 127.

Example:

```
S8SDI   XR11, $5, 1, 4
S8SDI   XR3, a1, -17, ptn3
```

1.5.19 LXB/LXBU

Syntax:

```
LXB    rd, rs, rt, STRD2
LXBU   rd, rs, rt, STRD2
```

Parameter:

```
STRD2: 0~2
```

Operation:

```
LXB:   rd= sign_ext32(Mem8 [rs + (rt << STRD2)]);
LXBU:  rd= zero_ext32(Mem8 [rs + (rt << STRD2)]);
```

Description:

Load a byte from the address formed by adding rs with rt<<STRD2, make sign-extension or zero-extension of the byte to update rd.

Example:

```
LXB  v1, $5, v0, 1
LXBU t0, t1, t2, 0
```

1.6 Multiplication with/without accumulation

1.6.1 S32MUL

Syntax:

```
S32MUL  XRa, XRd, rs, rt
```

Operation:

```
tmp64 = signed(rs) * signed(rt);
XRa = tmp64[63:32];
XRd = tmp64[31:00];
```

Description:

Make signed 32x32 multiply for rs and rt to form a 64-bit product, the top 32-bit of the product updates XRa, while the bottom one updates XRd.

Example:

```
S32MUL  XR11, XR7, $1, v0
S32MUL  XR0, XR4, $1, $3
S32MUL  XR9, XR0, v0, t0
```

1.6.2 S32MULU

Syntax:

```
S32MULU XRa, XRd, rs, rt
```

Operation:

```
tmp64 = unsign(rs) * unsign(rt);
XRa = tmp64[63:32];
XRd = tmp64[31:00];
```

Description:

Make unsigned 32x32 multiply for rs and rt to form a 64-bit product, the top 32-bit of the product updates XRa, while the bottom one updates XRd.

Example:

```
S32MULU     XR11, XR7, $1, v0
S32MULU     XR0, XR4, $1, $3
S32MULU     XR9, XR0, v0, t0
```

1.6.3 S32MADD

Syntax:

```
S32MADD XRa, XRd, rs, rt
```

Operation:

```
tmp64 = {XRa, XRd} + signed(rs) * signed(rt);
XRa = tmp64[63:32];
XRd = tmp64[31:00];
```

Description:

Make signed 32x32 multiply for rs and rt to form a 64-bit product, accumulate the product to the 64-bit value formed by binding XRa and XRd, the top 32-bit accumulation updates XRa while the bottom one updates XRd.

NOTE:

The instruction's executing will stain HI/LO, so take away available values in HI/LO ahead of time.

Example:

```
S32MADD    XR11, XR7, $1, v0
S32MADD    XR0, XR4, $1, $3
S32MADD    XR9, XR0, v0, t0
```

1.6.4 S32MADDU

Syntax:

```
S32MADDU       XRa, XRd, rs, rt
```

Operation:

```
tmp64 = {XRa, XRd} + unsign(rs) * unsign(rt);
XRa = tmp64[63:32];
XRd = tmp64[31:00];
```

Description:

Make unsigned 32x32 multiply for rs and rt to form a 64-bit product, accumulate the product to the 64-bit value formed by binding XRa and XRd, the top 32-bit accumulation updates XRa while the bottom one updates XRd.

NOTE:

The instruction's executing will stain HI/LO, so take away available values in HI/LO ahead of time.

Example:

```
S32MADDU    XR11, XR7, $1, v0
S32MADDU    XR0, XR4, $1, $3
S32MADDU    XR9, XR0, v0, t0
```

1.6.5 S32MSUB

Syntax:

```
S32MSUB XRa, XRd, rs, rt
```

Operation:

```
tmp64 = {XRa, XRd} - signed(rs) * signed(rt);
XRa = tmp64[63:32];
XRd = tmp64[31:00];
```

Description:

Make signed 32x32 multiply for rs and rt to form a 64-bit product, Use the 64-bit value formed by binding XRa and XRd to subtract the product, the top 32-bit of the minus result updates XRa while the bottom one updates XRd.

NOTE:

The instruction's executing will stain HI/LO, so take away available values in HI/LO ahead of time.

Example:

```
S32MSUB     XR11, XR7, $1, v0
S32MSUB     XR0, XR4, $1, $3
S32MSUB     XR9, XR0, v0, t0
```

1.6.6 S32MSUBU

Syntax:

```
S32MSUBU    XRa, XRd, rs, rt
```

Operation:

```
tmp64 = {XRa, XRd} - unsign(rs) * unsign(rt);
XRa = tmp64[63:32];
XRd = tmp64[31:00];
```

Description:

Make unsigned 32x32 multiply for rs and rt to form a 64-bit product, Use the 64-bit value formed by binding XRa and XRd to subtract the product, the top 32-bit of the minus result updates XRa while the bottom one updates XRd.

NOTE:

The instruction's executing will stain HI/LO, so take away available values in HI/LO ahead of time.

Example:

```
S32MSUBU    XR11, XR7, $1, v0
S32MSUBU    XR0, XR4, $1, $3
S32MSUBU    XR9, XR0, v0, t0
```

1.6.7 D16MUL

Syntax:

```
D16MUL  XRa, XRb, XRc, XRd, OPTN2
```

Parameter:

```
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
XRa=L32;
XRd=R32;
```

Description:

Make pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern. The left lane product updates XRa, the right lane one updates XRd.

Example:

```
D16MUL  XR1, XR3, XR1, XR7, XW
D16MUL  XR6, XR8, XR7, XR7, 0
D16MUL  XR6, XR8, XR7, XR7, 2
```

1.6.8 D16MULF

Syntax:

```
D16MULFXRa, XRb, XRc, OPTN2
```

Parameter:

```
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
{L32, R32} = {L32<<1, R32<<1};
If (MXU_CR.RD_EN)
    {L32, R32} = {round(L32), round(R32)};
XRa = {L32[31:16], R32[31:16]};
```

Description:

Make fractional pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern with rounding or not. Pack the two products' top half-word to XRa.

Example:

```
D16MULF     XR1, XR3, XR1, XW
D16MULF     XR6, XR8, XR7, 1
```

1.6.9 D16MULE

Syntax:

```
D16MULE XRa, XRb, XRc, XRd, OPTN2
```

Parameter:

```
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
{L32, R32} = {L32<<1, R32<<1};
If (MXU_CR.RD_EN)
    {L32, R32} = {round(L32), round(R32)};
XRa = L32;
XRd = R32;
```

Description:

Make fractional pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern with rounding or not. The left lane result updates XRa, the right lane one updates XRd.

Example:

```
D16MULE     XR1, XR3, XR1, XR9, XW
D16MULE     XR8, XR8, XR7, XR0, 3
```

1.6.10 D16MAC

Syntax:

```
D16MAC  XRa, XRb, XRc, XRd, APTN2, OPTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
XRa = XRa +/- L32;
XRd = XRd +/- R32;
```

Description:

Make pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern. XRa subsequently adds or subtracts (directed by left A|S) the left lane product to update itself, while XRd subsequently adds or subtracts (directed by right A|S) the right lane one to update itself.

Example:

```
D16MAC  XR1, XR3, XR1, XR9, SA, XW
D16MAC  XR3, XR8, XR7, XR4, 3, 2
```

1.6.11 D16MACF

Syntax:

```
D16MACF XRa, XRb, XRc, XRd, APTN2, OPTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
{L32, R32} = {L32<<1, R32<<1}
L32 = XRa +/- L32;
R32 = XRd +/- R32;
If (MXU_CR.RD_EN)
    {L32, R32} = {round(L32), round(R32)}
XRa = {L32[31:16], R32[31:16]}
```

Description:

Make fractional pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern. XRa subsequently adds or subtracts (directed by left A|S) the left lane result, while XRd subsequently adds or subtracts (directed by right A|S) the right lane one, round the two sum results if necessray, finally, pack the two results' top half-word to XRa.

Example:

```
D16MACF    XR1, XR3, XR1, XR9, SA, HW
D16MACF    XR3, XR8, XR7, XR4, 2, 0
```

1.6.12 D16MACE

Syntax:

```
D16MACF XRa, XRb, XRc, XRd, APTN2, OPTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
{L32, R32} = {L32<<1, R32<<1}
L32 = XRa +/- L32;
R32 = XRd +/- R32;
If (MXU_CR.RD_EN)
    {L32, R32} = {round(L32), round(R32)}
XRa = L32;
XRd = R32;
```

Description:

Make fractional pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern. XRa subsequently adds or subtracts (directed by left A|S) the left lane result, while XRd subsequently adds or subtracts (directed by right A|S) the right lane one, round the two sum results if necessray, updates XRa with the left result while XRd with the right one.

Example:

```
D16MACE    XR1, XR3, XR1, XR9, SA, XW
D16MACE    XR3, XR8, XR7, XR4, 1, 3
```

1.6.13 D16MADL

Syntax:

```
D16MADL     XRa, XRb, XRc, XRd, APTN2, OPTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{L32, R32} = pattern_MUL(XRb, XRc);
Short1 = XRa[31:16] +/- L32[15:00];
Short0 = XRa[15:00] +/- R32[15:00];
XRd = {Short1, Short0}
```

Description:

make pattern_MUL multiply for XRb and XRc according to the operand combinatorial pattern. Subsequently add/subtract two products' bottom half-words with the two 16-bit portions in XRa respectively. Finally, Pack two 16-bit sum results to XRd.

Example:

```
D16MADL     XR1, XR3, XR1, XR9, SA, XW
D16MADL     XR3, XR8, XR7, XR4, 0, 0
```

1.6.14 S16MAD

Syntax:

```
S16MAD XRa, XRb, XRc, XRd, APTN1, OPTN1
```

Parameter:

```
APTN1: A(0), S(1);
OPTN1: 00-XRb.H*XRc.H, 01-XRb.L*XRc.L, 10-XRb.H*XRc.L,
       11-XRb.L*XRc.H;
```

Operation:

```
If (OPTN2 == 0) XRd = XRa +/- XRb.H * XRc.H;
If (OPTN2 == 1) XRd = XRa +/- XRb.L * XRc.L;
If (OPTN2 == 2) XRd = XRa +/- XRb.H * XRc.L;
If (OPTN2 == 3) XRd = XRa +/- XRb.L * XRc.H;
```

Description:

Make 16x16 multiply for XRb and XRc according to the operand combinatorial pattern. XRa subsequently adds or subtracts the product to update XRd

Example:

```
S16MAD  XR1, XR3, XR1, XR9, S, 0
S16MAD  XR3, XR8, XR7, XR4, 0, 2
```

1.6.15 Q8MUL

Syntax:

```
Q8MUL   XRa, XRb, XRc, XRd
```

Operation:

```
Short3 = unsign(XRb[31:24]) * unsign(XRc[31:24]);
Short2 = unsign(XRb[23:16]) * unsign(XRc[23:16]);
Short1 = unsign(XRb[15:08]) * unsign(XRc[15:08]);
Short0 = unsign(XRb[07:00]) * unsign(XRc[07:00]);
XRa = {Short3, Short2};
XRd = {Short1, Short0};
```

Description:

Make four parallel unsigned 8x8 multiply for four corresponding byte pairs in XRb and XRc, two 16-bit products generated from the operand's MSB position update XRa, the other two from the operand's LSB position update XRd.

Example:

```
Q8MUL   XR1, XR3, XR4, XR9
```

1.6.16 Q8MULSU

Syntax:

```
Q8MULSU XRa, XRb, XRc, XRd
```

Parameter:

Operation:

```
Short3 = signed(XRb[31:24]) * unsign(XRc[31:24]);
Short2 = signed(XRb[23:16]) * unsign(XRc[23:16]);
Short1 = signed(XRb[15:08]) * unsign(XRc[15:08]);
Short0 = signed(XRb[07:00]) * unsign(XRc[07:00]);
XRa = {Short3, Short2};
XRd = {Short1, Short0};
```

Description:

Make four parallel signed 8x8 multiply for four corresponding byte pairs in XRb (contains four signed bytes) and XRc (contains four unsigned bytes), two 16-bit products generated from the operand's MSB position update XRa, the other two from the operand's LSB position update XRd.

Example:

```
Q8MULSU     XR1, XR3, XR4, XR9
```

1.6.17 Q8MAC

Syntax:

```
Q8MAC   XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Short3 = XRa[31:16] +/- unsign(XRb[31:24]) * unsign(XRc[31:24]);
Short2 = XRa[15:00] +/- unsign(XRb[23:16]) * unsign(XRc[23:16]);
Short1 = XRd[31:16] +/- unsign(XRb[15:08]) * unsign(XRc[15:08]);
Short0 = XRd[15:00] +/- unsign(XRb[07:00]) * unsign(XRc[07:00]);
XRa = {Short3, Short2};
XRd = {Short1, Short0};
```

Description:

Make four parallel unsigned 8x8 multiply for four corresponding byte pairs in XRb and XRc, two 16-bit products generated from the operand's MSB position accumulate to XRa which contains two packed 16-bits, the other two from the operand's LSB position accumulate to XRd which contains two packed 16-bits.
Note that the left A|S directs the add/subtract operation of Short3 and Short2 while the right one for Short1 and Short0.

Example:

```
Q8MAC   XR1, XR3, XR4, XR9, AA
```

1.6.18 Q8MACSU

Syntax:

```
Q8MACSU XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Short3 = XRa[31:16] +/- signed(XRb[31:24]) * unsign(XRc[31:24]);
Short2 = XRa[15:00] +/- signed(XRb[23:16]) * unsign(XRc[23:16]);
Short1 = XRd[31:16] +/- signed(XRb[15:08]) * unsign(XRc[15:08]);
Short0 = XRd[15:00] +/- signed(XRb[07:00]) * unsign(XRc[07:00]);
XRa = {Short3, Short2};
XRd = {Short1, Short0};
```

Description:

Make four parallel signed 8x8 multiply for four corresponding byte pairs in XRb (contains four signed bytes) and XRc (contains four unsigned bytes), two 16-bit products generated from the operand's MSB position accumulate to XRa which contains two packed 16-bits, the other two from the operand's LSB position accumulate to XRd which contains two packed 16-bits.

Note that left A|S directs the add/subtract operation of Short3 and Short2 while the right one for Short1 and Short0.

Example:

```
Q8MACSU     XR1, XR3, XR4, XR9, AA
```

1.6.19 Q8MADL (obsolete)

Syntax:

```
Q8MADL  XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
XRd[31:24] = XRa[31:24] +/- trunc_8(unsign(XRb[31:24]) *
       unsign(XRc[31:24]));
XRd[23:16] = XRa[23:16] +/- trunc_8(unsign(XRb[23:16]) *
       unsign(XRc[23:16]));
XRd[15:08] = XRa[15:08] +/- trunc_8(unsign(XRb[15:08]) *
       unsign(XRc[15:08]));
XRd[07:00] = XRa[07:00] +/- trunc_8(unsign(XRb[07:00]) *
       unsign(XRc[07:00]));
```

Description:

Make four parallel unsigned 8x8 multiply for four corresponding byte pairs in XRb and XRc, add each lower 8-bit of the four 16-bit products to the corresponding one in XRa, respectively. The final four 8-bit sum results update XRd.
Note that left A|S directs the add/subtract operation of bit [31:24] and [23:16] while the right one for [15:08] and [07:00].

NOTE:

It is not recommended to use this instruction for it may be discarded in the future.

1.7 Add and subtract

1.7.1 D32ADD

Syntax:

```
D32ADD XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Int1 = XRb +/- XRc;
Int0 = XRb +/- XRc;
XRa = Int1;
XRd = Int0;
```

Description:

Make dual 32-bit add/subtract operations for XRb and XRc, the dual results update XRa and XRd respectively. Note that left A|S directs add/subtract operation of Int1, while the right one for Int0.

NOTE:

The carry-out of the adder for Int1 updates the LC field of MXU_CR when XRa != XR0, the carry-out of the adder for Int0 updates the RC field of MXU_CR when XRd != XR0.

Example:

```
D32ADD  XR3, XR2, XR1, XR4, SA
```

1.7.2 D32ADDC

Syntax:

```
D32ADDC XRa, XRb, XRc, XRd
```

Operation:

```
XRa += XRb + LC;
XRd += XRc + RC;
```

Description:

Adding XRa and XRb together with LC to update XRa, adding XRd and XRc together with RC to update XRd.

NOTE:

The contents of LC and RC fields of MXU_CR retain until an executing of a D32ADD instruction.

Example:

```
D32ADDC     XR3, XR2, XR1, XR4
```

1.7.3 D32ACC

Syntax:

```
D32ACC XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Int1 = XRb +/- XRc;
Int0 = XRb +/- XRc;
XRa += Int1;
XRd += Int0;
```

Description:

Make dual 32-bit add/subtract operations for XRb and XRc, the dual results are added to XRa and XRd respectively. Note that left A|S directs add/subtract operation of Int1, while the right one for Int0.

Example:

```
D32ACC  XR3, XR2, XR1, XR4, SA
```

1.7.4 D32ACCM

Syntax:

```
D32ACCM XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
XRa +/-= (XRb + XRc);
XRd +/-= (XRb - XRc);
```

Description:

XRa adds or subtracts (directed by left A|S) the adding result of XRb and XRc to update XRa, meanwile XRd adds or subtracts (directed by right A|S) the subtracting result of XRb and XRc to update XRd.

Example:

```
D32ACCM    XR3, XR2, XR1, XR4, SA
```

1.7.5 D32ASUM

Syntax:

```
D32ASUM XRa, XRb, XRc, XRd, APTN2
```

Parameter:

```
APTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
XRa +/-= XRb;
XRd +/-= XRc;
```

Description:

XRa adds or subtracts (directed by left A|S) XRb to update XRa, XRd adds or subtracts (directed by right A|S) XRc to update XRd.

Example:

```
D32ASUM     XR3, XR2, XR1, XR4, SA
```

1.7.6 S32CPS

Syntax:

```
S32CPS  XRa, XRb, XRc
```

Operation:

```
If (XRc < 0) XRa = 0 – XRb;
Else XRa = XRb;
```

Description:

Get negative XRb or original XRb according to the sign of XRc to update XRa.

Example:

```
S32CPS  XR2, XR1, XR3
```

1.7.7 S32SLT

Syntax:

```
S32SLT  XRa, XRb, XRc
```

Operation:

```
XRa = signed(XRb)< signed(XRc)?1:0;
```

Description:

Update XRa by the bool result of the signed 32-bit less-than comparison

Example:

```
S32SLT  XR3, XR2, XR1
```

1.7.8 S32MOVZ

Syntax:

```
S32MOVZ XRa, XRb, XRc
```

Operation:

```
If (XRb == 0)
    XRa = XRc;
```

Description:

Conditonal move XRc to XRa if XRb equals zero.

Example:

```
S32MOVZ    XR3, XR2, XR1
```

1.7.9 S32MOVN

Syntax:

```
S32MOVN XRa, XRb, XRc
```

Operation:

```
If (XRb != 0)
    XRa = XRc;
```

Description:

Conditonal move XRc to XRa if XRb does not equal zero.

Example:

```
S32MOVN    XR3, XR2, XR1
```

1.7.10 Q16ADD

Syntax:

```
Q16ADD XRa, XRb, XRc, XRd, EPTN2, OPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
OPTN2: WW(00), LW(01), HW(10), XW(11);
```

Operation:

```
{short3, short2} = pattern_SUM(XRb, XRc);
{short1, short0} = pattern_SUM(XRb, XRc);
XRa = {short3, short2};
XRd = {short1, short0};
```

Description:

Make two parallel pattern_SUM operations to update XRa and XRd. It is determined by **OPTN2** that how the operands (XRb and XRc) are assembled. Note that the left A|S directs the add/subtract operation of short3 and short2, while the right one for short1 and short0.

Example:

```
Q16ADD  XR3, XR2, XR1, XR4, SA, WX
```

1.7.11 Q16ACC

Syntax:

```
Q16ACC XRa, XRb, XRc, XRd, EPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Short3 = XRb[31:16] +/- XRc[31:16];
Short2 = XRb[15:00] +/- XRc[15:00];
Short1 = XRb[31:16] +/- XRc[31:16];
Short0 = XRb[15:00] +/- XRc[15:00];
XRa[31:16] += Short3;
XRa[15:00] += Short2;
XRd[31:16] += Short1;
XRd[15:00] += Short0;
```

Description:

As above expressions. Note that the left A|S directs the sum/difference operation of short3 and short2, while the right one for short1 and short0.

Example:

```
Q16ACC  XR3, XR2, XR1, XR4, SA
```

1.7.12 Q16ACCM

Syntax:

```
Q16ACCM XRa, XRb, XRc, XRd, EPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Short3 = XRb[31:16];
Short2 = XRb[15:00];
Short1 = XRc[31:16];
Short0 = XRc[15:00];
XRa[31:16] +/-= Short3;
XRa[15:00] +/-= Short2;
XRd[31:16] +/-= Short1;
XRd[15:00] +/-= Short0;
```

Description:

Two packed 16-bits in XRb are added or subtracted (directed by left A|S) to corresponding ones in XRa, respectively.
Two packed 16-bits in XRc are added or subtracted (directed by left A|S) to corresponding ones in XRd, respectively.

Example:

```
Q16ACCM     XR3, XR2, XR1, XR4, SA
```

1.7.13 D16ASUM

Syntax:

```
D16ASUM XRa, XRb, XRc, XRd, EPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
L32 = sign_ext32(XRb[31:16]) + sign_ext32(XRb[15:00]);
R32 = sign_ext32(XRc[31:16]) + sign_ext32(XRc[15:00]);
XRa +/-= L32;
XRd +/-= R32;
```

Description:

Make sign_ext32 for two 16-bits in XRb and subsequently add them together to form a 32-bit result to be added or subtracted to XRa (directed by left A|S), same manipulation for XRc and XRd controlled by right A|S.

Example:

```
D16ASUM    XR3, XR2, XR1, XR4, SA
```

1.7.14 D16CPS

Syntax:

```
D16CPS  XRa, XRb, XRc
```

Operation:

```
If (XRc[31:16]<0) short1 = 0-XRb[31:16];
else short1 = XRb[31:16]
If (XRc[15:0]<0) short0 = 0-XRb[15:0];
else short0 = XRb[15:0]
XRa = {short1, short0};
```

Description:

It is a dual 16-bit packed operation. For each 16-bit portion, get negative value or original one in XRb according to the corresponding 16-bit value' sign in XRc to update corresponding 16-bit portion in XRa.

Example:

```
D16CPS  XR2, XR1, XR3
```

1.7.15 D16SLT

Syntax:

```
D16SLT  XRa, XRb, XRc
```

Operation:

```
XRa[31:16]= signed(XRb[31:16])< signed(XRc[31:16])?1:0;
XRa[15:00]= signed(XRb[15:00])< signed(XRc[15:00])?1:0;
```

Description:

It is a dual 16-bit packed operation. For each 16-bit portion, make signed 16-bit less-than comparison of XRb and XRc to get 16-bit bool result to update corresponding 16-bit portion in XRa.

Example:

```
D16SLT  XR3, XR2, XR1
```

1.7.16 D16MOVZ

Syntax:

```
D16MOVZ XRa, XRb, XRc
```

Operation:

```
If (XRb[31:16]== 0)
    XRa[31:16]= XRc[31:16];
If (XRb[15:00]== 0)
    XRa[15:00]= XRc[15:00];
```

Description:

It is a dual 16-bit packed operation. For each 16-bit portion, conditional move 16-bit value of XRc to corresponding 16-bit portion in XRa if corresponding 16-it value of XRb equals zero.

Example:

```
D16MOVZ    XR3, XR2, XR1
```

1.7.17 D16MOVN

Syntax:

```
D16MOVN XRa, XRb, XRc
```

Operation:

```
If (XRb[31:16]!= 0)
    XRa[31:16]= XRc[31:16];
If (XRb[15:00]!= 0)
    XRa[15:00]= XRc[15:00];
```

Description:

It is a dual 16-bit packed operation. For each 16-bit portion, conditional move 16-bit value of XRc to corresponding 16-bit portion in XRa if corresponding 16-it value of XRb equals non-zero.

Example:

```
D16MOVN    XR3, XR2, XR1
```

1.7.18 D16AVG

Syntax:

```
D16AVG  XRa, XRb, XRc
```

Operation:

```
Short1= (XRb[31:16] + XRc[31:16]) >> 1;
Short0= (XRb[15:00] + XRc[15:00]) >> 1;
XRa = {Short1, Short0};
```

Description:

Make dual 16-bit average operations for XRb (contain two packed 16-bits) and XRc (contain two packed 16-bits), pack two 16-bit average results to XRa.

Example:

```
D16AVG  XR3, XR2, XR1
```

1.7.19 D16AVGR

Syntax:

```
D16AVGR XRa, XRb, XRc
```

Operation:

```
Short1 = (XRb[31:16] + XRc[31:16] + 1) >> 1;
Short0 = (XRb[15:00] + XRc[15:00] + 1) >> 1;
XRa = {Short1, Short0};
```

Description:

Make dual 16-bit average with explicit rounding operations for XRb (contain two packed 16-bits) and XRc (contain two packed 16-bits), pack two 16-bit average results to XRa.

Example:

```
D16AVGR     XR3, XR2, XR1
```

1.7.20 Q8ADD (obsolete)

Syntax:

```
Q8ADD  XRa, XRb, XRc, EPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
Byte3 = XRb[31:24] +/- XRc[31:24];
Byte2 = XRb[23:16] +/- XRc[23:16];
Byte1 = XRb[15:08] +/- XRc[15:08];
Byte0 = XRb[07:00] +/- XRc[07:00];
XRa = {Byte3, Byte2, Byte1, Byte0};
```

Description:

Make quadruple add/subtract operations for XRb (contain four packed 8-bits) and XRc (contain four packed 8-bits), pack four 8-bit sum results to XRa. Note that the left A|S directs the sum/difference operation of byte3 and byte2, while the right one for byte1 and byte0.

NOTE:

It is not recommended to use this instruction for it may be discarded in the future.

Example:

```
Q8ADD   XR3, XR2, XR1, AA
```

1.7.21 Q8ADDE

Syntax:

```
Q8ADDE XRa, XRb, XRc, XRd, EPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
XRa[31:16] = zero_ext16(XRb[31:24]) +/- zero_ext16(XRc[31:24]);
XRa[15:00] = zero_ext16(XRb[23:16]) +/- zero_ext16(XRc[23:16]);
XRd[31:16] = zero_ext16(XRb[15:08]) +/- zero_ext16(XRc[15:08]);
XRd[15:00] = zero_ext16(XRb[07:00]) +/- zero_ext16(XRc[07:00]);
```

Description:

It is a quadruple packed 8-bit operation. For each 8-bit portion, zero extends unsigned byte in XRb and XRc to 16-bit to make 16-bit add/subtract operation. Note that the left A|S denotes the dual 16-bit plus(A)/minus(S) operations which results update XRa while the right one for XRd.

Example:

```
Q8ADDE  XR3, XR2, XR1, XR4, AS
```

1.7.22 Q8ACCE

Syntax:

```
Q8ACCE XRa, XRb, XRc, XRd, EPTN2
```

Parameter:

```
EPTN2: AA(00), AS(01), SA(10), SS(11);
```

Operation:

```
XRa[31:16] += zero_ext16(XRb[31:24]) +/- zero_ext16(XRc[31:24]);
XRa[15:00] += zero_ext16(XRb[23:16]) +/- zero_ext16(XRc[23:16]);
XRd[31:16] += zero_ext16(XRb[15:08]) +/- zero_ext16(XRc[15:08]);
XRd[15:00] += zero_ext16(XRb[07:00]) +/- zero_ext16(XRc[07:00]);
```

Description:

It is a quadruple packed 8-bit operation. For each 8-bit portion, zero extends unsigned byte in XRb and XRc to 16-bit to make 16-bit add/subtract operation. Note that the left A|S denotes the dual 16-bit plus(A)/minus(S) operations which results are accumulated to XRa while the right one for XRd.

Example:

```
Q8ACCE  XR3, XR2, XR1, XR4, AS
```

1.7.23 D8SUM

Syntax:

```
D8SUM  XRa, XRb, XRc
```

Operation:

```
XRa[31:16] = zero_ext16(XRb[31:24]) + zero_ext16(XRb[23:16]) +
       zero_ext16(XRb[15:8]) + zero_ext16(XRb[7:0]);
XRa[15:0] = zero_ext16(XRc[31:24]) + zero_ext16(XRc[23:16]) +
       zero_ext16(XRc[15:8]) + zero_ext16(XRc[7:0]);
```

Description:

Add four unsigned bytes of XRb together to update the upper 16-bit portion of XRa, add four unsigned bytes of XRc together to update the lower 16-bit portion of XRa.

Example:

```
D8SUM   XR3, XR2, XR1
```

1.7.24 D8SUMC

Syntax:

```
D8SUMC XRa, XRb, XRc
```

Operation:

```
XRa[31:16] = (zero_ext16(XRb[31:24]) + zero_ext16(XRb[23:16]) + 1)
       + (zero_ext16(XRb[15:8]) + zero_ext16(XRb[7:0]) + 1);
XRa[15:0] = (zero_ext16(XRc[31:24]) + zero_ext16(XRc[23:16]) + 1)
       + (zero_ext16(XRc[15:8]) + zero_ext16(XRc[7:0]) + 1);
```

Description:

Add four unsigned bytes of XRb together first and subseqently add value 2 to update the upper 16-bit portion of XRa,
add four unsigned bytes of XRc together first and subsequently add value 2 to update the lower 16-bit portion of XRa.

Example:

```
D8SUMC  XR3, XR2, XR1
```

1.7.25 Q8ABD

Syntax:

```
Q8ABD   XRa, XRb, XRc
```

Operation:

```
XRa[31:24] = abs(unsign(XRb[31:24]) – unsign(XRc[31:24]));
XRa[23:16] = abs(unsign(XRb[23:16]) – unsign(XRc[23:16]));
XRa[15:08] = abs(unsign(XRb[15:08]) – unsign(XRc[15:08]));
XRa[07:00] = abs(unsign(XRb[07:00]) – unsign(XRc[07:00]));
```

Description:

Update XRa with four packed 8-bit absolute differences calculated from four corresponding 8-bit pairs in XRb and XRc

Example:

```
Q8ABD   XR3, XR2, XR1
```

1.7.26 Q8SLT

Syntax:

```
Q8SLT   XRa, XRb, XRc
```

Operation:

```
XRa[31:24]= sign_ext16(XRb[31:24])< sign_ext16(XRc[31:24])?1:0;
XRa[23:16]= sign_ext16(XRb[23:16])< sign_ext16(XRc[23:16])?1:0;
XRa[15:08]= sign_ext16(XRb[15:08])< sign_ext16(XRc[15:08])?1:0;
XRa[07:00]= sign_ext16(XRb[07:00])< sign_ext16(XRc[07:00])?1:0;
```

Description:

Update XRa with quadruple signed less-than compare results calculated from XRb (contains four 8-bit values) and XRc (contains four 8-bit values)

Example:

```
Q8SLT   XR3, XR2, XR1
```

1.7.27 Q8SLTU

Syntax:

```
Q8SLTU  XRa, XRb, XRc
```

Operation:

```
XRa[31:24]= zero_ext16(XRb[31:24])< zero_ext16(XRc[31:24])?1:0;
XRa[23:16]= zero_ext16(XRb[23:16])< zero_ext16(XRc[23:16])?1:0;
XRa[15:08]= zero_ext16(XRb[15:08])< zero_ext16(XRc[15:08])?1:0;
XRa[07:00]= zero_ext16(XRb[07:00])< zero_ext16(XRc[07:00])?1:0;
```

Description:

Update XRa with quadruple unsigned less-than compare results calcuated from XRb (contains four 8-bit values) and XRc (contains four 8-bit values)

Example:

```
Q8SLTU  XR3, XR2, XR1
```

1.7.28 Q8MOVZ

Syntax:

```
Q8MOVZ  XRa, XRb, XRc
```

Operation:

```
If (XRb[31:24]== 0)
    XRa[31:24]= XRc[31:24];
If (XRb[23:16]== 0)
    XRa[23:16]= XRc[23:16];
If (XRb[15:08]== 0)
    XRa[15:08]= XRc[15:08];
If (XRb[07:00]== 0)
    XRa[07:00]= XRc[07:00];
```

Description:

It is a quadruple 8-bit packed operation. For each 8-bit portion, conditional move 8-bit value of XRc to corresponding 8-bit portion in XRa if corresponding 8-it value of XRb equals zero.

Example:

```
Q8MOVZ  XR3, XR2, XR1
```

1.7.29 Q8MOVN

Syntax:

```
Q8MOVN  XRa, XRb, XRc
```

Operation:

```
If (XRb[31:24]!= 0)
    XRa[31:24]= XRc[31:24];
If (XRb[23:16]!= 0)
    XRa[23:16]= XRc[23:16];
If (XRb[15:08]!= 0)
    XRa[15:08]= XRc[15:08];
If (XRb[07:00]!= 0)
    XRa[07:00]= XRc[07:00];
```

Description:

It is a quadruple 8-bit packed operation. For each 8-bit portion, conditional move 8-bit value of XRc to corresponding 8-bit portion in XRa if corresponding 8-it value of XRb equals non-zero.

Example:

```
Q8MOVN  XR3, XR2, XR1
```

1.7.30 Q8SAD

Syntax:

```
Q8SAD   XRa, XRb, XRc, XRd
```

Operation:

```
int3 = abs(zero_ext16(XRb[31:24]) – zero_ext16(XRc[31:24]));
int2 = abs(zero_ext16(XRb[23:16]) – zero_ext16(XRc[23:16]));
int1 = abs(zero_ext16(XRb[15:08]) – zero_ext16(XRc[15:08]));
int0 = abs(zero_ext16(XRb[07:00]) – zero_ext16(XRc[07:00]));
XRa = int3 + int2 + int1 + int0;
XRd += int3 + int2 + int1 + int0;
```

Description:

Typical SAD operation for motion estimation.

Example:

```
Q8SAD   XR3, XR2, XR1, XR4
```

1.7.31 Q8AVG

Syntax:

```
Q8AVG   XRa, XRb, XRc
```

Operation:

```
byte3 = (zero_ext9(XRb[31:24]) + zero_ext9(XRc[31:24])) >> 1;
byte2 = (zero_ext9(XRb[23:16]) + zero_ext9(XRc[23:16])) >> 1;
byte1 = (zero_ext9(XRb[15:08]) + zero_ext9(XRc[15:08])) >> 1;
byte0 = (zero_ext9(XRb[07:00]) + zero_ext9(XRc[07:00])) >> 1;
XRa = {byte3, byte2, byte1, byte0};
```

Description:

Make quadruple 8-bit average operations for XRb (contain four packed 8-bits) and XRc (contain four packed 8-bits), pack four 8-bit average results to XRa.

Example:

```
Q8AVG   XR3, XR2, XR1
```

1.7.32 Q8AVGR

Syntax:

```
Q8AVGR  XRa, XRb, XRc
```

Operation:

```
byte3 = (zero_ext9(XRb[31:24]) + zero_ext9(XRc[31:24]) + 1) >> 1;
byte2 = (zero_ext9(XRb[23:16]) + zero_ext9(XRc[23:16]) + 1) >> 1;
byte1 = (zero_ext9(XRb[15:08]) + zero_ext9(XRc[15:08]) + 1) >> 1;
byte0 = (zero_ext9(XRb[07:00]) + zero_ext9(XRc[07:00]) + 1) >> 1;
XRa = {byte3, byte2, byte1, byte0};
```

Description:

Make quadruple 8-bit average with explicit rounding operations for XRb (contain four packed 8-bits) and XRc (contain four packed 8-bits), pack four 8-bit average results to XRa.

Example:

```
Q8AVGR  XR3, XR2, XR1
```

1.8 Shift

1.8.1 D32SLL/D32SLR/D32SAR

Syntax:

```
D32SLL  XRa, XRb, XRc, XRd, SFT4
D32SLR  XRa, XRb, XRc, XRd, SFT4
D32SAR  XRa, XRb, XRc, XRd, SFT4
```

Parameter:

```
SFT4: range 0 ~ 15
```

Operation:

```
D32SLL:
XRa = XRb << SFT4;
XRd = XRc << SFT4;

D32SLR:
XRa = XRb (>>L) SFT4;
XRd = XRc (>>L) SFT4;

D32SAR:
XRa = XRb (>>A) SFT4;
XRd = XRc (>>A) SFT4;
```

Description:

Make two 32-bit parallel shift for XRb and XRc to update XRa and XRd respectively. For logic right shift, filling zero to the left drained bit positions. For arithmetic right shift, filling sign bit to the left drained bit positions.

Example:

```
D32SAR  XR3, XR2, XR5, 11
```

1.8.2 D32SARL

Syntax:

```
D32SARL XRa, XRb, XRc, SFT4
```

Parameter:

```
SFT4: range 0 ~ 15
```

Operation:

```
L32 = XRb (>>A) SFT4;
R32 = XRc (>>A) SFT4;
XRa = {L32[15:00], R32[15:00]};
```

Description:

Make two 32-bit parallel arithmetic right shift for XRb and XRc, pack two lower 16-bit shift results to update XRa.

Example:

```
D32SARL     XR3, XR2, XR5, 11
```

1.8.3 D32SLLV/D32SLRV/D32SARV

Syntax:

```
D32SLLV XRa, XRd, rb
D32SLRV XRa, XRd, rb
D32SARV XRa, XRd, rb
```

Parameter:

Operation:

```
D32SLLV:
XRa = XRa << rb[3:0];
XRd = XRd << rb[3:0];

D32SLRV:
XRa = XRa (>>L) rb[3:0];
XRd = XRd (>>L) rb[3:0];

D32SARV:
XRa = XRa (>>A) rb[3:0];
XRd = XRd (>>A) rb[3:0];
```

Description:

Make two 32-bit parallel shift for XRa and XRd. For logic right shift, filling zero to the left drained bit positions. For arithmetic right shift, filling sign bit to the left drained bit positions.

Example:

```
D32SARV    XR3, XR2, $21
```

1.8.4 D32SARW

Syntax:

```
D32SARW XRa, XRb, XRc, rb
```

Parameter:

Operation:

```
L32 = XRb (>>A) rb[3:0];
R32 = XRc (>>A) rb[3:0];
XRa = {L32[15:00], R32[15:00]};
```

Description:

Make two 32-bit parallel arithmetic right shift for XRb and XRc, pack two lower 16-bit shift results to update XRa.

Example:

```
D32SARW     XR3, XR2, XR5, $6
```

1.8.5 Q16SLL/Q16SLR/Q16SAR

Syntax:

```
Q16SLL  XRa, XRb, XRc, XRd, SFT4
Q16SLR  XRa, XRb, XRc, XRd, SFT4
Q16SAR  XRa, XRb, XRc, XRd, SFT4
```

Parameter:

```
SFT4: range 0 ~ 15
```

Operation:

```
Q16SLL:
Short3 = XRb[31:16] << sft4;
Short2 = XRb[15:00] << sft4;
Short1 = XRc[31:16] << sft4;
Short0 = XRc[15:00] << sft4;
XRa = {Short3, Short2}; XRd = {Short1, Short0};

Q16SLR:
Short3 = XRb[31:16] (>>L) sft4;
Short2 = XRb[15:00] (>>L) sft4;
Short1 = XRc[31:16] (>>L) sft4;
Short0 = XRc[15:00] (>>L) sft4;
XRa = {Short3, Short2}; XRd = {Short1, Short0};

Q16SAR:
Short3 = XRb[31:16] (>>A) sft4;
Short2 = XRb[15:00] (>>A) sft4;
Short1 = XRc[31:16] (>>A) sft4;
Short0 = XRc[15:00] (>>A) sft4;
XRa = {Short3, Short2}; XRd = {Short1, Short0};
```

Description:

Shift two packed 16-bits in XRb to update XRa, and shift two packed 16-bits in XRc to update XRd. For logic right shift, filling zero to the left drained bit positions. For arithmetic right shift, filling sign bit to the left drained bit positions.

Example:

```
Q16SAR  XR3, XR2, XR5, XR9, 11
```

1.8.6 Q16SLLV/Q16SLRV/Q16SARV

Syntax:

```
Q16SLLV XRa, XRd, rb
Q16SLRV XRa, XRd, rb
Q16SARV XRa, XRd, rb
```

Operation:

```
Q16SLLV:
Short3 = XRa[31:16] << rb[3:0];
Short2 = XRa[15:00] << rb[3:0];
Short1 = XRd[31:16] << rb[3:0];
Short0 = XRd[15:00] << rb[3:0];
XRa = {Short3, Short2}; XRd = {Short1, Short0};

Q16SLRV:
Short3 = XRa[31:16] (>>L) rb[3:0];
Short2 = XRa[15:00] (>>L) rb[3:0];
Short1 = XRd[31:16] (>>L) rb[3:0];
Short0 = XRd[15:00] (>>L) rb[3:0];
XRa = {Short3, Short2}; XRd = {Short1, Short0};

Q16SARV:
Short3 = XRa[31:16] (>>A) rb[3:0];
Short2 = XRa[15:00] (>>A) rb[3:0];
Short1 = XRd[31:16] (>>A) rb[3:0];
Short0 = XRd[15:00] (>>A) rb[3:0];
XRa = {Short3, Short2}; XRd = {Short1, Short0};
```

Description:

Shift two packed 16-bits in XRa to update XRa, and shift two packed 16-bits in XRd to update XRd. For logic right shift, filling zero to the left drained bit positions. For arithmetic right shift, filling sign bit to the left drained bit positions.

Example:

```
Q16SARV    XR3, XR9, $3
```

1.8.7 S32EXTR

Syntax:

```
S32EXTR XRa, XRd, rs, bits5
```

Operation:

```
position = rs[4:0];
left = 32 – position;
if (bits5 > left) {
    r_bits = bits5 – left; r_sft = 32 – r_bits;
    tmp = (XRa (<<L) r_bits) | (XRd (>>L) r_sft);
} else {
    r_bits = left – bits5;
    tmp = (XRa (>>L) r_bits);
}
XRa = tmp & ((1<<bits5) – 1);
```

Description:

Extract several bits (1~31 specified by constant value bits5) from the binding value {XRa, XRd} within bit range from bit position rs[4:0] to rs[4:0] + bits5 – 1, the extracting direction is from MSB of XRa to LSB of XRd.

NOTE:

Valid value range of constant bits5 is 1~31, 0 is unpredictable

Example:

```
S32EXTR    XR3, XR2, $5, 23
```

1.8.8 S32EXTRV

Syntax:

```
S32EXTRV       XRa, XRd, rs, rt
```

Operation:

```
position = rs[4:0];
sft5 = rt[4:0];
left = 32 – position;
if (sft5 > left) {
     r_bits = sft5 – left; r_sft = 32 – r_bits;
     tmp = (XRa (<<L) r_bits) | (XRd (>>L) r_sft);
} else {
     r_bits = left – sft5;
     tmp = (XRa (>>L) r_bits);
}
XRa = tmp & ((1<<sft5) – 1);
```

Description:

Extract several bits (1~31 specified by rt[4:0]) from the binding value {XRa, XRd} within bit range from bit position rs[4:0] to rs[4:0] + rt[4:0] – 1, the extracting direction is from MSB of XRa to LSB of XRd.

NOTE:

Valid value range of rt[4:0] is 1~31, 0 is unpredictable

Example:

```
S32EXTRV    XR3, XR2, $5, $1
```

1.9 MAX/MIN

1.9.1 S32MAX

Syntax:

```
S32MAX  XRa, XRb, XRc
```

Operation:

```
If ((signed)XRb > (signed)XRc) XRa = XRb;
Else XRa = XRc;
```

Description:

Compare XRb and XRc to choose maximum one to update XRa.

Example:

```
S32MAX  XR3, XR2, XR1
```

1.9.2 S32MIN

Syntax:

```
S32MIN  XRa, XRb, XRc
```

Operation:

```
If ((signed)XRb < (signed)XRc) XRa = XRb;
Else XRa = XRc;
```

Description:

Compare XRb and XRc to choose minimum one to update XRa.

Example:

```
S32MIN  XR3, XR2, XR1
```

1.9.3 D16MAX

Syntax:

```
D16MAX  XRa, XRb, XRc
```

Operation:

```
If ((signed)XRb[31:16] > (signed)XRc[31:16]) short1 = XRb[31:16];
Else short1 = XRc[31:16]);

If ((signed)XRb[15:0] > (signed)XRc[15:0]) short0 = XRb[15:0];
Else short0 = XRc[15:0]);

XRa = {short1, short0};
```

Description:

Compare two packed signed 16-bit pairs in XRb and XRc to choose two maximum 16-bit values to pack them to XRa.

Example:

```
D16MAX  XR3, XR2, XR1
```

1.9.4 D16MIN

Syntax:

```
D16MIN  XRa, XRb, XRc
```

Operation:

```
If ((signed)XRb[31:16] < (signed)XRc[31:16]) short1 = XRb[31:16];
Else short1 = XRc[31:16]);

If ((signed)XRb[15:0] < (signed)XRc[15:0]) short0 = XRb[15:0];
Else short0 = XRc[15:0]);

XRa = {short1, short0};
```

Description:

Compare two packed signed 16-bit pairs in XRb and XRc to choose two minimum 16-bit values to pack them to XRa.

Example:

```
D16MIN  XR3, XR2, XR1
```

1.9.5 Q8MAX

Syntax:

```
Q8MAX   XRa, XRb, XRc
```

Operation:

```
If ((signed)XRb[31:24] > (signed)XRc[31:24]) byte3 = XRb[31:24];
Else byte3 = XRc[31:24];

If ((signed)XRb[23:16] > (signed)XRc[23:16]) byte2 = XRb[23:16];
Else byte2 = XRc[23:16];

If ((signed)XRb[15:08] > (signed)XRc[15:08]) byte1 = XRb[15:08];
Else byte1 = XRc[15:08];

If ((signed)XRb[07:00] > (signed)XRc[07:00]) byte0 = XRb[07:00];
Else byte0 = XRc[07:00];

XRa = {byte3, byte2, byte1, byte0};
```

Description:

Compare four packed signed byte pairs in XRb and XRc to choose four maximum bytes to pack them to XRa.

Example:

```
Q8MAX   XR3, XR2, XR1
```

1.9.6 Q8MIN

Syntax:

```
Q8MIN   XRa, XRb, XRc
```

Operation:

```
If ((signed)XRb[31:24] < (signed)XRc[31:24]) byte3 = XRb[31:24];
Else byte3 = XRc[31:24];

If ((signed)XRb[23:16] < (signed)XRc[23:16]) byte2 = XRb[23:16];
Else byte2 = XRc[23:16];

If ((signed)XRb[15:08] < (signed)XRc[15:08]) byte1 = XRb[15:08];
Else byte1 = XRc[15:08];

If ((signed)XRb[07:00] < (signed)XRc[07:00]) byte0 = XRb[07:00];
Else byte0 = XRc[07:00];

XRa = {byte3, byte2, byte1, byte0};
```

Description:

Compare four packed signed byte pairs in XRb and XRc to choose four minimum bytes to pack them to XRa.

Example:

```
Q8MIN   XR3, XR2, XR1
```

1.10 Bitwise

1.10.1 S32AND

Syntax:

```
S32AND XRa, XRb, XRc
```

Operation:

```
XRa = XRb & XRc;
```

Description:

Use the result of XRb & XRc to update XRa.

Example:

```
S32AND  XR2, XR5, XR7
```

1.10.2 S32OR

Syntax:

```
S32OR  XRa, XRb, XRc
```

Operation:

```
XRa = XRb | XRc;
```

Description:

Use the result of XRb | XRc to update XRa.

Example:

```
S32OR   XR2, XR5, XR7
```

1.10.3 S32XOR

Syntax:

```
S32XOR XRa, XRb, XRc
```

Operation:

```
XRa = XRb ^ XRc;
```

Description:

Use the result of XRb ^ XRc to update XRa.

Example:

```
S32XOR  XR2, XR5, XR7
```

1.10.4 S32NOR

Syntax:

```
S32NOR XRa, XRb, XRc
```

Operation:

```
XRa = ~(XRb | XRc);
```

Description:

Use the result of ~(XRb | XRc) to update XRa.

Example:

```
S32NOR  XR2, XR5, XR7
```

1.11 Register move between GRF and XRF

1.11.1 S32M2I/S32I2M

Syntax:

```
S32M2I  XRa, rb
S32I2M  XRa, rb
```

Operation:

```
S32M2I:
rb = XRa;

S32I2M:
XRa = rb;
```

Description:

Move value between GRF of IU and XRF of MXU.

Example:

```
S32M2I  XR2, $3
```

NOTE: To access MXU_CR, XR16 is used. For example

```
S32M2I  XR16, $1      ; read content of MXU_CR to $1
ORI     $1, $1, 1
S32I2M  XR16, $1      ; enable MXU
```

Moreover, when MXU is toggled from disable to enable, there are at least **three** non-MXU instructions necessary between the instruction S32I2M (which enables MXU) and following available MXU instructions.

1.12 Miscellaneous

1.12.1 S32SFL

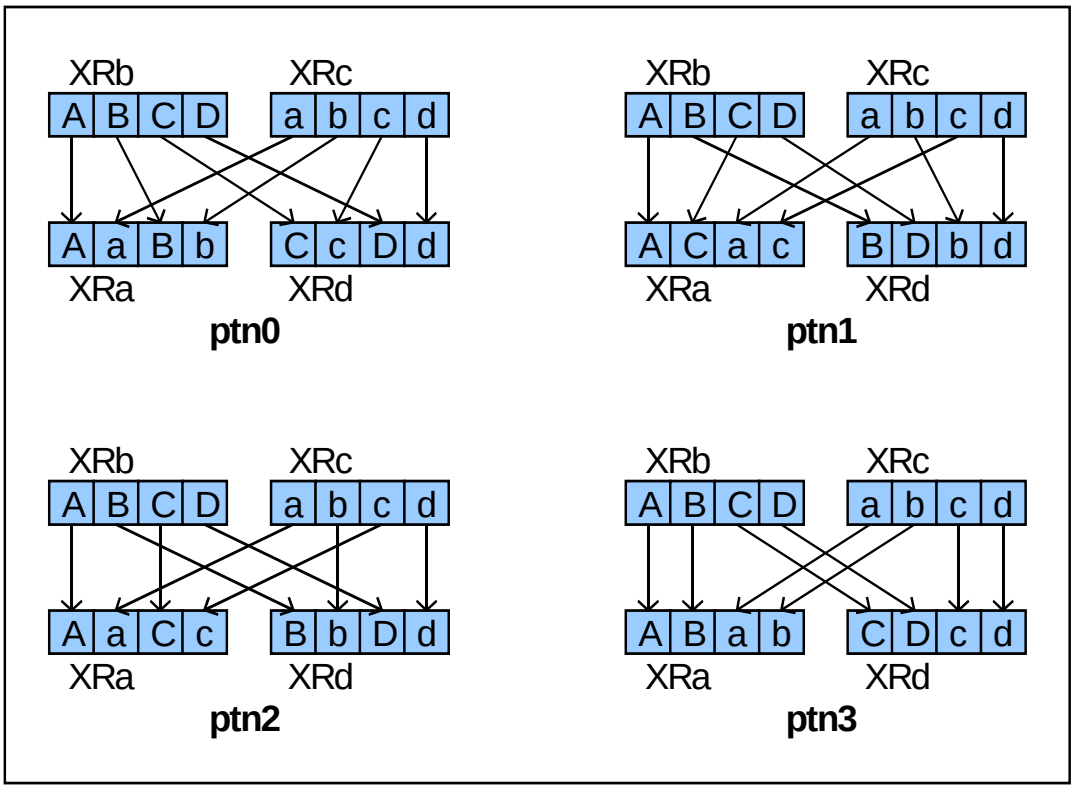
Syntax:

```
S32SFL  XRa, XRb, XRc, XRd, OPTN2
```

Parameter:

```
OPTN2: ptn0(00), ptn1(01), ptn2(10), ptn3(11)
```

Operation:



Description:

As in the figure above.

Example:

```
S32SFL  XR3, XR2, XR1, XR5, 1
S32SFL  XR3, XR2, XR1, XR5, ptn0
```

1.12.2 S32ALN/S32ALNI

Syntax:

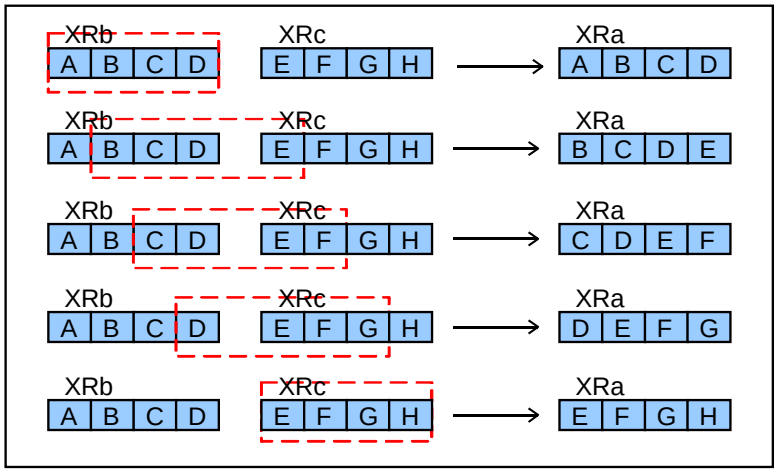
```
S32ALN  XRa, XRb, XRc, rs
S32ALNI XRa, XRb, XRc, OPTN3
```

Parameter:

```
OPTN3: ptn0(000), ptn1(001), ptn2(010), ptn3(011), ptn4(100),
101~111 are reserved
```

Operation:

```
Switch (rs[2:0]|OPTN3) {
     case 0:    XRa = {XRb[31:0]};
     case 1:    XRa = {XRb[23:0], XRc[31:24]};
     case 2:    XRa = {XRb[15:0], XRc[31:16]};
     case 3:    XRa = {XRb[07:0], XRc[31:08]};
     case 4:    XRa = {XRc[31:0]};
     default:   Undefined
}
```



Description:

Concatenate unaligned value of XRb and unaligned value of XRc to form an aligned word value to update XRa

Example:

```
S32ALN  XR3, XR2, XR1, $5
S32ALNI XR5, XR1, XR7, 4
S32ALNI XR5, XR1, XR7, ptn2
```

1.12.3 Q16SAT

Syntax:

```
Q16SAT  XRa, XRb, XRc
```

Operation:

```
byte3 = sat_8((signed)XRb[31:16]);
byte2 = sat_8((signed)XRb[15:00]);
byte1 = sat_8((signed)XRc[31:16]);
byte0 = sat_8((signed)XRc[15:00]);
XRa = {byte3, byte2, byte1, byte0};
```

Description:

XRa is updated by four packed 8-bit saturation results (0 ~ 255) calclulated from four 16-bit signed values of XRb and XRc.

Example:

```
Q16SAT  XR3, XR2, XR1
```

1.12.4 Q16SCOP

Syntax:

```
Q16SCOP XRa, XRb, XRc, XRd
```

Operation:

```
short3 = XRb[31:16]<0? 0xFFFF : (XRb[31:16]>0? 1 : 0);
short2 = XRb[15:00]<0? 0xFFFF : (XRb[15:00]>0? 1 : 0);
short1 = XRc[31:16]<0? 0xFFFF : (XRc[31:16]>0? 1 : 0);
short0 = XRc[15:00]<0? 0xFFFF : (XRc[15:00]>0? 1 : 0);
XRa = {short3, short2};
XRd = {short1, short0};
```

Description:

Determine the 16-bit value range, the determination is as follow:
range = x<0? –1 : (x > 0?) 1 : 0;

Example:

```
Q16SCOP    XR3, XR2, XR1, XR5
```

1.12.5 S32LUI

Syntax:

```
S32LUI XRa, S8, OPTN3
```

Parameter:

```
OPTN3: ptn0(000), ptn1(001), ptn2(010), ptn3(011), ptn4(100),
       ptn5(101), ptn6(110), ptn7(111)
```

Operation:

```
tmp8 = S8;
switch(OPTN3) {
case 0: XRa = {24'b0, tmp8};
case 1: XRa = {16'b0, tmp8, 8'b0};
case 2: XRa = {8'b0, tmp8, 16'b0};
case 3: XRa = {tmp8, 24'b0};
case 4: XRa = {8'b0, tmp8, 8'b0, tmp8};
case 5: XRa = {tmp8, 8'b0, tmp8, 8'b0};
case 6: XRa = {{8{sign of tmp8}}, tmp8, {8{sign of tmp8}}, tmp8};
case 7: XRa = {tmp8, tmp8, tmp8, tmp8};
}
```

Description:

Permutate the immediate value (S8) in terms of specific pattern to form a word to update XRa

Example:

```
S32LUI  XR11, 0x7F, 6
S32LUI  XR11, 0xFF, ptn4
```