The RISC-V IME Set Specification

SpacemiT

Version 20240422

Table of Contents

Preamble

This document is in the Development state

Assume everything can change, before being accepted as standard.

Copyright and license information

Chapter 1. Introduction

For convolution and matrix multiplication, which account for the highest proportion of computing power in AI applications, dedicated acceleration instructions have been added based on the RISCV Vector 1.0 standard instructions. Following the RISC-V IME extension standard, and reusing Vector register resources, these instructions can bring more than a tenfold performance improvement to AI applications at a very small hardware cost.

Chapter 2. Programmer's Model

This extension is based on the standard RISCV vector registers and does not add any state control registers and matrix registers, ensuring that the programming model remains as similar as possible to the standard RVV programming model. It includes the usage of rounding modes, configuration instructions such as vsetvli/vsetivli/vsetvl, etc. Unlike the standard RISCV Vector 1.0, the matrix extension instructions will select the corresponding matrix multiply and accumulate (MAC) unit based on the vl and SEW values obtained from the configuration instructions.

This extension instructions only support cases where LMUL is less than or equal to 1.

The VLEN supported by this extension is from 128 to 4096, while SEW is only supported 4, 8 and 16. And a new variable called *Copy* is introduced. It refers to the number of copies of the MAC unit. In other words, when *copy* is 1, one instruction can only perform one MAC computation for a pair of input data. And when *copy* is 2, one instruction can perform two MAC computation for two pairs of input data.

The MAC (M*N*K[x *Copies*]) unit corresponding to different vl and SEW are as follows:

Table 1. MAC unit for different vl and SEW

vl*SEW	128	256	512	1024	2048	4096
SEW=4	2x2x8[x2]	4x4x16	4x4x16[x2]	8x8x32	8x8x32[x2]	16x16x64
SEW=8	2x2x4[x2]	4x4x8	4x4x8[x2]	8x8x16	8x8x16[x2]	16x16x32
SEW=16	2x2x2[x2]	4x4x4	4x4x4[x2]	8x8x8	8x8x8[x2]	16x16x16

For different VLEN of different hardware, the supported vl configuration are in the following table.

Table 2. Vl supported for different VLEN

VLEN	128	256	512	1024	2048	4096
vl*SEW	128	128/256	128/256/512	128/256/512/1024	128/256/512/1024/2048	28/256/512/1024/2048/4096

For example, when VLEN is 1024 and SEW is 8, the selection of the MAC unit:

```
Example of selection of MAC unit:

let VLEN = 1024, then:

 vsetvli t0, x0, e8, m1, ta, ma   # select the 8x8x16 MAC unit, whose SEW is 8.

 li      t0, 256
 vsetvli t0, t0, e8, m1, ta, ma   # select the 4x4x8 MAC unit, whose SEW is 8
```

As mentioned before, LMUL is only support less than or equal to 1. The max MAC unit implemented by hardware should not exceed the configuration of vl*SEW equalling to VLEN. For example, when VLEN is 1024 and SEW is 8, the hardware can select the MAC units from 2x2x4[x2], 4x4x8, 4x4x8[x2] and 8x8x16.

When the MAC unit select by configure instruction is not support by the hardware, an illegal instruction will occur during execution. And the hardware need not to check the overlap of vd.

Chapter 3. Instruction description

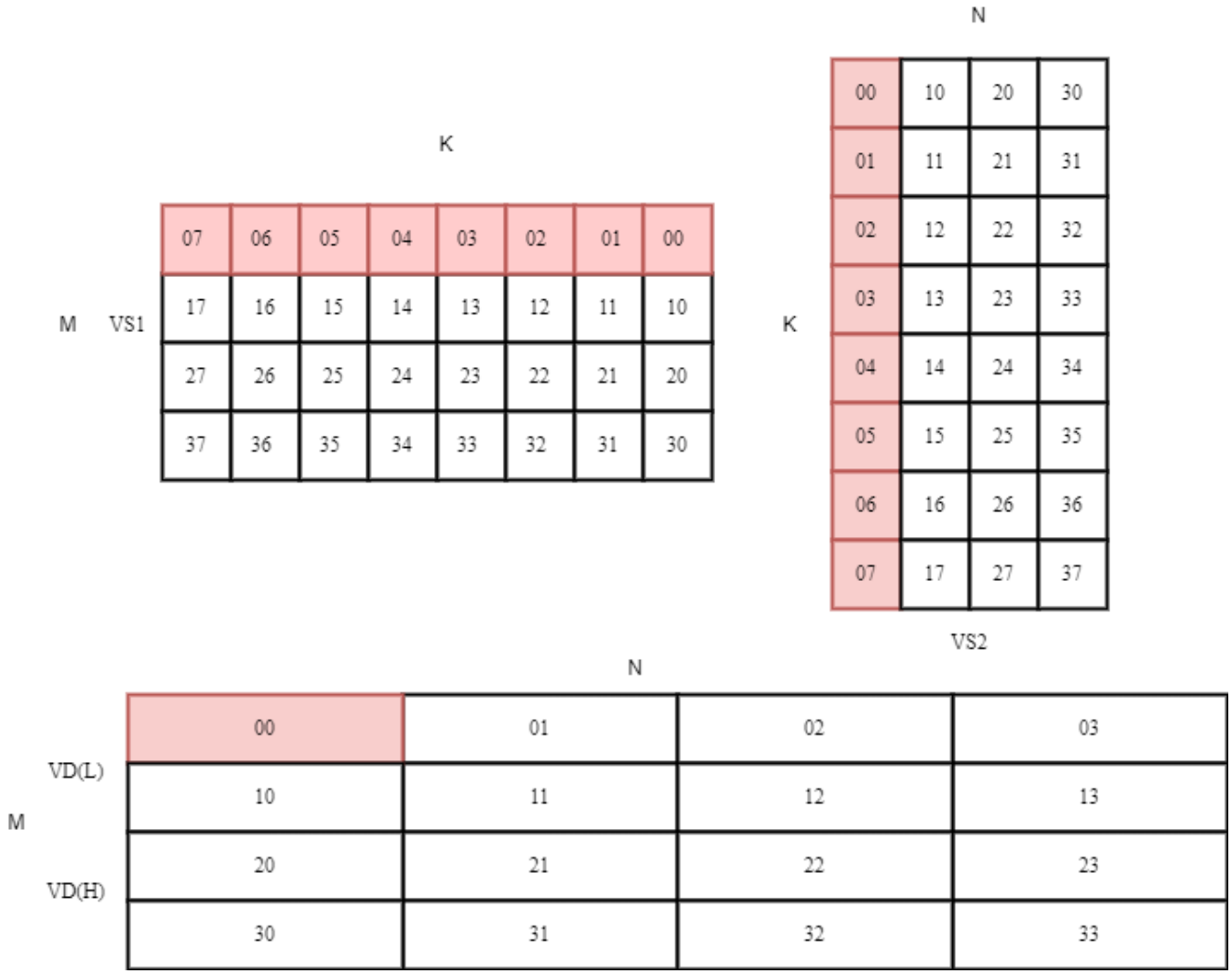
AI extension instructions are divided into two categories based on their functionality: dot-product matrix multiply-accumulate instructions and sliding-window dot-product instructions. Different values of VLEN and SEW correspond to different MAC uints.

3.1. Dot-product matrix multiply-accumulate instructions

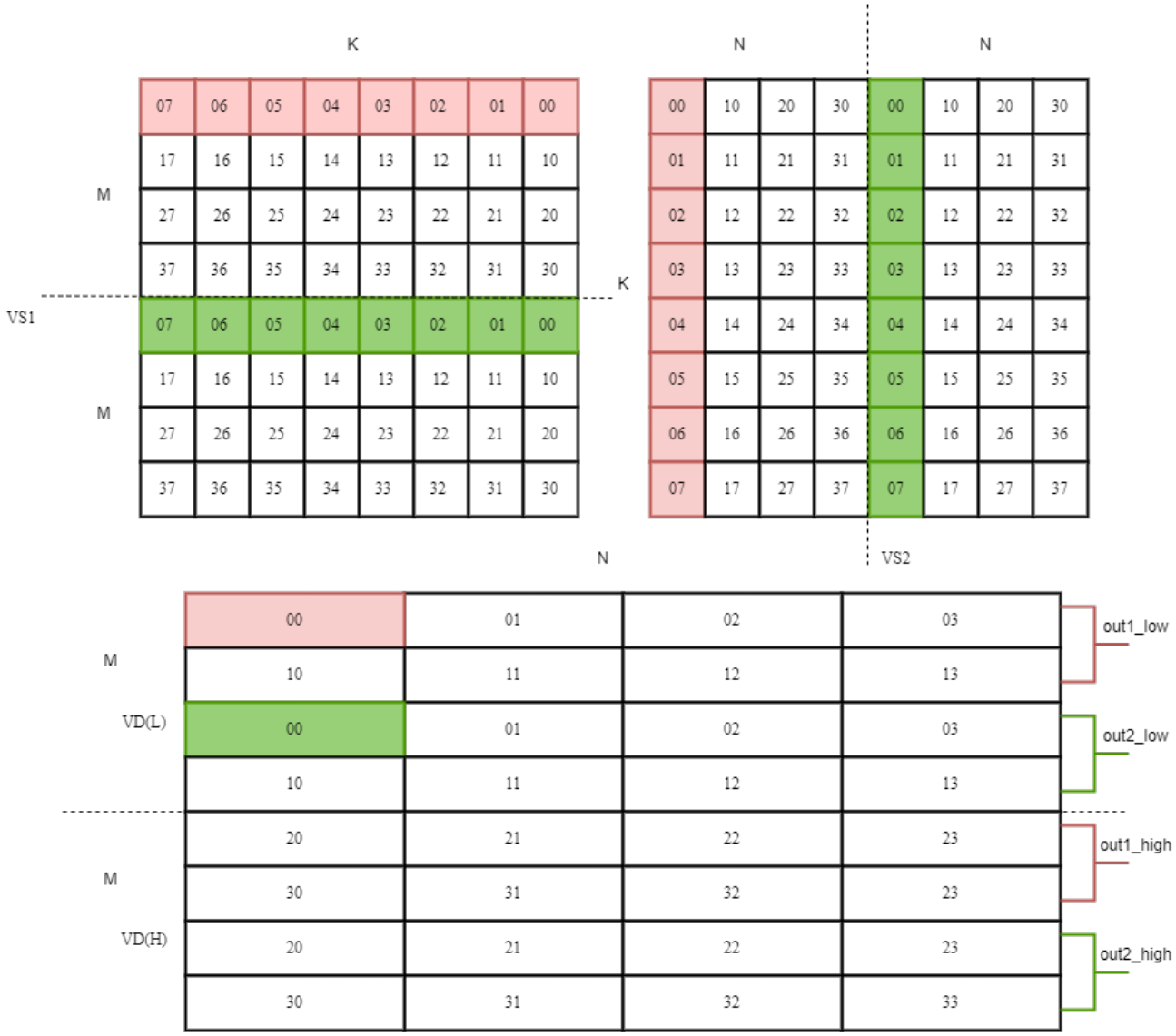
Dot-product multiply-accumulate instructions can be categorized into two types: integer dot-product matrix multiply-accumulate instructions and floating-point dot-product matrix multiply-accumulate instructions. During matirx multiplication and accumulating computation, the layout of input and output data in registers is considered as two-dimensional data. The layout of input of integer instructions and floating-point instructions is identical, but layout of output is different. The integer instructions require two registers to store thire output, but the floating-point instructions only require one.

3.1.1. Integer dot-product matrix multiply-accumulate instructions

Let's take a closer look at the layout of dot-product matrix multiply-accumulate instructions. The input A in VS1 is treated as *copies**(M, K) matrices, while the input B in VS2 is treated as *copies**(K, N). The result C will be stored in two sequential registers (the index of VD(L) must be even), and its elements will be treated as *copies**(M, N) matrices. For example, when the VLEN=256 and SEW=8, the layout is as described below:



At this time, assume the max MAC unit (4x4x8) has been implemented. Then the input matrix A is treated as a (4, 8) matrix and B is treated as a (8, 4) matrix. After matrix multiplication, the resulting matrix C is of size (4, 4), whose data type is int32. When VLEN=512 and SEW=8, the *copy* is 2, the layout can be as stated as follows.



Input data is stored in VS1 and VS1+1 (the index of VS1 must be even), and the elements of matrix A are sliding-packing from the two (2*4,8) matrices in VS1 and VS1+1. Input B is placed in VS2, with its elements viewed as two (8,4) matrices. The result C consists of two (M,N) matrices, they are stored in two sequential registers (the index of VD(L) must be even).

The integer dot-product matrix multiply-accumulate instructions take matrix A and matrix B as input, and accumulate the multiplication result of A and B into C.

- shape of A: M rows, K columns [x Copies]
- shape of B: K rows, N columns [x Copies]
- shape of C: M rows, N columns [x Copies]

> *The value of Copies can be either 1 or 2, Copies=(sqrt(VLEN/64) == floor(sqrt(VLEN/64)) ? 1 : 2)*

As mentioned before, this instructions function can be descripted as follows:

```
Copies=(sqrt(VLEN/64) == floor(sqrt(VLEN/64)) ? 1 : 2)
for (cp = 0; cp < Copies; cp++) {
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j ++) {
            for (k = 0; k < K; k++) {
                C[cp * M * N + i * N + j] +=
                    int32(A[cp * M * K + i * N + k] * B[cp * K *N + k * N + j]);
                }
             }
         }
     }
 }
```

The data type of supported by the the integer dot-product matrix multiply-accumulate instructions is in the following table.

Table 3. Data type of vmadot

instructions	Operand Type A	Operand Type B	Accumulator Type C
vmadot	int4/int8/int16	int4/int8/int16	int32/int32/fp32
vmadotu	uint4/uint8/uint16	uint4/uint8/uint16	int32/int32/fp32
vmadotsu	int4/int8/int16	uint4/uint8/uint16	int32/int32/fp32
vmadotus	uint4/uint8/uint16	int4/int8/int16	int32/int32/fp32

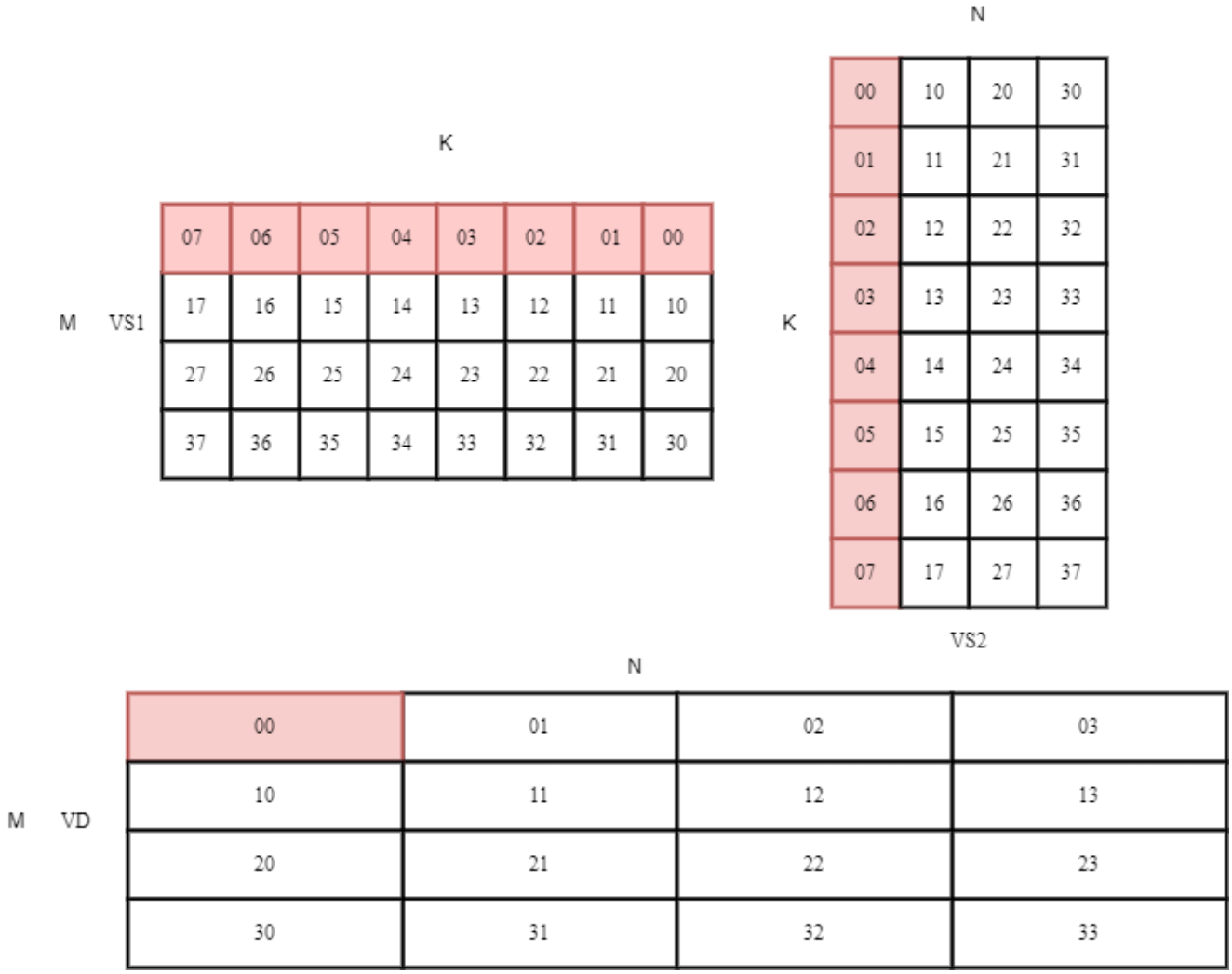
And the usage of the instructions:

```
# vs1 refers to input A, vs2 refers to input B, and vd refers to output C.

vmadot      vd,     vs1,    vs2
vmadotu     vd,     vs1,    vs2
vmadotsu    vd,     vs1,    vs2
vmadotus    vd,     vs1,    vs2
```

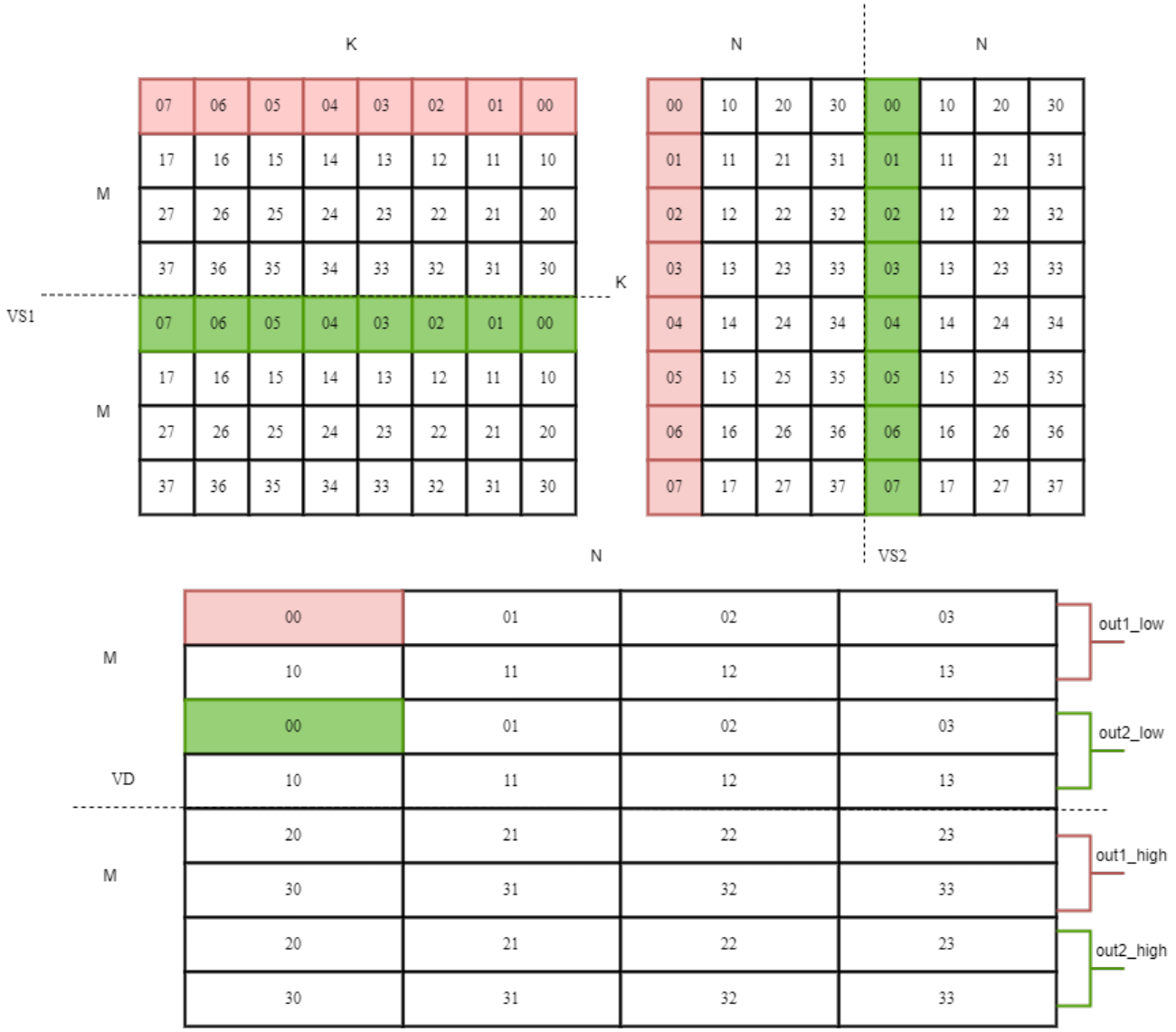
the index of vd must be even.

3.1.2. Floating point dot-product matrix multiply-accumulate instructions

Same as the integer dot-product matrix multiply-accumulate instructions, the floating point dot-product matrix multiply-accumulate also support *copy*=1 and *copy*=2. The input A in VS1 is considered as *copies**(M, K) matrices, while the input B in VS2 is considered as *copies**(K, N). The result C will be stored in two sequential registers (the index of VD(L) must be even), and its elements will be considered as *copies**(M, N) matrices,. For example, when the VLEN=256 and SEW=8, the layout is as follows:



At this time, assume the max MAC unit (4x4x8) has been implemented. Then the input matrix A is considered as a (4, 8) matrix and B is considered as a (8, 4) matrix. After matrix multiplication, the resulting matrix C is of size (4, 4), whose data type is fp16 or bfp16. When the VLEN=512 and SEW=8, *copy* is 2, the layout is shown as follows.



Input data is stored in VS1 and VS1+1 (the index of VS1 must be even), and the elements of matrix A are sliding-packing from the two (2*4,8) matrices in VS1 and VS1+1. Input B is stored in VS2, with its elements considered as two (8,4) matrices. The result C consists of two (M,N) matrices, they are stored in one registers.

The floating point dot-product matrix multiply-accumulate instructions take matrix A and matrix B as input, and accumulate the multiplication result of A and B into C.

- shape of A: M rows, K columns [x Copies]
- shape of B: K rows, N columns [x Copies]
- shape of C: M rows, N columns [x Copies]

The value of Copies can be either 1 or 2, Copies=(sqrt(VLEN/64) == floor(sqrt(VLEN/64)) ? 1 : 2)

The function is the same as the integer one.

```
assume FP = fp16 or bfp16
Copies=(sqrt(VLEN/64) == floor(sqrt(VLEN/64)) ? 1 : 2)
for (cp = 0; cp < Copies; cp++) {
    for (i = 0; i < M; i++) {
```

```
        for (j = 0; j < N; j ++) {
            for (k = 0; k < K; k++) {
                C[cp * M * N + i * N + j] +=
                    FP(A[cp * M * K + i * N + k] * B[cp * K *N + k * N + j]);
                }
             }
         }
     }
 }
```

The data type of supported by the the floating-point dot-product matrix multiply-accumulate instructions is in the following table.

Table 4. Data type of vfmadot

instructions	Operand Type A	Operand Type B	Accumulator Type C
vfmadot	fp4/fp8/fp16/bfp16	fp4/fp8/fp16/bfp16	fp16/fp16/fp16/bfp16

And the usage of the instructions:

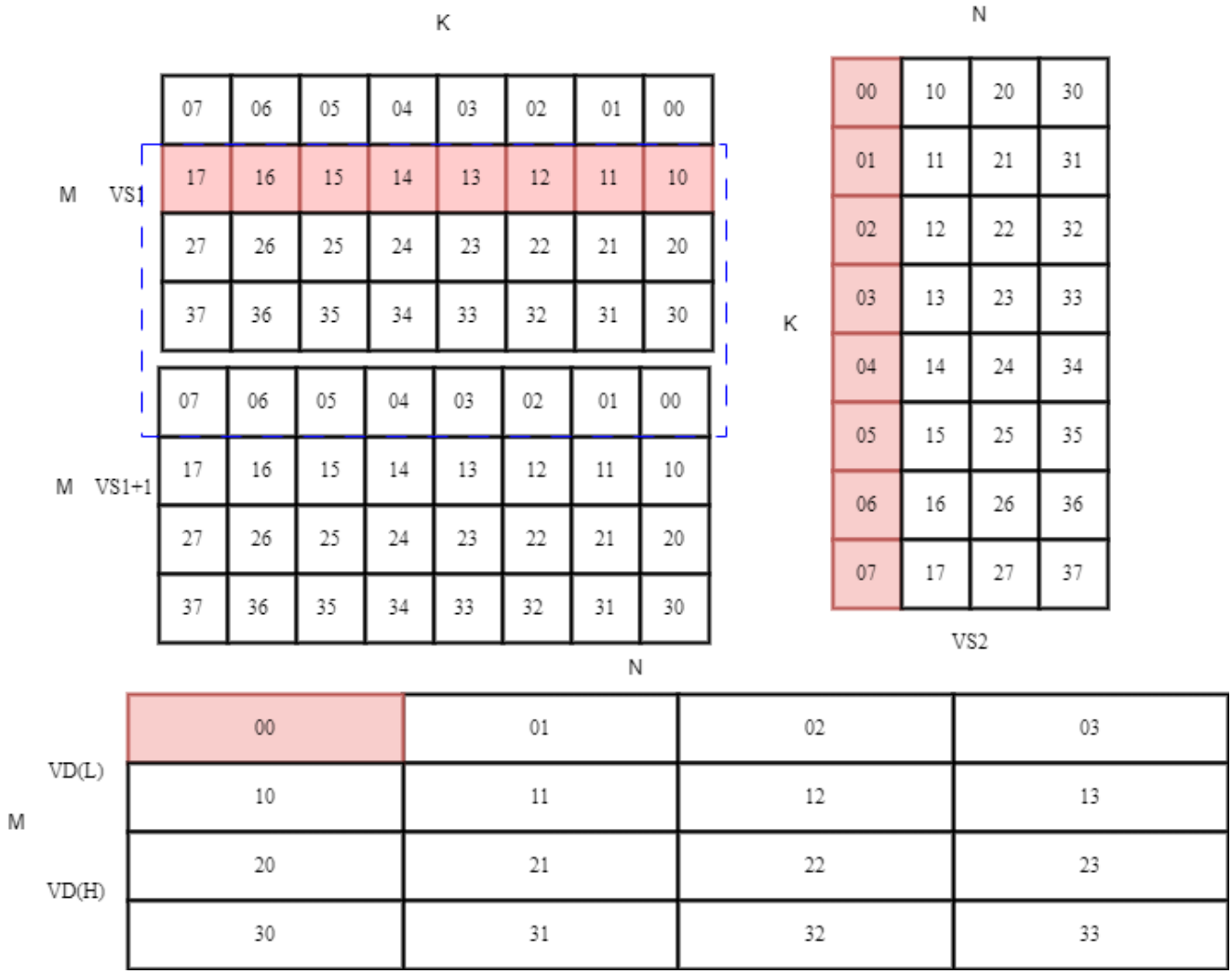
```
# vs1 refers to input A, vs2 refers to input B, and vd refers to output C.

vfmadot     vd,     vs1,    vs2
```

3.2. Sliding-window dot-product matrix multiply-accumulate instruction

The sliding-window dot-product matrix multiply-accumulate instructions can select specified values from two sequential registers, VS1 and VS1+1 to be used as matrix A. The data mode of data of matrix B and Matrix C are the same as the instructions without slide. Depending on the data type, it is further categorised into integer and floating-point types.

3.2.1. Integer sliding-window dot-product matrix multiply-accumulate instruction

During matrix multiply computation, the layout of input and output data in the registers also can be divided into two situations. The input A will be selected from VS1 and VS1+1 (the index of VS1 must be even), and the selected elements will be considered as *copies* * (M,K) matrices. Input B is stored in VS2, with elements considered as *copies* * (K,N) matrices. The result C will be stored in two sequential registers (the index of VD(L) must be even), with elements considered as *copies* * (M,N) matrices. For example, when the VLEN=256 and SEW=8, the configuration would be as illustrated in the following diagram:



Elements of A are selected from a (2*4,8) matrix formed by combining VS1 and VS1+1, and matching elements are selected through a specified sliding value. As demonstrated previously, with a slide value of 1, the blue frame slides down 8(1*K) elements. The resulting values form an (4,8) matrix, which serves as input matrix A for the matrix multiply-accumulate calculation. For matrix B and C, are the same as the instructions without slideing.

In the case of VLEN=512, SEW=8, where *copy* is set to 2, the layout would be as shown in the following diagram:

The elements in VS1 and VS1+1 can be considered as two (2*4, 8) matrices. Then, in the same manner as with *copy*=1, the matching elements are selected using a specified slide value. The data obtained from the slide are two (M,K) matrices, which serve as input matrices A for the matrix multiply-accumulate computation.

Sliding-window dot-product matrix multiply-accumulate instructions also take matrix A and matrix B as input, and accumulate the multiplication result of A and B into C.

- shape of A: M rows, K columns [x Copies]
- shape of B: K rows, N columns [x Copies]
- shape of C: M rows, N columns [x Copies]

The function description:

```
Copies=(sqrt(VLEN/64) == floor(sqrt(VLEN/64)) ? 1 : 2)
for (cp = 0; cp < Copies; cp++) {
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j ++) {
            for (k = 0; k < K; k++) {
                C[cp * M * N + i * N + j] +=
                    int32(A[cp * M * K + slide * K + i * N + k] * B[cp * K *N + k * N
+ j]);
            }
         }
      }
   }
```

```
}
```

The data type of supported by the the integer sliding-window dot-product matrix multiply-accumulate instructions is in the following table.

Table 5. Data type of vmadot-x

category	instructions	Operand Type A	Operand Type B	Accumulator Type C
slide-1	vmadot1	int4/int8/int16	int4/int8/int16	int32/int32/fp32
	vmadot1u	uint4/uint8/uint16	uint4/uint8/uint16	int32/int32/fp32
	vmadot1su	int4/int8/int16	uint4/uint8/uint16	int32/int32/fp32
	vmadot1us	uint4/uint8/uint16	int4/int8/int16	int32/int32/fp32
slide-2	vmadot2	int4/int8/int16	int4/int8/int16	int32/int32/fp32
	vmadot2u	uint4/uint8/uint16	uint4/uint8/uint16	int32/int32/fp32
	vmadot2su	int4/int8/int16	uint4/uint8/uint16	int32/int32/fp32
	vmadot2us	uint4/uint8/uint16	int4/int8/int16	int32/int32/fp32
slide-3	vmadot3	int4/int8/int16	int4/int8/int16	int32/int32/fp32
	vmadot3u	uint4/uint8/uint16	uint4/uint8/uint16	int32/int32/fp32
	vmadot3su	int4/int8/int16	uint4/uint8/uint16	int32/int32/fp32
	vmadot3us	uint4/uint8/uint16	int4/int8/int16	int32/int32/fp32
slide-n	vmadotn	int4/int8/int16	int4/int8/int16	int32/int32/fp32
	vmadotnu	uint4/uint8/uint16	uint4/uint8/uint16	int32/int32/fp32
	vmadotnsu	int4/int8/int16	uint4/uint8/uint16	int32/int32/fp32
	vmadotnus	uint4/uint8/uint16	int4/int8/int16	int32/int32/fp32

And the usage of the instructions:

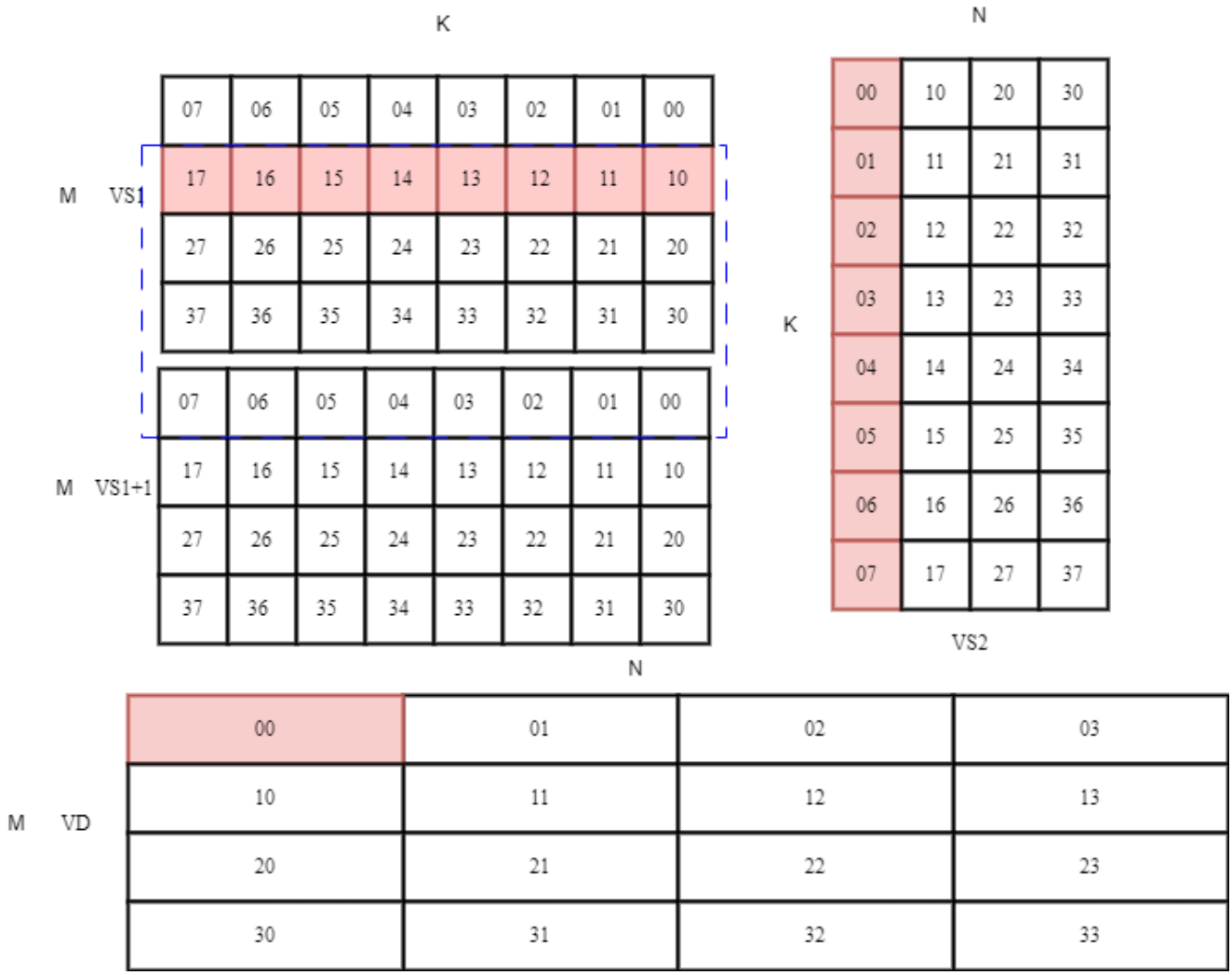
```
# vs1 refers to input A, vs2 refers to input B, and vd refers to output C.

# slide 1
vmadot1      vd,     vs1,    vs2
vmadot1u     vd,     vs1,    vs2
vmadot1su    vd,     vs1,    vs2
vmadot1us    vd,     vs1,    vs2

# slide 2
vmadot2      vd,     vs1,    vs2
vmadot2u     vd,     vs1,    vs2
vmadot2su    vd,     vs1,    vs2
vmadot2us    vd,     vs1,    vs2

# slide 3
vmadot3      vd,     vs1,    vs2
vmadot3u     vd,     vs1,    vs2
vmadot3su    vd,     vs1,    vs2
vmadot3us    vd,     vs1,    vs2

# slide 4
vmadotn      vd,     vs1,    vs2,  t0
```

```
vmadotnu     vd,     vs1,     vs2,  t0
vmadotnsu    vd,     vs1,     vs2,  t0
vmadotnus    vd,     vs1,     vs2,  t0
```

the index of vd and vs1 must be even, and the slide value only support place in t0

3.2.2. Floating point sliding-window dot-product matrix multiply-accumulate instruction

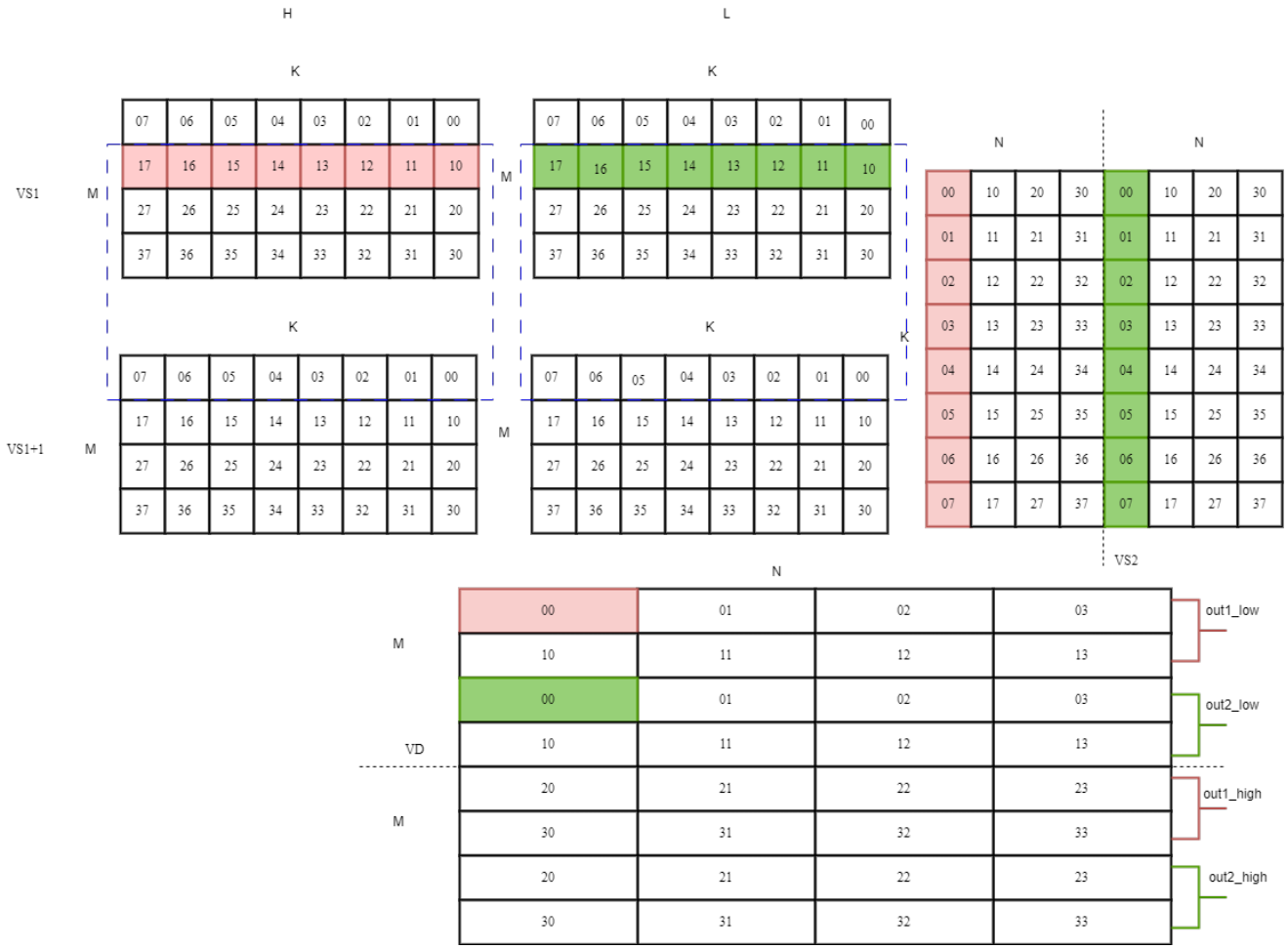
Same as the integer one, the layout of input and output data in the registers also can be divided into two situations. The input A will be selected from VS1 and VS1+1 (the index of VS1 must be even), and the selected elements will be considered as *copies* * (M,K) matrices. Input B is stored in VS2, with elements considered as *copies* * (K,N) matrices. The result C will be stored in one register, with elements considered as *copies* * (M,N) matrices. For example, when the VLEN=256 and SEW=8, the layout would be as illustrated in the following diagram:



Elements of A are selected from a (2*4,8) matrix formed by combining VS1 and VS1+1, and the matching elements are selected through a specified sliding value. As demonstrated previously, with a slide value of 1, the blue frame slides down 8(1*K) elements. The resulting values form an (4,8) matrix, which serves as input matrix A for the matrix multiply-accumulate calculation. For matrix B and C, are the same as the instructions without slideing.

In the case of VLEN=512, SEW=8, where *copy* is set to 2, the layout would be as shown in the following

diagram:



The elements in VS1 and VS1+1 can be considered as two (2*4, 8) matrices. Then, in the same manner as with *copy*=1, the matching elements are selected using a specified slide value. The data obtained from the slide are two (M,K) matrices, which serve as input matrices A for the matrix multiply-accumulate computation.

Sliding-window dot-product matrix multiply-accumulate instructions also take matrix A and matrix B as input, and accumulate the multiplication result of A and B into C.

- shape of A: M rows, K columns [x Copies]
- shape of B: K rows, N columns [x Copies]
- shape of C: M rows, N columns [x Copies]

The function description:

```
assume FP = fp16 or bfp16
Copies=(sqrt(VLEN/64) == floor(sqrt(VLEN/64)) ? 1 : 2)
for (cp = 0; cp < Copies; cp++) {
    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j ++) {
            for (k = 0; k < K; k++) {
                C[cp * M * N + i * N + j] +=
                    FP(A[cp * M * K + slide * K + i * N + k] * B[cp * K *N + k * N +
j]);
            }
```

```
            }
        }
    }
}
```

The data type of supported by the the floating-point sliding-window dot-product matrix multiply-accumulate instructions is in the following table.

Table 6. Data type of vfmadot-x

category	instructions	Operand Type A	Operand Type B	Accumulator Type C
slide-1	vfmadot1	fp4/fp8/fp16/bfp16	fp4/fp8/fp16/bfp16	fp16/fp16/fp16/bfp16
slide-2	vfmadot2	fp4/fp8/fp16/bfp16	fp4/fp8/fp16/bfp16	fp16/fp16/fp16/bfp16
slide-3	vfmadot3	fp4/fp8/fp16/bfp16	fp4/fp8/fp16/bfp16	fp16/fp16/fp16/bfp16
slide-n	vfmadotn	fp4/fp8/fp16/bfp16	fp4/fp8/fp16/bfp16	fp16/fp16/fp16/bfp16

And the usage of the instructions:

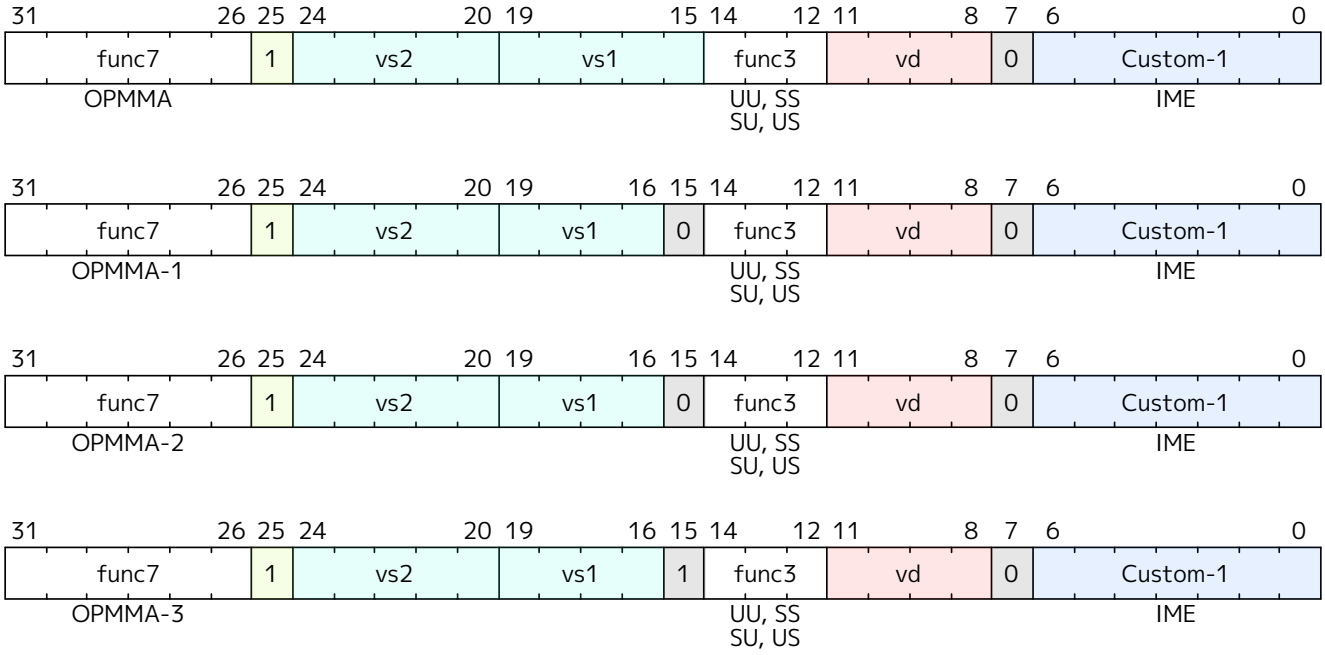
```
# vs1 refers to input A, vs2 refers to input B, and vd refers to output C.

# slide 1
vfmadot1      vd,     vs1,    vs2

# slide 2
vfmadot2      vd,     vs1,    vs2

# slide 3
vfmadot3      vd,     vs1,    vs2

# slide 4
vfmadotn      vd,     vs1,    vs2,  t0
```

the index of vs1 must be even, and the slide value only support place in t0

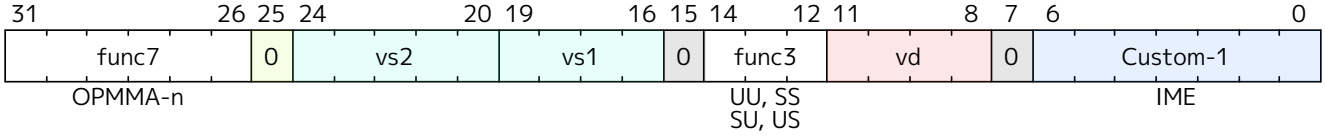
Chapter 4. Instruction fromat

Formats for matrix extention instructions is under OPMMA and OPFMMA major opcode

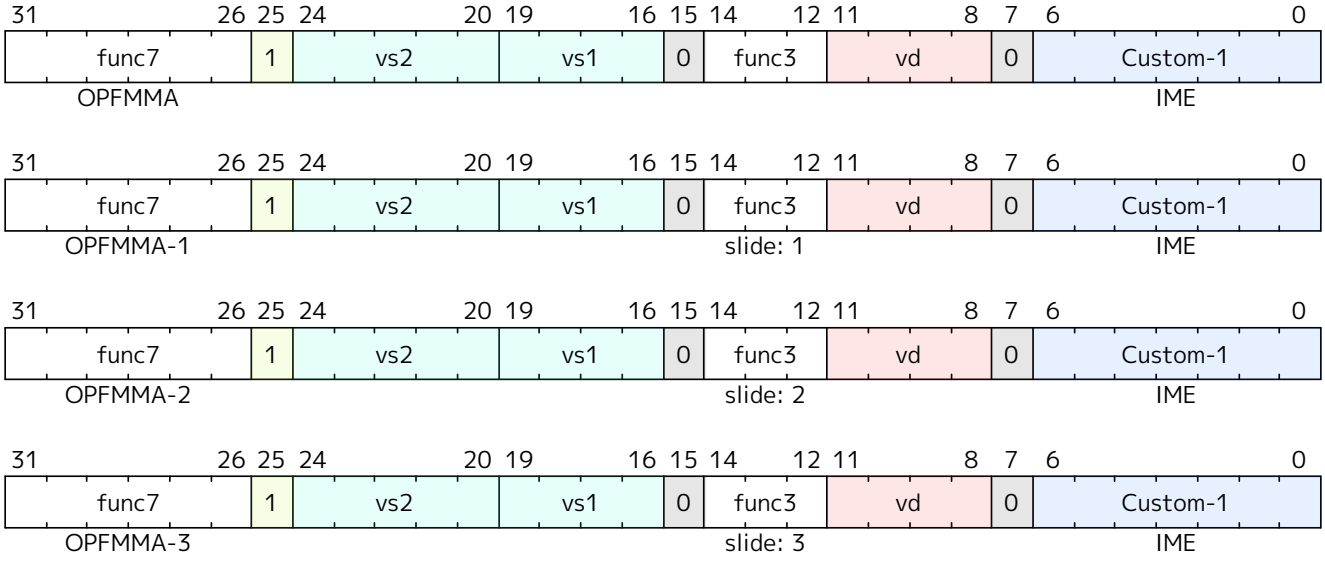
For integr extension, the func3 field is used to distinguish the difference of input data type. UU denotes unsigned integer multiply with unsigned integer, US represents unsigned integer multiply with signed integer, SU indicates signed integer multiply with unsigned integer, SS signifies signed integer multiply with signed integer.



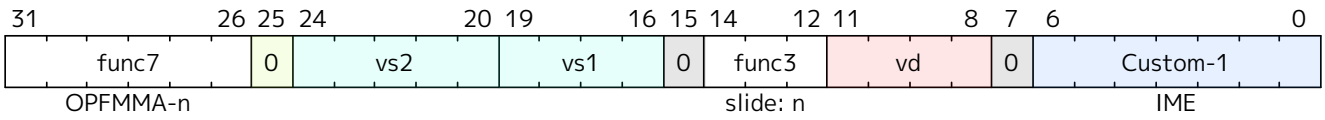
The vm field is used to indicate the instructions with n sliding value.



For floating-point extension, the func3 field is used to distinguish the difference of sliding value.



The vm field is also used to indicate the instructions with n sliding value.


31
26
25
24
20
19
16
15
14
12
11
8
7
6
0
func7
0
vs2
vs1
0
func3
vd
0
Custom-1
OPFMMA-n
slide: n
IME

Chapter 5. Instruction list

Integer					Float				
funct7					funct7				
OPMMA-0	uu	ss	us	su	OPFMMA-0	ff			
OPMMA-1	uu	ss	us	su	OPFMMA-1	ff			
OPMMA-2	uu	ss	us	su	OPFMMA-2	ff			
OPMMA-3	uu	ss	us	su	OPFMMA-3	ff			
OPMMA-n	uu	ss	us	su	OPFMMA-n	ff			

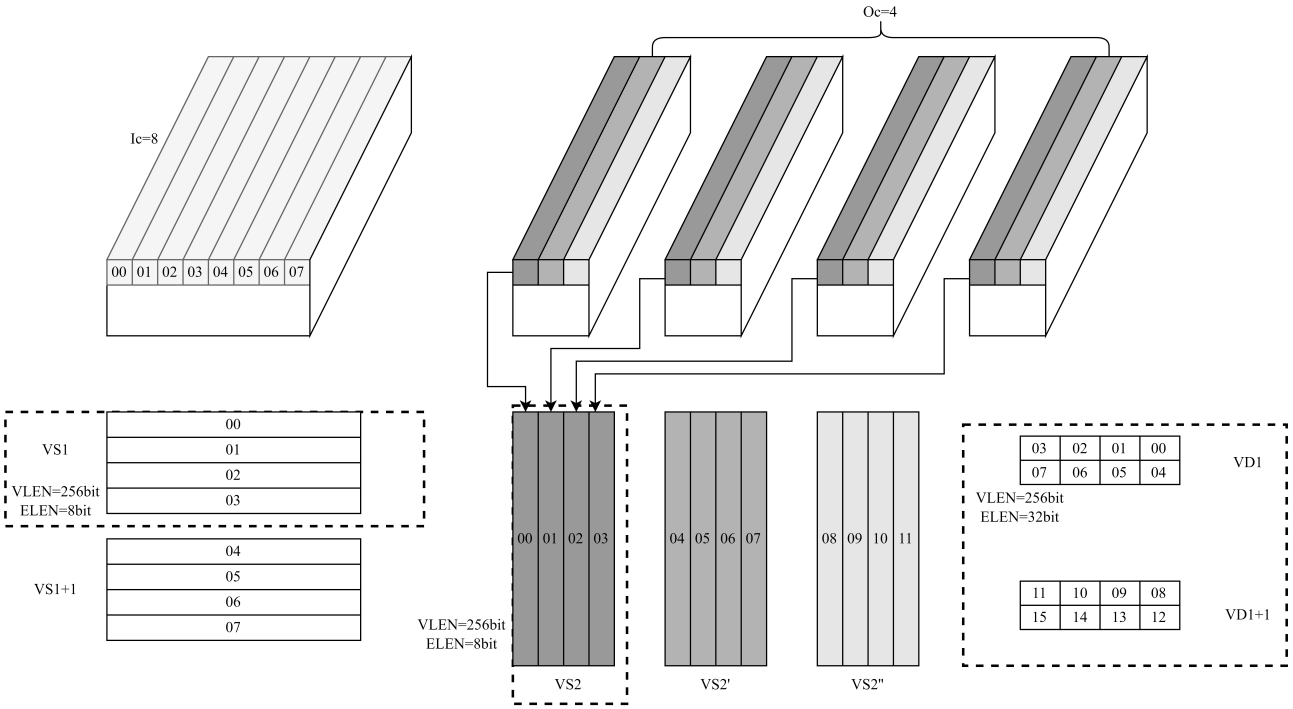
funct7						funct7					
111000	uu	ss	us	su	vmadot	111010	ff				vfmadot
111001	uu	ss	us	su	vmadot1	111010	ff				vfmadot1
111001	uu	ss	us	su	vmadot2	111010	ff				vfmadot2
111001	uu	ss	us	su	vmadot3	111010	ff				vfmadot3
111001	uu	ss	us	su	vmadotn	111010	ff				vfmadotn

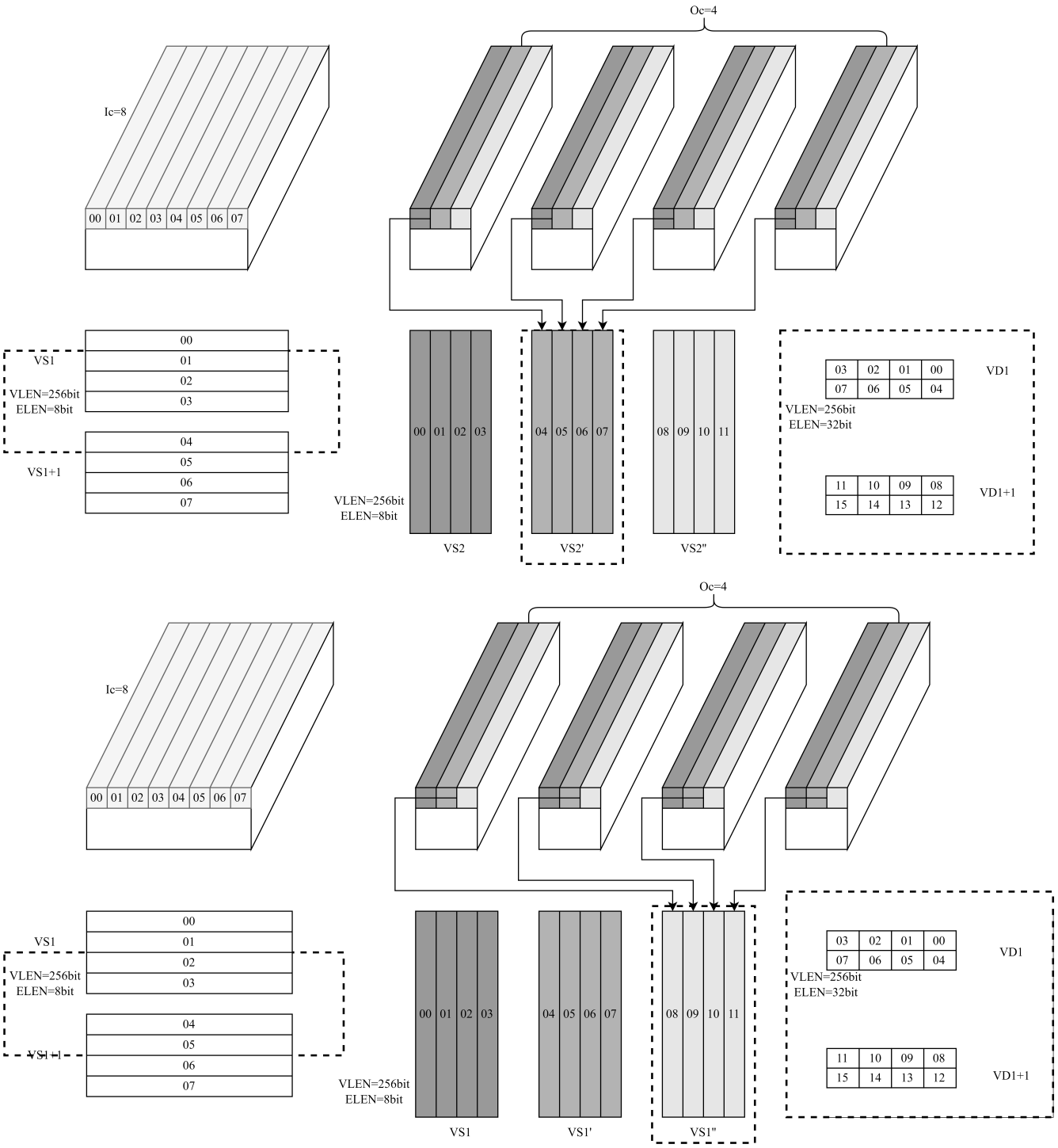
Chapter 6. Example of convolution compute

The computation of 2D convolution can be accelerated by utilising dot-product matrix multiply-accumulate instructions and sliding-window dot-product matrix multiply-accumulate instructions. Giving an input feature map size of 1x3x8x8(NHWC), kernel size of 3x3, with stride = 1, padding = 0, and an output channel number = 4, the size of the output feature map will be 1x4x6x6(HWC).

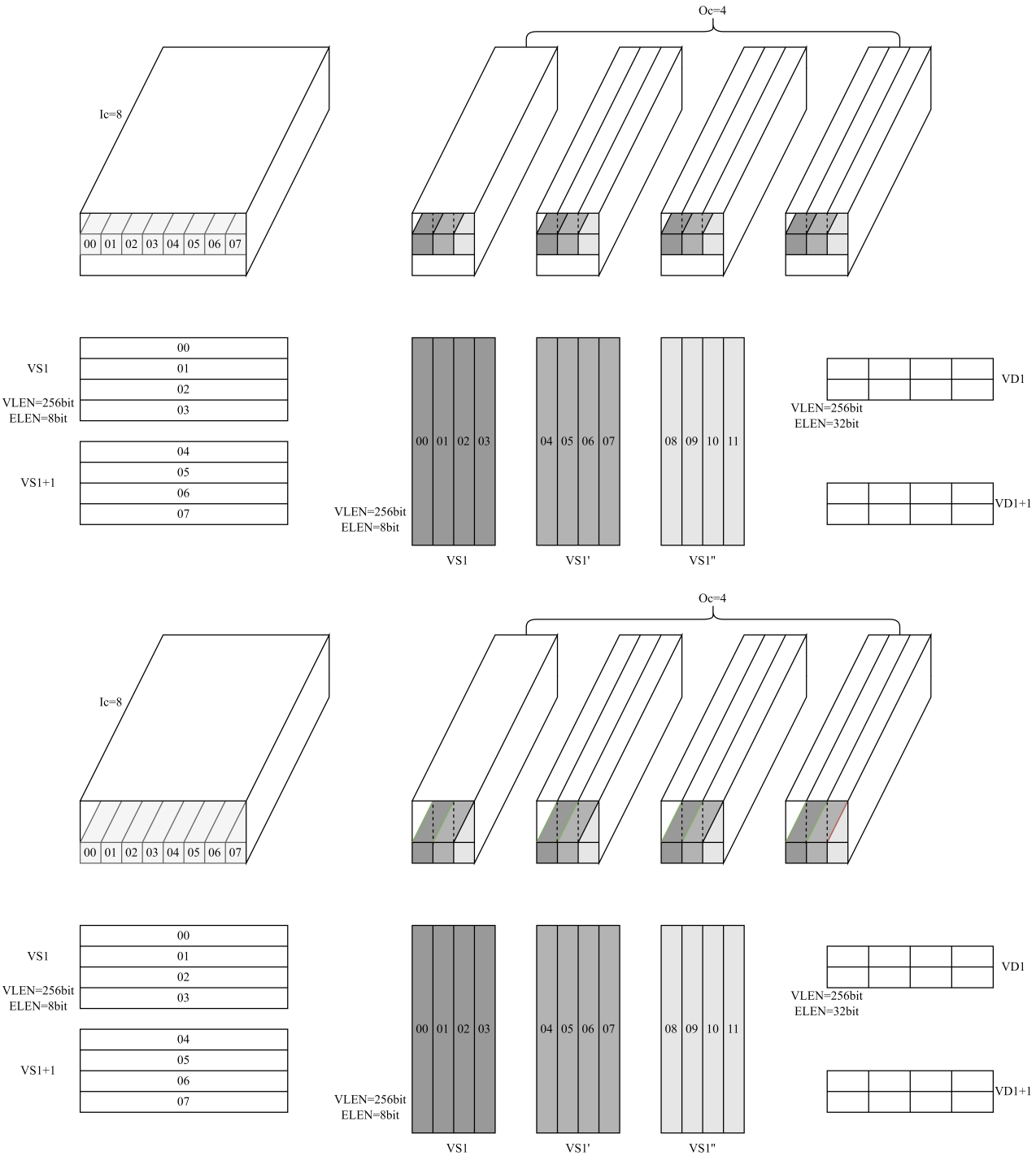
The compute shown in the following 3 figures can be performed, by employing the following three instructions. Each time, the value stored in the current VD vector register will be accumulated with the value obtained from this matrix multiplication.

```
vmadot     vd, vs1, vs2
vmadot1    vd, vs1, vs2'
vmadot2    vd, vs1, vs2''
```

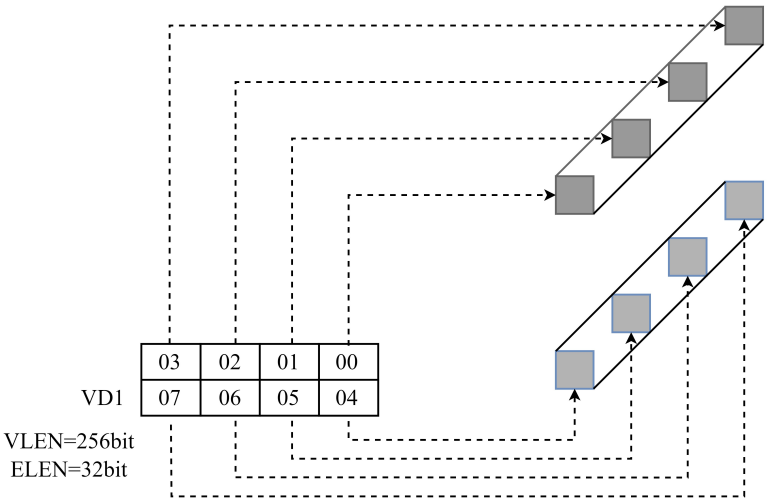


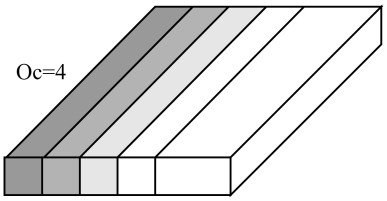


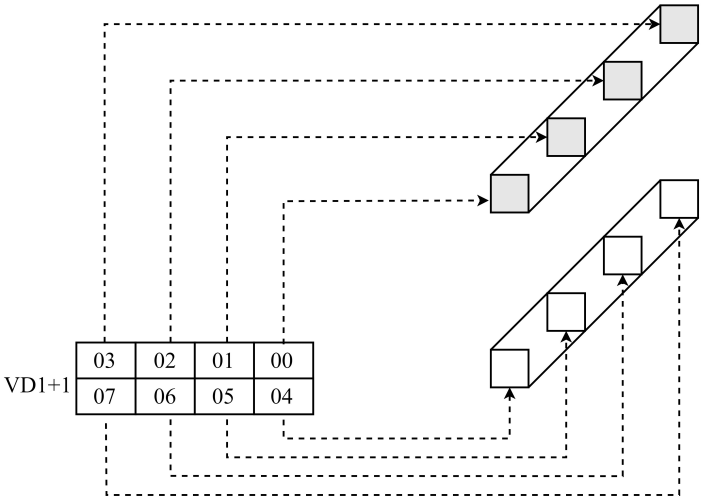
This implementation accomplishes the calculation of the first row within the receptive field of the feature map and the first row of the convolution. The calculations for the second and third rows are carried out in the same accumulative manner as described above, as shown in the following tow figures, respectively.



After completing the above operations, the values stored in the target vector register correspond to the output feature map as shown in following figure.







To slide the input feature map, padding with zeros is performed due to the row direction being less than 8. At this point, the values in the VS1 register only need to be transferred from the previous round's VS1+1 register, and the values in the VS1+1 register can be set to zero. By following the representation method shown in the following figure, the output feature map can be computed, thus completing the 2D convolution operation.