Copyright Notice

Contact Information

Should you have any problems with the information contained herein or any suggestions, please contact Nuclei System Technology by email support@nucleisys.com, or visit "Nuclei User Center" website http://user.nucleisys.com for supports or online discussion.

Revision History

Rev.	Revision Date	Revised Section	Revised Content
1.5.0	2020/1/20	N/A	1. First version as the full English
2.0.0	2020/4/30	1	2. RISC-V P extension update from v0.5 to v0.5.4

Table of Contents

List of Figures

1. Overview of Nuclei SIMD DSP Instructions

The Packed-SIMD DSP of Nuclei Processor Core basically follows the RISC-V “P” Extension Proposal (Version 0.5.4). User can easily get the original copy from “Nuclei User Center” website http://user.nucleisys.com or from other public channels.

Besides, Nuclei added some SIMD DSP instructions to improve SIMD DSP performance further. For the details of the Nuclei added SIMD DSP instructions, please refer to “Appendix A” of this document for more details.

2. Introduction of NMSIS

2.1. Background

Many microcontroller-based applications can benefit from efficient digital signal processing libraries. In order to quickly and easily handle a variety of complex DSP functions, Nuclei have established a NMSIS DSP library, which is also compatible to ARM open-source CMSIS DSP library, helping users to handle complex DSP calculations on the processor more conveniently.

For more details of the NMSIS, please refer to its online doc from http://doc.nucleisys.com/nmsis.

2.2. DSP Library Functions

In NMSIS DSP library, it includes many practical DSP functions. The library is divided into a number of functions each covering a specific category:

- Basic Math Function: Support basic math function, e.g. add, sub, mul, div, etc.
- Fast Math Function: Mainly include sin, cos, sqrt functions, etc.
- Complex Math Function: Mainly include vector calculation and module operation.
- Filter Function: IIR, FIR, LMS, etc.
- Matrix Function: Support matrix calculation.
- Transform Function: Include cfft/ciff, rfft/rifft calculation, etc.
- Motor Control Function: Mainly include PID control functions.
- Statistics Function: Include average, RMS functions.
- Support Function: Include data-copy, transformation between integers and floating-point.
- Interpolation Function: Support interpolation calculation.

The library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. All the library functions are declared in the file riscv_math.h. The functions end with _f32 operating on 32-bit floating-point values. The functions end with _q8, _q15, q31 operating on integers.

For more details of the library functions, please refer to NMSIS online doc from http://doc.nucleisys.com/nmsis.

2.3. DSP Intrinsic Functions

When doing calculation, users can directly call the functions in the NMSIS DSP library to perform efficiently and quickly. When the required functions are not found in the library, users can also directly call the DSP intrinsic functions to meet the requirements and handle related data processing.

For more details of the intrinsic functions, please refer to NMSIS online doc from http://doc.nucleisys.com/nmsis.

3. Example of DSP Program

This section will use a simple example to introduce how to set up a project, and operate by calling the NMSIS DSP library.

The example is named as “demo_dsp”, in this project, the program want to calculate the average value of the arrays for different data types. As a demo, the program will use the native reference C program and call the DSP library function to calculate the result respectively, and show their results and performance cycles.

Please refer to “application/baremetal/demo_dsp” directory from Nuclei-SDK (https://github.com/Nuclei-Software/nuclei-sdk) for more details about the “demo_dsp” program.

The code structure of this program and the flow of this project are described in detail below.

- demo_dsp.c is the program source code file. The detail of the code is explained below:

```
……

#define BENCH_INIT()          enter_cycle=__get_rv_cycle(); \
                                  printf("CSV, BENCH START, %llu\n", enter_cycle);
#define BENCH_START(func)    start_cycle=__get_rv_cycle();
#define BENCH_END(func)       end_cycle=__get_rv_cycle(); \
                                  cycle=end_cycle-start_cycle; \
                                  printf("CSV, %s, %llu\n", #func, cycle);
#define BENCH_FINISH()        exit_cycle=__get_rv_cycle(); \
                                  cycle=exit_cycle-enter_cycle; \
                                                 printf("CSV, BENCH END, %llu\n", cycle);

// Defined a comparison function which compares “the result calculated by DSP library” with “the result of the reference native C Code”.
// Use BENCH_START and BENCH_END macro to record the clock cycles required to execute the program, and print the final result.
// In the DSP library, riscv_mean_f32, riscv_mean_q7, riscv_mean_q15, and riscv_mean_q31 are the averaging functions for 32-bit floating point, 8-bit, 16-bit, and 32-bit integers arrays respectively.
// At the same time, use the C Code program to perform the averaging operation, and also use BENCH_START and BENCH_END to record the clock cycles required to execute the program, and print the result.


// define f32_mean_compare function
void f32_mean_compare()
{
    BENCH_START(riscv_mean_f32);
    riscv_mean_f32(f32_array, ARRAY_SIZE, &f32_out);
    BENCH_END(riscv_mean_f32);

    BENCH_START(ref_mean_f32);
    ref_mean_f32(f32_array, ARRAY_SIZE, &f32_out_ref);
    BENCH_END(ref_mean_f32);
```

```
    printf("riscv vs ref: %f, %f\n", f32_out, f32_out_ref);
}

void q7_mean_compare()
{
    BENCH_START(riscv_mean_q7);
    riscv_mean_q7(q7_array, ARRAY_SIZE, &q7_out);
    BENCH_END(riscv_mean_q7);

    BENCH_START(ref_mean_q7);
    ref_mean_q7(q7_array, ARRAY_SIZE, &q7_out_ref);
    BENCH_END(ref_mean_q7);

    printf("riscv vs ref: %d, %d\n", q7_out, q7_out_ref);

}

void q15_mean_compare()
{
    BENCH_START(riscv_mean_q15);
    riscv_mean_q15(q15_array, ARRAY_SIZE, &q15_out);
    BENCH_END(riscv_mean_q15);

    BENCH_START(ref_mean_q15);
    ref_mean_q15(q15_array, ARRAY_SIZE, &q15_out_ref);
    BENCH_END(ref_mean_q15);

    printf("riscv vs ref: %d, %d\n", q15_out, q15_out_ref);
}

void q31_mean_compare()
{
    BENCH_START(riscv_mean_q31);
    riscv_mean_q31(q31_array, ARRAY_SIZE, &q31_out);
    BENCH_END(riscv_mean_q31);

    BENCH_START(ref_mean_q31);
    ref_mean_q31(q31_array, ARRAY_SIZE, &q31_out_ref);
    BENCH_END(ref_mean_q31);

      printf("riscv vs ref: %d, %d\n", q31_out, q31_out_ref);
}

//In main function, the comparison function defined in the previous code is called. It will compare the average result and speed calculated by the processor with the result and speed calculated by the C Code.

int main(int argc, char **argv)
{
   //……
   BENCH_INIT();
   f32_mean_compare();
   q7_mean_compare();
   q15_mean_compare();
   q31_mean_compare();
   BENCH_FINISH();

   return 0;

}
```

- The ref_mean.c file is an averaging operation program for different data types in C Code, which is used to compare with the results of processor. The code is explained as follows:

```
// Use C code to take the  average of input data for different data types such as 32-bit floating point, 8-bit, 16-bit, and
32-bit integers value. The results will be compared with the results of processor.

// 32-bit floating point average function
void ref_mean_f32(
  float32_t * pSrc,
  uint32_t blockSize,
  float32_t * pResult)
{
   uint32_t i;
   float32_t sum=0;
   for(i=0;i<blockSize;i++)
   {
                 sum += pSrc[i];
   }
   *pResult = sum / (float32_t)blockSize;
}

// 32-bit interger average function
void ref_mean_q31(
  q31_t * pSrc,
  uint32_t blockSize,
  q31_t * pResult)
{
   uint32_t i;
   q63_t sum=0;
   for(i=0;i<blockSize;i++)
   {
                 sum += pSrc[i];
   }
   *pResult = (q31_t) (sum / (int32_t) blockSize);
}

// 16-bit interger average function
void ref_mean_q15(
  q15_t * pSrc,
  uint32_t blockSize,
  q15_t * pResult)
{
   uint32_t i;
   q31_t sum=0;
   for(i=0;i<blockSize;i++)
   {
                 sum += pSrc[i];
   }
   *pResult = (q15_t) (sum / (int32_t) blockSize);
}

// 8-bit interger average function
void ref_mean_q7(
  q7_t * pSrc,
  uint32_t blockSize,
  q7_t * pResult)
{
   uint32_t i;
   q31_t sum=0;
   for(i=0;i<blockSize;i++)
   {
                 sum += pSrc[i];
   }
   *pResult = (q7_t) (sum / (int32_t) blockSize);
}
```

- riscv_math.h includes all the functions supported by NMSIS DSP library and provide the name of these function, the relevant codes are listed as following :

```
// 8-bit interger average function
  /**
   * @brief  Mean value of a Q7 vector.
   * @param[in]  pSrc       is input pointer
   * @param[in]  blockSize  is the number of samples to process
   * @param[out] pResult    is output value.
   */
  void riscv_mean_q7(
  const q7_t * pSrc,
        uint32_t blockSize,
        q7_t * pResult);

// 16-bit interger average function
  /**
   * @brief  Mean value of a Q15 vector.
   * @param[in]  pSrc       is input pointer
   * @param[in]  blockSize  is the number of samples to process
   * @param[out] pResult    is output value.
   */
  void riscv_mean_q15(
  const q15_t * pSrc,
        uint32_t blockSize,
        q15_t * pResult);

// 32-bit interger average function
  /**
   * @brief  Mean value of a Q31 vector.
   * @param[in]  pSrc       is input pointer
   * @param[in]  blockSize  is the number of samples to process
   * @param[out] pResult    is output value.
   */
  void riscv_mean_q31(
  const q31_t * pSrc,
        uint32_t blockSize,
        q31_t * pResult);

// 32-bit floating point average function
  /**
   * @brief  Mean value of a floating-point vector.
   * @param[in]  pSrc       is input pointer
   * @param[in]  blockSize  is the number of samples to process
   * @param[out] pResult    is output value.
   */
  void riscv_mean_f32(
  const float32_t * pSrc,
        uint32_t blockSize,
        float32_t * pResult);
```

- The above part of code is the functions used in this example. riscv_mean_f32, riscv_mean_q7, riscv_mean_q15, riscv_mean_q31 are average functions for 32-bit floating point, 8-bit, 16-bit, and 32-bit fixed-point values. When users need to use other functions, users can also find their corresponding function names in the riscv_math.h header file.

- If there are no available functions to meet the requirements, the user can directly call the DSP intrinsic functions. The intrinsic functions can be found in the *core_feature_dsp.h* header file. The example segment code of the intrinsic functions are as below:

```
// kabs8 (simd 8-bit saturating absolute)
__STATIC_FORCEINLINE unsigned long __RV_KABS8(unsigned long a)
{
    unsigned long result;
    __ASM volatile("kabs8 %0, %1" : "=r"(result) : "r"(a));
    return result;
}
```

About how to run the *demo_dsp* program, please refer to Nuclei-SDK https://github.com/Nuclei-Software/nuclei-sdk) for more details. After run the program, the printout message on the serial port is as shown in Figure 3 -1, the terminal prints the result calculated by the averaging function from NMSIS DSP library and the result of the C Code.

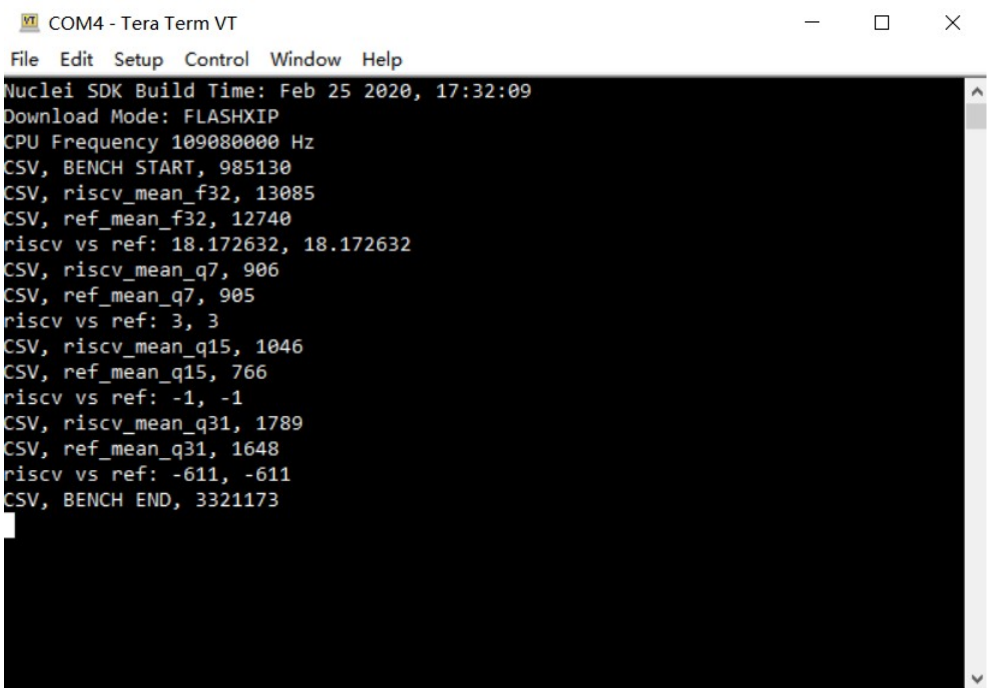

Figure 3-1 Printout message after running demo_dsp program

4. Appendix A: Nuclei SIMD DSP Additional Instruction

A.1 DKHM8 (64-bit SIMD Signed Staturating Q7 Multiply)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKHM8 1000111	Rs2	Rs1	111	Rd	1111111 GE80B

Syntax:

```
DKHM8 Rd, Rs1, Rs2
```

Purpose:
Do Q7xQ7 element multiplications simultaneously. The Q14 results are then reduced to Q7 numbers again.

Description:
For the "DKHM8" instruction, multiply the top 8-bit Q7 content of 16-bit chunks in Rs1 with the top 8-bit Q7 content of 16-bit chunks in Rs2. At the same time, multiply the bottom 8-bit Q7 content of 16-bit chunks in Rs1 with the bottom 8-bit Q7 content of 16-bit chunks in Rs2.
The Q14 results are then right-shifted 7-bits and saturated into Q7 values. The Q7 results are then written into Rd. When both the two Q7 inputs of a multiplication are 0x80, saturation will happen. The result will be saturated to 0x7F and the overflow flag OV will be set.

Operations:

```
op1t = Rs1.B[x+1]; op2t = Rs2.B[x+1]; // top
op1b = Rs1.B[x]; op2b = Rs2.B[x]; // bottom

for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x80 != aop | 0x80 != bop) {
    res = (aop s* bop) >> 7;
  } else {
    res= 0x7F;
    OV = 1;
  }
}
Rd.H[x/2] = concat(rest, resb);
x=0,2,4,6
```

Exceptions: None

Privilege level: All

Note: None

Intrinsic functions:

```
unsigned long long __dkhm8(unsigned long long a, unsigned long long b);

int8x8_t __v_dkhm8(int8x8_t a, int8x8_t b);
```

A.2 DKHM16 (64-bit SIMD Signed Saturating Q15 Multiply)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKHM16 1000011	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKHM16 Rd, Rs1, Rs2
```

Purpose:
Do Q15xQ15 element multiplications simultaneously. The Q30 results are then reduced to
Q15 numbers again.

Description:

Operations:

```
op1t = Rs1.H[x+1]; op2t = Rs2.H[x+1]; // top
op1b = Rs1.H[x]; op2b = Rs2.H[x]; // bottom

for ((aop,bop,res) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
  if (0x8000 != aop | 0x8000 != bop) {
    res = (aop s* bop) >> 15;
  } else {
  res= 0x7FFF;
  OV = 1;
  }
}
Rd.W[x/2] = concat(rest, resb);
x=0,2
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dkhm16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkhm16(int16x4_t a, int16x4_t b);
```

A.3 DKABS8 (64-bit SIMD 8-bit Saturating Absolute)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	DKABS8 10000	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKABS8 Rd, Rs1
```

Purpose:
Get the absolute value of 8-bit signed integer elements simultaneously.

Description:
This instruction calculates the absolute value of 8-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x80, this instruction generates 0x7f as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.B[x];
if (src == 0x80) {
  src = 0x7f;
  OV = 1;
} else if (src[7] == 1)
  src = -src;
}
Rd.B[x] = src;
x=7…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dkabs8(unsigned long long a);

int8x8_t __v_dkabs8(int8x8_t a);
```

A.4 DKABS16 (64-bit SIMD 16-bit Saturating Absolute)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP 1010110	DKABS16 10001	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKABS16 Rd, Rs1
```

Purpose:
Get the absolute value of 16-bit signed integer elements simultaneously.

Description:
This instruction calculates the absolute value of 16-bit signed integer elements stored in Rs1 and writes the element results to Rd. If the input number is 0x8000, this instruction generates 0x7fff as the output and sets the OV bit to 1.

Operations:

```
src = Rs1.H[x];
if (src == 0x8000) {
src = 0x7fff;
OV = 1;
} else if (src[15] == 1)
src = -src;
}
Rd.H[x] = src;
x=3…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dkabs16(unsigned long long a);

int16x4_t __v_dkabs16(int16x4_t a);
```

A.5 DKSLRA8 (64-bit SIMD 8-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKSLRA8 0101111	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSLRA8 Rd, Rs1, Rs2
```

Purpose:
Do 8-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q7 saturation for the left shift.

Description:
The 8-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically
based on the value of Rs2[3:0]. Rs2[3:0] is in the signed range of [-23, 23-1]. A positive Rs2[3:0] means logical left shift and a negative Rs2[3:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[3:0]. However, the behavior of “Rs2[3:0]==-23 (0x8)” is defined to be
equivalent to the behavior of “Rs2[3:0]==-(23-1) (0x9)”.

Operations:

```
if (Rs2[3:0] < 0) {
  sa = -Rs2[3:0];
  sa = (sa == 8)? 7 : sa;
  Rd.B[x] = SE8(Rs1.B[x][7:sa]);
} else {
  sa = Rs2[2:0];
  res[(7+sa):0] = Rs1.B[x] <<(logic) sa;
  if (res > (2^7)-1) {
    res[7:0] = 0x7f; OV = 1;
  } else if (res < -2^7) {
    res[7:0] = 0x80; OV = 1;
  }
  Rd.B[x] = res[7:0];
}
x=7…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __kslra8(unsigned long long a, unsigned long long b);

int8x8_t __v_kslra8(int8x8_t a, int b);
```

A.6 DKSLRA16 (64-bit SIMD 16-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKSLRA16 0101011	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSLRA16 Rd, Rs1, Rs2
```

Purpose:
Do 16-bit elements logical left (positive) or arithmetic right (negative) shift operation with Q15 saturation for the left shift.

Description:
The 16-bit data elements of Rs1 are left-shifted logically or right-shifted arithmetically based on the value of Rs2[4:0]. Rs2[4:0] is in the signed range of [-2^4, 2^4-1]. A positive Rs2[4:0] means logical left shift and a negative Rs2[4:0] means arithmetic right shift. The shift amount is the absolute value of Rs2[4:0]. However, the behavior of “Rs2[4:0]==-2^4 (0x10)” is defined to be equivalent to the behavior of “Rs2[4:0]==-(2^4-1) (0x11)”.

Operations:

```
if (Rs2[4:0] < 0) {
  sa = -Rs2[4:0];
  sa = (sa == 16)? 15 : sa;
  Rd.H[x] = SE16(Rs1.H[x][15:sa]);
} else {
  sa = Rs2[3:0];
  res[(15+sa):0] = Rs1.H[x] <<(logic) sa;
  if (res > (2^15)-1) {
    res[15:0] = 0x7fff; OV = 1;
  } else if (res < -2^15) {
    res[15:0] = 0x8000; OV = 1;
  }
  d.H[x] = res[15:0];
}
x=3…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dkslra16(unsigned long long a, int b);
```

```
int16x4_t __v_dkslra16(int16x4_t a, int b);
```

A.7 DKADD8(64-bit SIMD 8-bit Signed Saturating Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKADD8 0001100	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKADD8 Rd, Rs1, Rs2
```

Purpose:
Do 8-bit signed integer element saturating additions simultaneously.

Descriptions:
This instruction adds the 8-bit signed integer elements in Rs1 with the 8-bit signed integer elements in Rs2. If any of the results are beyond the Q7 number range (-27 ≤ Q7 ≤ 27-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd

Operations:

```
res[x] = Rs1.B[x] + Rs2.B[x];
if (res[x] > 127) {
res[x] = 127;
OV = 1;
} else if (res[x] < -128) {
res[x] = -128;
OV = 1;
}
Rd.B[x] = res[x];
x=7…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dkadd8(unsigned long long a, unsigned long long b);

int8x8_t __v_dkadd8(int8x8_t a, int8x8_t b);
```

A.8 DKADD16(64-bit SIMD 16-bit Signed Saturating Addition)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKADD16 0001000	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKADD16 Rd, Rs1, Rs2
```

Purpose:
Do 16-bit signed integer element saturating additions simultaneously.

Description:
This instruction adds the 16-bit signed integer elements in Rs1 with the 16-bit signed integer elements in Rs2. If any of the results are beyond the Q15 number range (-215 ≤ Q15 ≤ 215-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] + Rs2.H[x];
if (res[x] > 32767) {
res[x] = 32767;
OV = 1;
} else if (res[x] < -32768) {
res[x] = -32768;
OV = 1;
}
Rd.H[x] = res[x];
x=3…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dkadd16(unsigned long long a, unsigned long long b);

int16x4_t __v_dkadd16(int16x4_t a, int16x4_t b);
```

A.9 DKSUB8(64-bit SIMD 8-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKSUB8 0001101	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSUB8 Rd, Rs1, Rs2
```

Purpose:
Do 8-bit signed elements saturating subtractions simultaneously.

Description:
This instruction subtracts the 8-bit signed integer elements in Rs2 from the 8-bit signed integer elements in Rs1. If any of the results are beyond the Q7 number range (-27 ≤ Q7 ≤ 27-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.B[x] - Rs2.B[x];
if (res[x] > (2^7)-1) {
res[x] = (2^7)-1;
OV = 1;
} else if (res[x] < -2^7) {
res[x] = -2^7;
OV = 1;
}
Rd.B[x] = res[x];
x=7…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dksub8(unsigned long long a, unsigned long long b);

int8x8_t __v_dksub8(int8x8_t a, int8x8_t b);
```

A.10 DKSUB16(64-bit SIMD 16-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
DKSUB16 0001001	Rs2	Rs1	111	Rd	GE80B 1111111

Syntax:

```
DKSUB16 Rd, Rs1, Rs2
```

Purpose:
Do 16-bit signed integer elements saturating subtractions simultaneously.

Description:
This instruction subtracts the 16-bit signed integer elements in Rs2 from the 16-bit signed integer elements in Rs1. If any of the results are beyond the Q15 number range (-215 ≤ Q15 ≤ 215-1), they are saturated to the range and the OV bit is set to 1. The saturated results are written to Rd.

Operations:

```
res[x] = Rs1.H[x] - Rs2.H[x];
if (res[x] > (2^15)-1) {
res[x] = (2^15)-1;
OV = 1;
} else if (res[x] < -2^15) {
res[x] = -2^15;
OV = 1;
}
Rd.H[x] = res[x];
x=3…0
```

Exceptions: None
Privilege level: All
Note: None
Intrinsic functions:

```
unsigned long long __dksub16(unsigned long long a, unsigned long long b);
```

```
int16x4_t __v_dksub16(int16x4_t a, int16x4_t b);
```

A.11 EXPD80, EXPD81, EXPD82, EXPD83

EXPD80 Expand and Copy Byte 0 to 32 bit

EXPD81 Expand and Copy Byte 1 to 32 bit

EXPD82 Expand and Copy Byte 2 to 32 bit

EXPD83 Expand and Copy Byte 3 to 32 bit

Type: DSP

Format:

31 25	24 20	19 15	14 12	11 7	6 0
EXPD 0010010	xxxxx	Rs1	111	Rd	GE80B 1111111

Instr	Xxxxx
EXPD80	00000
EXPD81	00001
EXPD82	00010
EXPD83	00011

Syntax:

```
EXPD80 Rd, Rs1
EXPD81 Rd, Rs1
EXPD82 Rd, Rs1
EXPD83 Rd, Rs1
```

Purpose:
Copy 8-bit data from 32-bit chunks into 4 bytes in a register.

Description:
(EXPD80) Moves Rs1.B[0][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]
(EXPD81) Moves Rs1.B[1][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]
(EXPD82) Moves Rs1.B[2][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]
(EXPD83) Moves Rs1.B[3][7:0] to Rd.[0][7:0], Rd.[1][7:0], Rd.[2][7:0], Rd.[3][7:0]

Operations:

```
Rd.W[x][31:0] = CONCAT(Rs1.B[0][7:0], Rs1.B[0][7:0], Rs1.B[0][7:0], Rs1.B[0][7:0]);//EXPD80
Rd.W[x][31:0] = CONCAT(Rs1.B[1][7:0], Rs1.B[1][7:0], Rs1.B[1][7:0], Rs1.B[1][7:0]);//EXPD81
Rd.W[x][31:0] = CONCAT(Rs1.B[2][7:0], Rs1.B[2][7:0], Rs1.B[2][7:0], Rs1.B[2][7:0]);//EXPD82
Rd.W[x][31:0] = CONCAT(Rs1.B[3][7:0], Rs1.B[3][7:0], Rs1.B[3][7:0], Rs1.B[3][7:0]);//EXPD83
X=0
```

Exceptions: None
Privilege level: All
Note:
Intrinsic functions:

EXPD80:

```
unsigned long __expd80(unsigned long a, unsigned long b);

uint8x4_t __v_expd80(uint8x4_t a, uint8x4_t b);
```

EXPD81:

```
unsigned long __expd81(unsigned long a, unsigned long b);

uint8x4_t __v_expd81(uint8x4_t a, uint8x4_t b);
```

EXPD82:

```
unsigned long __expd82(unsigned long a, unsigned long b);

uint8x4_t __v_expd82(uint8x4_t a, uint8x4_t b);
```

EXPD83:

```
unsigned long __expd83(unsigned long a, unsigned long b);

uint8x4_t __v_expd83(uint8x4_t a, uint8x4_t b);
```