Lab 2: SIMD

Vector programming and processing 1

In this first programming and processing lab, we will write and run several simple examples that are well suited for vector (SIMD) processing.

Each studied example is composed of **two parts**:

- A **main** program with the function under test, written in C, and
- The same function **reimplemented** in assembly code using the **RISC-V vecto**r extensions.

In this lab we focus on simple arithmetic functions such as **vector addition**, **matrix multiplication**, and the **scalar (dot) product**. These exercises introduce the basic principles and functionality of vector programming and processing. We will use different data types, including bytes (`int8_t`), integers (`int32_t`), and floating-point types.

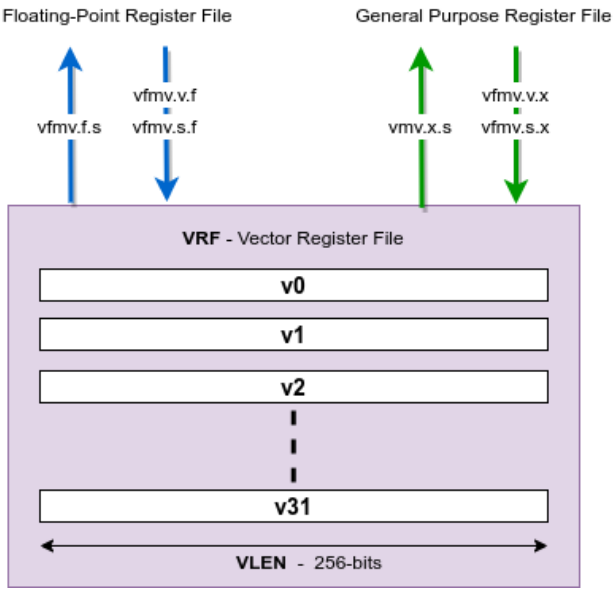
After **assembling and linking the programs**, we obtain **executable code** that can be tested directly on our development board.

This enables us to **compare the execution times of scalar and vector implementations** and to evaluate the resulting **speedup**, which depends on both the data type and the vector/matrix size.

For each example, we also explain the role and meaning of the vector instructions used in the program. In this context, it is assumed that the reader is already familiar with the basic constructs of the C programming language.

2.1 Adding two vectors

The simplest starting example is the **addition of two vectors**. Since the **X60** processors on the K1 SoC integrate a vector processing unit with a register block consisting of **32 registers, each 256 bits** wide, we can easily determine the level of parallelism.



- **32 elements** for **8-bit** data (**bytes**),
- **16 elements** for **16-bit** data (**half-words**),
- **8 elements** for **32-bit** data (**words**: **integers** or single-precision **floats**),
- **4 elements** for **64-bit** data (**double** words: **long integers** or **double**-precision floats).

Fig 2.1 Vector register block of RISC-V with 256-bit vector extension - **RV64GCCV**

Our first example starts with **bytes**.

2.1.1 `vector_byte_add.c` and `vector_byte_add_rvv.s`

The first part of the example is written in plain C code (including external **`rvv`** function):

```
  1 // vector_byte_add.c
  2 #include <stdio.h>
  3 #include <stdint.h>
  4 #include <stddef.h>
  5 #include <time.h>
  6 // external assembly function
  7 extern void vector_byte_add_rvv(const int8_t *a, const int8_t *b, int8_t *c, size_t n);
  8
  9 int main(void) {
 10     clock_t startv, endv;
 11     clock_t start, end;
 12     double elapsed_secv;
 13     double elapsed_sec;
 14     const int N = 32;
 15     int8_t a[N], b[N], c[N];
 16     int steps=1000000;
 17     // Initialize test vectors
 18       for (int i = 0; i < N; i++) { a[i] = i; b[i] = i * 2; }
 19     startv=clock();
 20     // Call vector addition
 21     for(int i=0;i<steps;i++) vector_byte_add_rvv(a, b, c, N);
 22     endv=clock();
 23     elapsed_secv = (double)(endv - startv) / CLOCKS_PER_SEC;
 24     start=clock();
 25     for(int i=0;i<steps;i++)
 26       { for( int j=0;j<32;j++) c[j] = a[j] + b[j];}
 27     end=clock();
 28     elapsed_sec = (double)(end - start) / CLOCKS_PER_SEC;
 29     // Print results
 30     printf("Addition result:\n");
 31     for (int i = 0; i < N; i++) {
 32         printf("%d + %d = %d\n", a[i], b[i], c[i]);
 33     }
 34     printf("Vector execution time: %.6f sec\n",elapsed_secv);
 35     printf("Scalar execution time: %.6f sec\n",elapsed_sec);
 36     printf("Speed-up: %.3f \n",elapsed_sec/elapsed_secv);
 37     return 0;
 38 }
```

```
    .text
    .globl vector_byte_add_rvv
vector_byte_add_rvv:
    beqz    a3, .Ldone            # if n == 0, return

.Lloop:
    # Choose VL for remaining elements, e8 lanes, LMUL=1.
    # TA/MA: tail-agnostic, mask-agnostic (simple and fast).
    vsetvli t0, a3, e8, m1, ta, ma   # t0 := VL (number of lanes we'll process)
    # Load a chunk from a and b
    vle8.v v0, (a0)               # v0 <- a[i .. i+VL-1]
    vle8.v v1, (a1)               # v1 <- b[i .. i+VL-1]
    # Vector add
    vadd.vv v2, v0, v1             # v2 <- v0 + v1
    # Store result to c
    vse8.v v2, (a2)
    # Advance pointers by VL * sizeof(int32_t) = VL * 4
    addi    t1, t0, 1              # t1 = VL * 4 (bytes)
    add     a0, a0, t1
    add     a1, a1, t1
    add     a2, a2, t1
    # Decrease remaining element count by VL and loop
    sub     a3, a3, t0
    bnez    a3, .Lloop

.Ldone:
    ret
```

```
%gcc -march=rv64gcv vector_byte_add_rvv.s vector_byte_add.c -o vector_byte_add
% ./vector_byte_add
Vector addition result:
30 + 60 = 90
31 + 62 = 93
Vector execution time: 0.013217 sec
Scalar execution time: 0.017914 sec
Speed-up: 1.355 sec..
30 + 60 = 90
31 + 62 = 93
Vector execution time: 0.020161 sec
Scalar execution time: 0.696353 sec
Speedup: 34.540
```

Now let us recompile the same example with **-O1** and **-O2 options**:

```
--------------------------------------------------------------------------------
% gcc -O2 -march=rv64gcv -fopt-info-vec vector_byte_add_rvv.s vector_byte_add.c -o vector_byte_add
vector_byte_add.c:26:21: optimized: loop vectorized using 16 byte vectors
vector_byte_add.c:18:23: optimized: loop vectorized using 4 byte vectors
% ./vector_byte_add
..
30 + 60 = 90
31 + 62 = 93
Vector execution time: 0.013217 sec
Scalar execution time: 0.017914 sec
Speed-up: 1.355 sec
--------------------------------------------------------------------------------
```

Code explanation

1. Function Entry and Zero-Check

```
    beqz    a3, .Ldone              # if n == 0, return
```

Purpose: Checks if the number of elements n (in register `a3`) is zero.

- **Action:** If `n==0`, it branches to the **`.Ldone` label** and returns immediately. This handles the edge case efficiently.

2. Vector Configuration (The Heart of RVV)

```
.Lloop:
    vsetvli t0,a3,e8,m1,ta,ma  # t0:=VL (number of lanes to process)
```

`vsetvli (Set Vector Length)`: This is the **most critical instruction**. It configures the vector unit and determines **how many elements** will be processed in this loop iteration.

  - **`t0`:** The destination register that will hold the actual **Vector Length (VL)** for this iteration.
  - **`a3`:** The source register, which is the **number of elements remaining to be processed**.
  - **`e8`:** Element width is **8 bits** (a byte). This means **each "lane"** in the vector register **holds one byte**.
  - **`m1`: LMUL=1** (Vector Register Grouping multiplier of **1**). This means we are using **single vector registers** (**e.g., `v0,v1`**), not groups of them.
  - **`ta, ma`: Tail Agnostic** and **Mask Agnostic**. This tells the hardware it doesn't need to preserve the values in the **leftover lanes** (**tail**) or **mask bits** beyond the active **VL**, which is **the fastest option** for simple loops like this.

- **Result:** The hardware calculates the **maximum number of elements** (**`VL`**) it can **process in one go** based on the available **vector register length** (**VLEN**) and the requested configuration (**`e8, m1`**), but without exceeding the remaining count **`a3`**. This value is placed in **`t0`**.

3. Loading Data into Vector Registers

```
    vle8.v v0,(a0)              # v0 <- a[i .. i+VL-1]
    vle8.v v1,(a1)              # v1 <- b[i .. i+VL-1]
```

`vle8.v` (Vector Load Element (8-bit)): These instructions **load contiguous data** from memory into vector registers.

  - **`v0,(a0)`: Loads `VL`** (from **`t0`**) **bytes** from the memory address in **`a0`** into vector register **`v0`**.
  - **`v1,(a1)`: Loads `VL` bytes** from the memory address in **`a1`** into vector register **`v1`**.

- The processor only reads/writes the number of bytes specified by **`VL`**, making it safe even if the pointers are near the end of a page.

4. The Actual Computation (Vector Addition):

```
    vadd.vv v2, v0, v1              # v2 <- v0 + v1
```

vadd.vv (**Vector Add Vector-Vector**): This performs the **element-wise addition**.

- **v2:** Destination vector register. Each lane in **v2** will hold the sum of the corresponding lanes in **v0** and **v1**.
- **v0, v1:** Source operand vector registers.
- The operation is performed on **all VL active lanes in parallel**.

5. Storing the Result

```
    vse8.v v2, (a2)
```

vse8.v (**Vector Store Element** (8-bit)): This instruction **stores data** from a **vector register** back to memory.

- v2,(a2)**:** Stores **VL** bytes from vector register **v2** to the memory address in **a2** (the **destination pointer c**).

6. **Pointer Arithmetic**

```
    addi    t1, t0, 0               # t1 = VL * 1 (bytes)
    add     a0, a0, t1
    add     a1, a1, t1
    add     a2, a2, t1
```

Purpose: Advances all three pointers (**a0,a1,a2**) by the number of bytes we just processed to prepare for the **next chunk**.

- **addi t1, t0, 0:** This instruction is **incorrect**. It simply copies **t0 (VL) to t1**. Since we are processing bytes (e8), the pointer advance in bytes is exactly equal to the number of elements processed (VL). The correct way to write this would be **mv t1, t0**. The author likely meant **addi t1, t0, 0** as a way to move the value.
- **add a0, a0, t1:** Advances the **source pointer a** by **VL** bytes.
- **add a1, a1, t1:** Advances the **source pointer b** by **VL** bytes.
- **add a2, a2, t1:** Advances the **destination pointer c** by **VL** bytes.

7. **Loop Control**

```
    sub     a3, a3, t0          # Decrease remaining element count by VL
    bnez    a3, .Lloop          # if remaining != 0, loop again

.Ldone:
    ret
```

- **sub a3, a3, t0:** Subtracts the number of elements we just processed (**VL**, in **t0**) from the total remaining count (**a3**).
- **bnez a3, .Lloop:** Checks if the remaining count is now zero. If not, it **branches back** to **.Lloop** to process the **next chunk**.
- **ret:** When all elements are processed (**a3==0**), the function returns to the caller.

Summary and Key RVV Concepts Illustrated:

1. `vsetvli` **is the Master Control:** It **dynamically determines the chunk size** (**`VL`**) for each loop iteration based on the hardware's capabilities and the amount of data left. This makes the code portable across different RISC-V implementations with different vector register lengths (**VLEN**).
2. **Element Agnostic:** The code is written for bytes (**`e8`**), but changing just the **`vsetvli`** instruction to **`e16, e32`, or `e64`** would make it work for **shorts, ints, or long**s, respectively. The **load**/**store** (**`vle8.v/vse8.v`**) and sometimes the arithmetic instructions would also need their width suffix updated.
3. **True Single-Loop Structure:** The loop overhead is minimal. One **`vsetvli`** configures the hardware to process a **variable-sized chunk** of data with the subsequent vector instructions.
4. **Automatic Safety:** The **`VL`** value ensures the load, compute, and store operations only act on valid data, even for the last partial chunk. There is no need for complex array bounds checking inside the loop.

2.1.2 `vector_int_add.c` and `vector_int_add_rvv.s`

The following example is almost the same as the previous one; the main difference is the size and type of data elements – **`int32_t`**

```
  1 // vector__int_add.c
  2 #include <stdio.h>
  3 #include <stdint.h>
  4 #include <stddef.h>
  5 #include <time.h>
  6
  7 // Our assembly function (from previous code)
  8 extern void vector_int_add_rvv(const int32_t *a, const int32_t *b, int32_t *c, size_t n);
  9
 10 int main(void) {
 11     clock_t startv, endv;
 12     clock_t start, end;
 13     double elapsed_secv;
 14     double elapsed_sec;
 15     const int N = 32;
 16     int32_t a[N], b[N], c[N];    // 32-bit integer data
 17     int steps=1000000;
 18     // Initialize test vectors
 19     for (int i = 0; i < N; i++) { a[i] = i; b[i] = i * 2; }
 20     startv=clock();
 21     // Call vector addition
 22     for(int i=0;i<steps;i++) vector_int_add_rvv(a, b, c, N);
 23     endv=clock();
 24     elapsed_secv = (double)(endv - startv) / CLOCKS_PER_SEC;
 25     // Print results
 26     printf("Vector addition result:\n");
 27     for (int i = 0; i < N; i++) {
 28         printf("%d + %d = %d\n", a[i], b[i], c[i]);
 29     }
 30     start=clock();
 31     // Call addition
 32       for(int i=0;i<steps;i++){for(int j=0;j<32;j++)c[j]=a[j]+ b[j];}
 33     end=clock();
 34     elapsed_sec = (double)(end - start) / CLOCKS_PER_SEC;
 35     // Print results
 36     printf("Addition result:\n");
 37     for (int i = 0; i < N; i++) {
 38         printf("%d + %d = %d\n", a[i], b[i], c[i]);
 39     }
 40     printf("Vector execution time: %.6f sec\n",elapsed_secv);
 41     printf("Scalar execution time: %.6f sec\n",elapsed_sec);
 42     printf("Speed-up: %.3f sec\n",elapsed_sec/elapsed_secv);
 43     return 0;
 44 }
```

```
    .text
    .globl vector_int_add_rvv
vector_int_add_rvv:
    beqz    a3, .Ldone            # if n == 0, return

.Lloop:
    # Choose VL for remaining elements, e32 lanes, LMUL=1.
    # TA/MA: tail-agnostic, mask-agnostic (simple and fast).
    vsetvli t0, a3, e32, m1, ta, ma   # t0 := VL (number of 32-bit lanes we'll process)
    # Load a chunk from a and b
    vle32.v v0, (a0)               # v0 <- a[i .. i+VL-1] (32-bit integers)
    vle32.v v1, (a1)               # v1 <- b[i .. i+VL-1] (32-bit integers)
    # Vector add
```

```
    vadd.vv v2, v0, v1             # v2 <- v0 + v1 (32-bit addition)
    # Store result to c
    vse32.v v2, (a2)
    # Advance pointers by VL * sizeof(int32_t) = VL * 4 (bytes)
    slli    t1, t0, 2              # t1 = VL * 4 (bytes) - CORRECTED
    add     a0, a0, t1
    add     a1, a1, t1
    add     a2, a2, t1
    # Decrease remaining element count by VL and loop
    sub     a3, a3, t0
    bnez    a3, .Lloop

.Ldone:
    ret
```

```
% gcc -march=rv64gcv vector_int_add_rvv.s vector_int_add.c -o vector_int_add
% ./vector_int_add
Vector addition result:
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
..
30 + 60 = 90
31 + 62 = 93
Vector execution time: 0.041034 sec
Scalar execution time: 0.697534 sec
Speed-up: 16.999
```

Now let us recompile the same example with -O2 option:

```
%gcc -O2 -march=rv64gcv -fopt-info-vec vector_int_add_rvv.s vector_int_add.c -o vector_int_add
vector_int_add.c:32:46: optimized: loop vectorized using 16 byte vectors
% ./vector_int_add
..
30 + 60 = 90
31 + 62 = 93
Vector execution time: 0.042629 sec
Scalar execution time: 0.082662 sec
Speed-up: 1.939 sec
```

To do:

Rewrite the same example (with main test) of vector addition using **floating point** data. The following is the assembly part:

```
    .text
    .globl vector_float_add_rvv
vector_float_add_rvv:
    beqz    a3, .Ldone

.Lloop:
    # Explicit float configuration (some assemblers support this)
    vsetvli t0, a3, e32, m1, ta, ma   # Process single-precision floats
    vle32.v v0, (a0)
    vle32.v v1, (a1)
    vfadd.vv v2, v0, v1           # Single-precision floating-point add
    vse32.v v2, (a2)
    # Pointer arithmetic (same as integer version)
    slli    t1, t0, 2
    add     a0, a0, t1
    add     a1, a1, t1
    add     a2, a2, t1
    sub     a3, a3, t0
    bnez    a3, .Lloop

.Ldone:
    ret
```

Key RISC-V Vector Floating-Point Instructions:

- **`vfadd.vv`:** Vector floating-point add (vector-vector)
- **`vfadd.vf`**: Vector floating-point add (vector-scalar)
- **`vfsub.vv`**: Vector floating-point subtract
- **`vfmul.vv`**: Vector floating-point multiply
- **`vfdiv.vv`**: Vector floating-point divide
- **`vfmin.vv, vfmax.vv`**: Vector floating-point min/max

Notes:

1. The element size remains `e32` since single-precision floats are 32 bits
2. The pointer arithmetic is identical to the integer version since both **`int32_t`** and **`float`** are **4 bytes**
3. The load/store instructions (**`vle32.v/vse32.v`**) are the same since they just **move raw bytes**; the **interpretation as floats** happens in the **`vfadd`** instruction
4. The same code structure works for **`double`** (64-bit floating-point) by changing **`e32`** to **`e64`** and the pointer arithmetic to **`slli t1, t0, 3`** (multiply by **8 bytes**)

2.2 Vector average value : reduction

The following example with the assembly code for vector operations and the main C code for scalar operations. In assembly code notice the use of **reduction** instruction:

```
 vredsum.vs v2, v2, v4                # v2[0] = sum of widened lanes
```

```
# avg_i32_rvv.s
# int32_t avg_i32_rvv(const int32_t *src, size_t n);
# RV64 + V (RVV 1.0). Returns 0 if n == 0.
    .text
    .globl  avg_i32_rvv
    .type   avg_i32_rvv, @function
avg_i32_rvv:
    beqz    a1, .ret_zero           # if (n==0) return 0

    mv      a2, a1                  # save original n for the final divide
    li      t5, 0                   # 64-bit scalar accumulator sum = 0

1:  beqz    a1, 2f                  # while (remaining > 0)
    # Set VL for remaining elements, SEW=32, LMUL=1
    vsetvli t0, a1, e32, m1, ta, ma     # t0 := vl
    # Load a chunk of int32
    vle32.v v0, (a0)
    # Widen to e64 (LMUL=m2) and reduce-sum to a 64-bit scalar
    vsetvli x0, t0, e64, m2, ta, ma     # same element count in widened type
    vsext.vf2 v2, v0                    # v2 (e64,m2) = sign-extend(v0)
    vmv.v.i  v4, 0                      # zero seed
    vredsum.vs v2, v2, v4               # v2[0] = sum of widened lanes
    vmv.x.s  t2, v2                     # t2 = chunk sum (64-bit)
    add      t5, t5, t2                 # accumulate
    # Advance pointers/counters
    slli     t1, t0, 2                  # bytes = vl * 4
    add      a0, a0, t1                 # src += vl
    sub      a1, a1, t0                 # remaining -= vl
    bnez     a1, 1b

2:  # Average = sum / original_n  (truncates toward zero)
    mv       a0, t5                     # a0 = 64-bit sum
    # a2 holds original n (>0)
    div      a0, a0, a2                 # signed divide, trunc toward zero
    sext.w   a0, a0                     # return as int32_t (RV64 sign-extend)
    ret

.ret_zero:
    li       a0, 0
    ret
    .size avg_i32_rvv, .-avg_i32_rvv
```

```
  4 #include <stdlib.h>
  5 #include <time.h>
  6
  7 // Assembly function prototype
  8 extern int32_t avg_i32_rvv(const int32_t *src, size_t N);
  9
 10 int steps=10000;
 11
 12 int32_t  avg(const int32_t *src, int MAX)
 13 {
 14     int64_t sum = 0;
 15     for (int i = 0; i < MAX; i++) sum += src[i];
 16     return (int32_t) sum/MAX;
 17 }
 18
 19 int main(void) {
 20     enum { N = 1024 };
 21     int32_t data[N];
 22     clock_t startv, endv;
 23     clock_t start, end;
 24     double elapsed_secv;
 25     double elapsed_sec;
 26     int32_t avg_vec, avg_c;
 27
 28     // Fill test data with a simple pattern
 29     for (int i = 0; i < N; i++) {
 30         data[i] = i;  // (i % 50) - 25; // values from -25 to 24
 31     }
 32
 33     // Call the RVV assembly function
 34     startv=clock();
 35     for(int i=0;i<steps;i++) avg_vec = avg_i32_rvv(data, 1024);
 36     endv=clock();
 37     elapsed_secv = (double)(endv - startv) / CLOCKS_PER_SEC;
 38     printf("Vector execution time in sec: %.6f\n",elapsed_secv);
 39
 40     // Call the scalar function
 41     start=clock();
```

```
42      for(int i=0;i<steps;i++) avg_c=avg(data,N);
43      end=clock();
44      elapsed_sec = (double)(end - start) / CLOCKS_PER_SEC;
45      printf("Scalar execution time in sec: %.6f\n",elapsed_sec);
46      printf("Speed-up: %.3f\n",elapsed_sec/elapsed_secv);
47
48      // Display results
49      printf("Vector execution result: %d\n", avg_vec);
50      printf("Scalar execution result: %d\n", avg_c);
51
52      // Optional: check if they match
53      if (avg_vec == avg_c) {
54          printf("Test PASSED\n");
55          return 0;
56      } else {
57          printf("Test FAILED\n");
58          return 1;
59      }
60 }
61
```

```
% gcc -march=rv64gcv avg_i32_rvv.s average_int32.c -o average_int32
% ./average_int32
Vector execution time in sec: 0.026015
Scalar execution time in sec: 0.141505
Speed-up: 5.439
Vector execution result: 511
Scalar execution result: 511
Test PASSED
```

Compilation with optimization:

```
gcc -O3 -march=rv64gcv -fopt-info-vec avg_i32_rvv.s average_int32.c -o average_int32
average_int32.c:15:23: optimized: loop vectorized using variable length vectors
average_int32.c:29:23: optimized: loop vectorized using variable length vectors
orangepi@orangepirv2:~/RV.Parallel/codes/book_lab2$ ./average_int32
Vector execution time in sec: 0.025848
Scalar execution time in sec: 0.011378
Speed-up: 0.440
Vector execution result: 511
Scalar execution result: 511
Test PASSED
```

To do:

Write the same example for **byte data elements**