

# CSE302 - Building Customized Computer

## Final project: SIMD instruction

20202026 - Nguyen Minh Duc  
20202027 - Thu Phuong Nguyen

### 1 Introduction

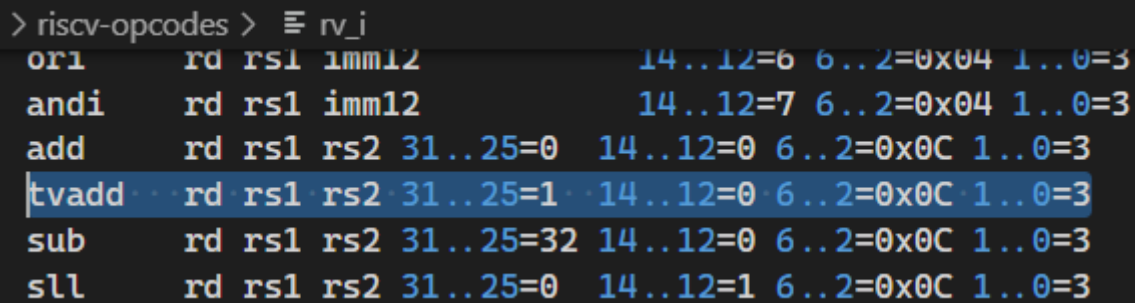
SIMD, short for Single Instruction Multiple Data, is a computing method that enables the processing of multiple data within a single instruction with the help of parallelism. Typically SIMD instructions perform operations on bit vectors containing multiple values. The idea of SIMD can be applied to a lot of operations such as addition, multiplication, load, and store, which, in turn, can help write complex operations such as matrix multiplication more efficiently. However, SIMD has some limitations, one of them is not being able to handle different kinds of operations. In this project, we are going to create a new toy SIMD instruction, called `tvadd` to perform 8 additions in parallel. For this project, we will work on 8-bit integers.

### 2 Experiment

In this section, we will briefly summarize our experiment, which contains adding a new instruction, the implementation, and reporting the execution.

#### 2.1 Adding a new instruction

There are two ways of adding the SIMD idea into `spike`. The first is to add a true new `tvadd` instruction to the encoding of `spike`. We tried to add the instruction according to the tutorial in the slides



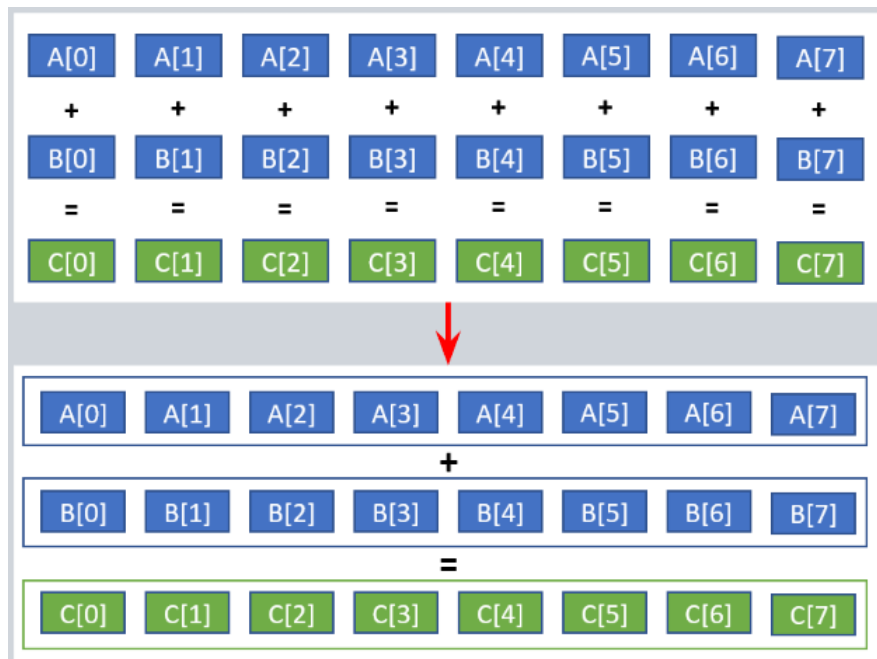
```
> riscv-opcodes > ≡ rv_i
ori      rd rs1 imm12      14..12=6 6..2=0x04 1..0=3
andi     rd rs1 imm12      14..12=7 6..2=0x04 1..0=3
add       rd rs1 rs2 31..25=0 14..12=0 6..2=0x0C 1..0=3
tvadd     rd rs1 rs2 31..25=1 14..12=0 6..2=0x0C 1..0=3
sub       rd rs1 rs2 31..25=32 14..12=0 6..2=0x0C 1..0=3
sll       rd rs1 rs2 31..25=0 14..12=1 6..2=0x0C 1..0=3
```

After this, we tried to build the `op-codes` but the simulator failed to execute the program. Although this approach is desirable, it requires either calculating the bit encoding of the instruction accurately or modifying the compiler and assembler. So, we decide to follow the second approach: modifying the existing `add` instruction.

#### 2.2 Implementation

In this project, we use 64-bit registers to load, store, and conduct computations on eight 8-bit integers. The idea is to treat the 64-bit number as the concatenation of the 8 smaller integers and

do element-wise operations on them in a parallel way.



A similar idea could be done using just C++ alone using pointers type casting and normal addition, here is an example

</> source code

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main() {
5      int8_t a[8];
6      for(int i = 0; i < 8; i++) a[i] = i + 1;
7
8      int8_t b[8];
9      for(int i = 0; i < 8; i++) b[i] = i;
10
11     int8_t c[8];
12
13     int64_t *ptr_a = (int64_t*)a;
14     int64_t *ptr_b = (int64_t*)b;
15     int64_t res_c = *ptr_a + *ptr_b;
16     int64_t *ptr_c = (int64_t*)c;
17     *ptr_c = res_c;
18
19     for(int i = 0; i < 8; i++) {
20         cout << (int)a[i] << " + " << (int)b[i] << " = " << (int)c[i] << '\n';
21     }
22
23     return 0;
24 }
25

```

However, doing this yields incorrect results when there's integer overflow or underflow happening in one of the 8 `int8_t`'s. However, we could employ this implementation to **spike** in a similar manner. Firstly, we define a new type, called `vecbit`, which is just an alias for the `uint64_t` type. This newly created type will contain a bit vector of size 8. To load and store this kind of bit vector, we have to rely on the type casting mechanism of C.

```

4  typedef uint64_t vecbit;
5
6  vecbit __load_vec(int8_t *ptr) {
7      return *(uint64_t*)ptr;
8  }
9
10 void __store_vec(int8_t *ptr, vecbit value) {
11     *(uint64_t*)ptr = value;
12 }

```

The only thing left is to implement the `tvadd` logic in our source code.

```

14 void __tvadd(int8_t *z, int8_t *x, int8_t *y) {
15     vecbit vecx = __load_vec(x);
16     vecbit vecy = __load_vec(y);
17     vecbit vecz = 0;
18     // vecz = vecx + vecy
19     __asm__ volatile("add %0, %1, %2"
20                     : "=r"(vecz) : "r"(vecx), "r"(vecy));
21     __store_vec(z, vecz);
22 }

```

Lastly, we have to modify the `add` instruction of `spike`. The full implementation is as follows

```

1  require_rv64;
2  FILE* log_file = p->get_log_file();
3  // add rd, rs1, rs2
4  // this is equivalent to rd = rs1 + rs2
5  if(pc == 0x10218) { // This specific address is invoking SIMD
6      reg_t x = RS1; // Read rs1
7      reg_t y = RS2; // Read rs2
8      reg_t z = 0; // Initialize the sum to be 0
9      // Simulate the parallelism of SIMD
10     for(int i = 7; i >= 0; i--) {
11         // Performing element-wise addition
12         z += (uint8_t)((x >> (i << 3)) + (y >> (i << 3)));
13         z <<= 8 * (i != 0);
14     }
15     // Report execution of our newly added tvadd
16     fprintf(log_file, "OUR ADD: %ld + %ld = %ld!!!!\n", x, y, z);
17     // Write the result to the register
18     WRITE_RD(sext_xlen(z));
19 } else {
20     // Performing normal addition
21     WRITE_RD(sext_xlen(RS1 + RS2));
22 }
23

```

In this implementation, we only apply SIMD when the program counter is at the specific, which is the line when we called `add` in `__tvadd`. In addition, we use a for loop to “simulate” the parallelism. In the for loop, we execute the addition element-wise for each of the eight consecutive bits, which correspond to an `int8_t`. The `fprintf` at line 16 is to report the execution of SIMD to check if the instruction is executed as intended. The log from `spike` confirms the execution of `tvadd`.

```

385277 core 0: 0x0000000000010210 (0xfe843783) ld a5, -24(s0)
385278 core 0: 0x0000000000010214 (0xfe843703) ld a4, -32(s0)
385279 core 0: 0x0000000000010218 (0x00e787b3) add a5, a5, a4
385280 OUR ADD: 6160972530438686141 + -7996710985908134043 = -1836021029974629342!!!!
385281 core 0: 0x000000000001021c (0xfc43c23) sd a5, -40(s0)
385282 core 0: 0x0000000000010220 (0xfd843583) ld a1, -40(s0)

```

## 2.3 Implementation in Rocket chip

```

1  class ALU(implicit p: Parameters) extends CoreModule()(p) {
2      val io = IO(new Bundle {
3          // Checking if the add is using simd
4          val using_simd = Input(UInt(1.W))
5
6          val dw = Input(UInt(SZ_DW.W))
7          val fn = Input(UInt(SZ_ALU_FN.W))
8          val in2 = Input(UInt(xLen.W))
9          val in1 = Input(UInt(xLen.W))
10         val out = Output(UInt(xLen.W))
11         val adder_out = Output(UInt(xLen.W))
12         val cmp_out = Output(Bool())
13     })
14
15     // ADD, SUB
16     val in2_inv = Mux(isSub(io.fn), ~io.in2, io.in2)
17     val in1_xor_in2 = io.in1 ^ in2_inv
18     when(io.using_simd) {
19         val x = io.in1; // Read rs1
20         val y = in2_inv; // Read rs2
21         val z = 0; // Initialize the sum to be 0
22         // Simulate the parallelism of SIMD
23         for(i <- 8 to 1 by -1) {
24             // Performing element-wise addition
25             z += ((x >> (i << 3)) + (y >> (i << 3)));
26             z <<= 8 * (i != 0);
27         }
28         // Write the result to the register
29         io.adder_out := z
30     }.otherwise {
31         io.adder_out := io.in1 + in2_inv + isSub(io.fn)
32     }
33     ...
34 }
35

```

Here, we need to modify the ALU unit of Rocket chip to perform SIMD operation in a parallel way. The logic is the same as we did in `spike`, we just need to translate from C++ to Scala. Lastly, we need to assign the flag `using_simd` when the program counter is at the specified address in the execution stage. However, we have yet to figure it out, so our implementation could not be applied to the hardware.

## 3 Results

In this section, we will discuss some interesting results that we found after implementing and testing `tvadd`. We will compare `tvadd` with normal vector addition. The metric for comparison is **number of clock cycles**. Here is the implementation of the normal add:

```

1  void vector_add_norm(int8_t *c, int8_t *a, int8_t *b, int n) {
2      for(int i = 0; i < n; i++) {
3          c[i] = a[i] + b[i];
4      }
5  }
6

```

And here is with tvadd:

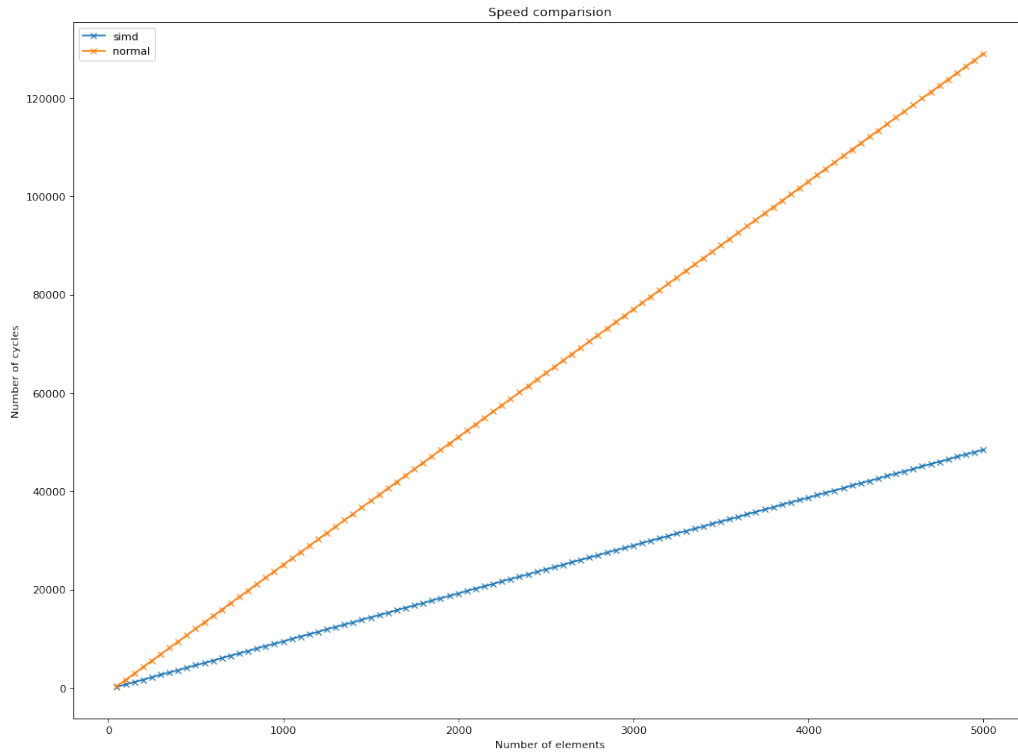
```
1 void vector_add_simd(int8_t *c, int8_t *a, int8_t *b, int n) {
2     int i = 0;
3     for(; i < n; i += 8) {
4         __tvadd(c + i, a + i, b + i);
5     }
6     for(; i < n; i++) {
7         c[i] = a[i] + b[i];
8     }
9 }
10
```

Here is the result when applying both of them on a random array of size 10:

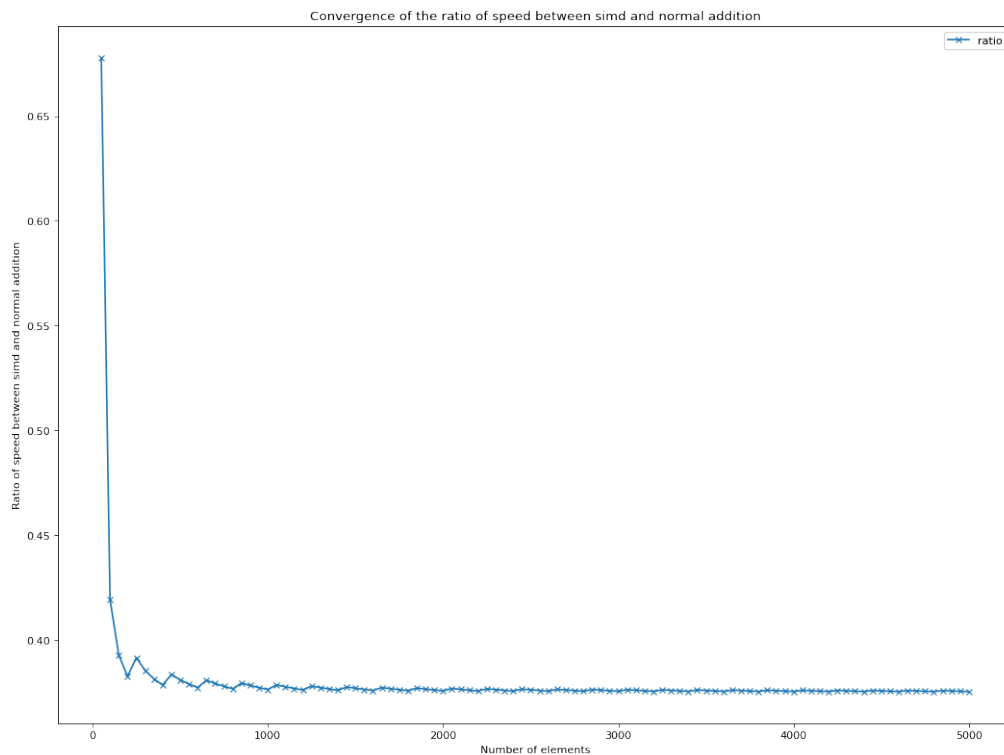
```
vagrant@ubuntu-bionic:~/lab3-v1/lab3$ make run-e1
spike -l --log=e1.log pk e1
bbl loader

-----
SIMD Test! A benchmark between SIMD and normal addition
-----
Initialize 2 random arrays of size 10!
-----
Calculating using SIMD
The number of cycles passed when using SIMD: 202
-128 + 101 = -27
-59 + 95 = 36
-68 + 123 = 55
-53 + 9 = -44
-33 + -70 = -103
43 + -7 = 36
-128 + 5 = -123
85 + -111 = -26
102 + -27 = 75
29 + 98 = 127
-----
Calculating using normal addition
The number of cycles passed when using normal addition: 298
-128 + 101 = -27
-59 + 95 = 36
-68 + 123 = 55
-53 + 9 = -44
-33 + -70 = -103
43 + -7 = 36
-128 + 5 = -123
85 + -111 = -26
102 + -27 = 75
29 + 98 = 127
-----
```

As we can see, the SIMD approach needs about 90 clock cycles less than the normal addition. When we compare them on multiple sizes of arrays, the gap in speed becomes very clear.



When we investigate the ratio of the number of clock cycles between them, the ratio starts to converge to about 0.37 after the number of array elements reaches 500.



The graph suggests that SIMD takes **0.37 times** cycles that the conventional addition method takes to finish the job. Roughly **2.7 times** increased in speed.