```
// src/q15_axpy_challenge.c
// Single-solution RVV challenge: Q15 y = a + alpha * b  (saturating to
Q15)
//

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

// -------------------- Scalar reference (no intrinsics) ----------------
----
static inline int16_t sat_q15_scalar(int32_t v) {
    if (v >  32767) return  32767;
    if (v < -32768) return -32768;
    return (int16_t)v;
}

void q15_axpy_ref(const int16_t *a, const int16_t *b,
                  int16_t *y, int n, int16_t alpha)
{
    for (int i = 0; i < n; ++i) {
        int32_t acc = (int32_t)a[i] + (int32_t)alpha * (int32_t)b[i];
        y[i] = sat_q15_scalar(acc);
    }
}

// -------------------- RVV include per ratified v1.0 spec --------------
----
#if __riscv_v_intrinsic >= 1000000
  #include <riscv_vector.h>  // v1.0 test macro & header inclusion
#endif

// -------------------- RVV implementation (mentees edit only here) -----
----
void q15_axpy_rvv(const int16_t *a, const int16_t *b,
                  int16_t *y, int n, int16_t alpha)
{
#if !defined(__riscv) || !defined(__riscv_vector) || (__riscv_v_intrinsic
< 1000000)
    // Fallback (keeps correctness off-target)
    q15_axpy_ref(a, b, y, n, alpha);
#else
    int32_t vl;  // Vector length
    for (int i = 0; i < n; i += vl) {
        vl = vsetvl_e16m2(n - i);
        vint16m2_t va = vle16_v_i16m2(&a[i], vl);
        vint16m2_t vb = vle16_v_i16m2(&b[i], vl);
        vint32m4_t vprod = vmul_vx_i32m4(vb, alpha, vl);
        vint32m4_t va_ext = vwadd_vx_i32m4(va, 0, vl);
        vint32m4_t vacc = vadd_vv_i32m4(va_ext, vprod, vl);
        vint16m2_t vy = vnclip_wx_i16m2(vacc, 15, vl);
        vse16_v_i16m2(&y[i], vy, vl);
    }

#endif
}

// -------------------- Verification & tiny benchmark -------------------
----
static int verify_equal(const int16_t *ref, const int16_t *test, int n,
int32_t *max_diff) {
    int ok = 1;
    int32_t md = 0;
    for (int i = 0; i < n; ++i) {
        int32_t d = (int32_t)ref[i] - (int32_t)test[i];
        if (d < 0) d = -d;
        if (d > md) md = d;
        if (d != 0) ok = 0;
    }
    *max_diff = md;
    return ok;
}

#if defined(__riscv)
static inline uint64_t rdcycle(void) { uint64_t c; asm volatile ("rdcycle
%0" : "=r"(c)); return c; }
#endif

int main(void) {
    int ok = 1;
    const int N = 4096;
    int16_t *a  = (int16_t*)aligned_alloc(64, N * sizeof(int16_t));
    int16_t *b  = (int16_t*)aligned_alloc(64, N * sizeof(int16_t));
    int16_t *y0 = (int16_t*)aligned_alloc(64, N * sizeof(int16_t));
    int16_t *y1 = (int16_t*)aligned_alloc(64, N * sizeof(int16_t));

    // Deterministic integer data (no libm)
    srand(1234);
    for (int i = 0; i < N; ++i) {
        a[i] = (int16_t)((rand() % 65536) - 32768);
        b[i] = (int16_t)((rand() % 65536) - 32768);
    }

    const int16_t alpha = 3; // example scalar gain

    uint32_t c0 = rdcycle();
    q15_axpy_ref(a, b, y0, N, alpha);
    uint32_t c1 = rdcycle();
    printf("Cycles ref: %u\n", c1 - c0);

    int32_t md = 0;

#if defined(__riscv)
    c0 = rdcycle();
    q15_axpy_rvv(a, b, y1, N, alpha);
    c1 = rdcycle();
    ok = verify_equal(y0, y1, N, &md);
    printf("Verify RVV: %s (max diff = %d)\n", ok ? "OK" : "FAIL", md);
    printf("Cycles RVV: %llu\n", (unsigned long long)(c1 - c0));
#endif

    free(a); free(b); free(y0); free(y1);
    return ok ? 0 : 1;
}




/*
// For the simulation, I used qemu-riscv64 to run the compiled ELF
binary.
// Here is how I compiled and ran the code:

riscv64-unknown-elf-gcc -march=rv64imcbvzifencei -mabi=lp64d
q15_axpy_challenge.c -o q15_axpy.elf

qemu-riscv64 q15_axpy.elf

AND THE OUTPUT IS:
Cycles ref: 6229328
Verify RVV: OK (max diff = 0)
Cycles RVV: 4070398

@author Kouachi Corneille EKON
@linkedin: https://www.linkedin.com/in/ekon-ihc
*/
```