X64 NASM Assembly Exercises: Algorithmic Programming

Easy Exercises (1-2 hours each)

String Operations

1. **strlen** - Implement `int strlen(const char* str)` that returns the length of a null-terminated C string.
2. **strcpy** - Implement `char* strcpy(char* dest, const char* src)` that copies a string from src to dest and returns dest.
3. **strchr** - Implement `char* strchr(const char* str, int c)` that finds the first occurrence of character c in string str.

Math - Basic

4. **fibonacci** - Implement `long fib(int n)` that computes the n-th Fibonacci number (n ≤ 40).
5. **factorial** - Implement `long factorial(int n)` that computes n! for n ≤ 20.
6. **gcd** - Implement `int gcd(int a, int b)` using Euclidean algorithm.
7. **power_mod** - Implement `long power_mod(long base, long exp, long mod)` that computes (base^exp) % mod.
8. **is_prime** - Implement `int is_prime(long n)` that returns 1 if n is prime, 0 otherwise (trial division).

Array Operations

9. **array_sum** - Implement `long array_sum(const long* arr, int n)` that sums all elements in an array.
10. **array_min_max** - Implement `void array_min_max(const int* arr, int n, int* min, int* max)` that finds both min and max in one pass.
11. **array_reverse** - Implement `void array_reverse(int* arr, int n)` that reverses an array in-place.
12. **memset** - Implement `void* memset(void* ptr, int value, size_t num)` that fills memory with a constant byte.

Bit Operations

13. **popcount** - Implement `int popcount(unsigned long x)` that counts the number of set bits (without using POPCNT instruction).
14. **reverse_bits** - Implement `unsigned int reverse_bits(unsigned int x)` that reverses the bit order in a 32-bit integer.
15. **is_power_of_two** - Implement `int is_power_of_two(unsigned long x)` efficiently.

CPU/System

16. **rdtsc_read** - Implement `unsigned long read_tsc(void)` that reads the CPU timestamp counter using RDTSC.
17. **cpu_brand** - Implement `void get_cpu_brand(char* buffer)` that retrieves CPU brand string using CPUID (buffer size >= 48 bytes).

Computational Geometry - Basic

18. **point_distance** - Implement `float point_distance(float x1, float y1, float x2, float y2)` computing Euclidean distance between two 2D points.
19. **point_in_rect** - Implement `int point_in_rect(float px, float py, float rx, float ry, float rw, float rh)` checking if point is inside rectangle.

Hash/Checksum

20. **simple_hash** - Implement `unsigned int simple_hash(const char* str)` using the djb2 hash algorithm.
21. **xor_checksum** - Implement `unsigned char xor_checksum(const unsigned char* data, int len)` that XORs all bytes.

Number Theory

22. **lcm** - Implement `long lcm(long a, long b)` that computes least common multiple.
23. **mod_inverse** - Implement `long mod_inverse(long a, long m)` using extended Euclidean algorithm (assume gcd(a,m)=1).

Data Structures - Basic

24. **linear_search** - Implement `int linear_search(const int* arr, int n, int target)` returning index or -1.
25. **binary_search** - Implement `int binary_search(const int* sorted_arr, int n, int target)` on sorted array.

Medium Exercises (3-5 hours each)

String Operations - Optimized

1. **strcmp** - Implement `int strcmp(const char* s1, const char* s2)` with two versions: scalar and AVX2-optimized. Compare performance on strings of various lengths (10, 100, 1000, 10000 bytes).
2. **memcpy_optimized** - Implement `void* memcpy(void* dest, const void* src, size_t n)` with AVX2 optimization for large blocks. Compare with scalar version.
3. **strstr** - Implement `char* strstr(const char* haystack, const char* needle)` using naive algorithm and compare with Boyer-Moore-Horspool variant.

Math - Intermediate

4. **matrix_multiply** - Implement `void matmul(const float* A, const float* B, float* C, int n)` for n×n matrices using AVX/AVX2 instructions. Test with n = 8, 16, 32, 64.
5. **fibonacci_matrix** - Implement `long fib_fast(int n)` using matrix exponentiation method for computing large Fibonacci numbers ($n \leq 90$).
6. **sieve_eratosthenes** - Implement `int sieve(int n, int* primes)` that finds all primes up to n using Sieve of Eratosthenes. Return count of primes.
7. **polynomial_eval** - Implement `double poly_eval(const double* coeffs, int degree, double x)` using Horner's method with AVX2 for SIMD evaluation of 4 polynomials simultaneously.

Computational Geometry

8. **rect_intersection** - Implement `int rect_intersect(float x1, float y1, float w1, float h1, float x2, float y2, float w2, float h2, float* out_x, float* out_y, float* out_w, float* out_h)` that computes intersection of two axis-aligned rectangles. Return 1 if intersects, 0 otherwise.
9. **convex_hull_2d** - Implement `int convex_hull(const float* points_xy, int n, int* hull_indices)` using Graham scan algorithm for 2D points. Return number of hull vertices.

Algorithms - Data Structures

10. **quicksort** - Implement `void quicksort(int* arr, int n)` using in-place partitioning.
11. **rolling_hash** - Implement `unsigned long rolling_hash(const char* str, int window_size, unsigned long* hashes, int* count)` that computes all rolling hashes of given window size using Rabin-Karp polynomial rolling hash.

Hash/Crypto

12. **crc32** - Implement `unsigned int crc32(const unsigned char* data, size_t len)` using table-lookup method (IEEE polynomial 0x04C11DB7).

Graph Algorithms

13. **adjacency_matrix_transpose** - Implement `void transpose_graph(const int* adj, int* adj_t, int n)` that transposes an adjacency matrix representation of a directed graph.

CPU Features

14. **cpuid_features** - Implement `unsigned long get_cpu_features(void)` that detects and returns a bitmask of CPU features (SSE, AVX, AVX2, AVX512F, AES-NI, etc.) using CPUID.
15. **memcmp_avx2** - Implement `int memcmp(const void* s1, const void* s2, size_t n)` with AVX2 optimization. Compare performance with scalar version on aligned/unaligned data.

Hard Exercises (Multiple evenings)

Advanced Math & SIMD

1. **arctan_avx** - Implement `void arctan_avx(const float* input, float* output, int n)` computing arctangent using Taylor series or CORDIC algorithm with AVX/AVX512 instructions. Compare precision and performance with `atanf` from libm for n=1000000 random inputs in [-1, 1].
2. **matrix_multiply_avx512** - Implement `void matmul_avx512(const float* A, const float* B, float* C, int n)` using AVX512 instructions with loop tiling/blocking optimization. Compare with AVX2 version. Test with n = 128, 256, 512, 1024.
3. **fft_complex** - Implement `void fft(float* real, float* imag, int n)` computing Fast Fourier Transform (Cooley-Tukey radix-2) for complex numbers. n must be power of 2. Test with n = 256, 1024.

String/Encoding

4. **base64_encode_decode** - Implement both `int base64_encode(const unsigned char* input, int len, char* output)` and `int base64_decode(const char* input, unsigned char* output, int* out_len)`. Optimize encoding with SIMD. Test with random binary data and text files.
5. **regex_simple** - Implement `int regex_match(const char* pattern, const char* text)` supporting only: literal characters, `.` (any char), `*` (zero or more of previous), `^` (start), `$` (end). Use backtracking algorithm.

Computational Geometry - Advanced

6. **line_segment_intersection** - Implement `int segment_intersect(float x1, float y1, float x2, float y2, float x3, float y3, float x4, float y4, float* ix, float* iy)` that finds intersection point of two line segments using parametric equations. Return 1 if intersects, 0 if parallel/no intersection.
7. **point_in_polygon** - Implement `int point_in_polygon(float px, float py, const float* vertices_xy, int n)` using ray-casting algorithm to determine if point is inside polygon. Test with concave and convex polygons.

Algorithms - Advanced

8. **floyd_warshall** - Implement `void floyd_warshall(int* dist, int n)` computing all-pairs shortest paths. Input/output is n×n adjacency matrix (INT_MAX for no edge). Test with graphs of n = 50, 100, 200 vertices.
9. **merge_sort_avx** - Implement `void mergesort(int* arr, int n)` with AVX2-optimized merge operation for sorting integers. Compare performance with scalar mergesort on arrays of size 10000, 100000, 1000000.

Lock-Free Data Structures

10. **treiber_stack** - Implement lock-free stack using Treiber's algorithm with `void push(struct node** head, long value)` and `int pop(struct node** head, long* value)` using CMPXCHG (compare-and-swap). Test with single-threaded correctness (push/pop 10000 elements). Note: focus on correct assembly implementation of CAS loop, actual concurrency testing optional.

Testing Notes

Test Data Sources:

- **String operations**: Generate random ASCII strings, use Lorem Ipsum text, Project Gutenberg texts
- **Math**: Use known sequences (OEIS), WolframAlpha for verification
- **Matrices**: Generate random matrices, use identity/zero matrices for edge cases
- **Geometry**: Generate random points/rectangles, use known geometric configurations
- **Graphs**: Use adjacency matrices from standard graph datasets (SNAP, NetworkX generated graphs)
- **Numerical**: Compare against `libm` functions, use known mathematical constants

Performance Testing:

- Use RDTSC for cycle-accurate timing
- Repeat tests 100-1000 times and take median
- Test on cold and warm cache
- Use `perf stat` for hardware counter analysis

AVX512 Note:
Your AMD CPU might not have AVX512. Check with:

```
grep avx512 /proc/cpuinfo
```

If not available, substitute AVX512 exercises with AVX2 equivalents or advanced AVX2 optimizations (e.g., better cache utilization, loop unrolling).

Calling Convention:
All functions follow System V AMD64 ABI (Linux x64):

- Integer/pointer args: RDI, RSI, RDX, RCX, R8, R9
- Float args: XMM0-XMM7
- Return: RAX (integer), XMM0 (float)
- Caller-saved: RAX, RCX, RDX, RSI, RDI, R8-R11, XMM0-XMM15
- Callee-saved: RBX, RBP, R12-R15

Good luck with your assembly programming practice!