ARM CPU中SIMD特性介绍

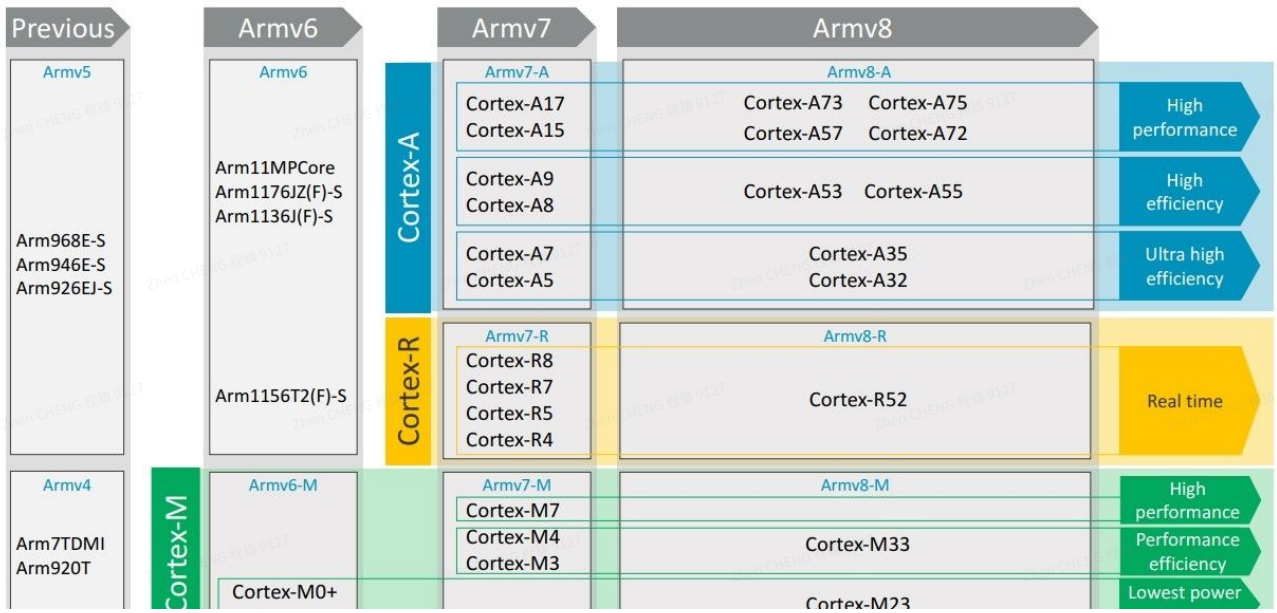
Arm架构

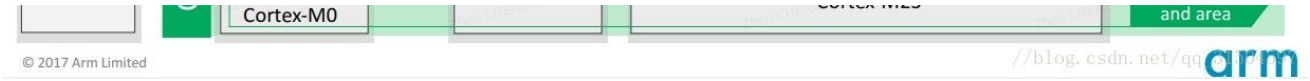
定义

- Advanced RISC Mechines
- ARM处理器非常适用于移动通信领域，符合其主要设计目标为低成本、高性能、低功耗的特性。

处理器系列

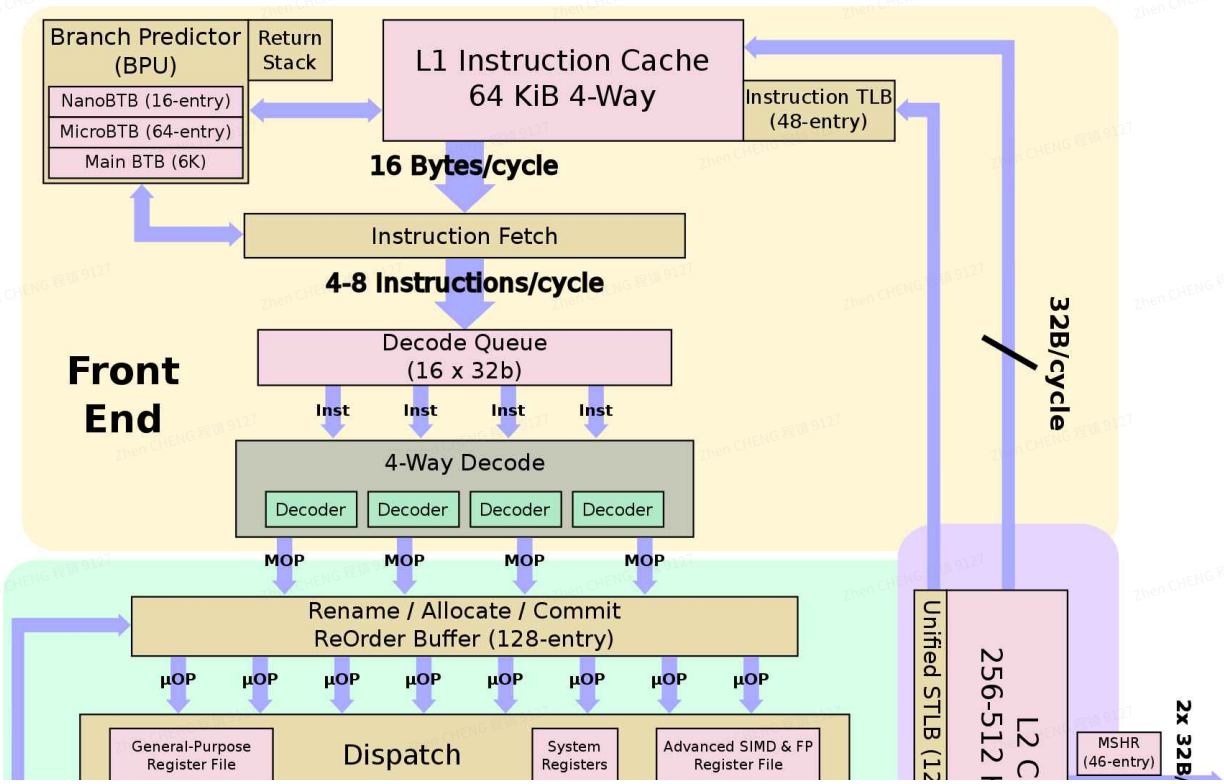
- **Application**
  - 用于具有高计算要求、运行丰富操作系统及提供交互媒体和图形体验的应用领域，如智能手机、平板电脑、汽车娱乐系统、数字电视，智能本、电子阅读器、家用网络、家用网关和其他各种产品。
- **Real-time**
  - 针对需要运行实时操作的系统应用，面向如汽车制动系统、动力传动解决方案、大容量存储控制器等深层嵌入式实时应用。
- **Microcontroller**
  - 面向微控制器领域，主要针对成本和功耗敏感的应用，如智能测量、人机接口设备、汽车和工业控制系统等。
- **SecurCore**
  - 主打安全的Cortex-SC系列，主要用于安全芯片。

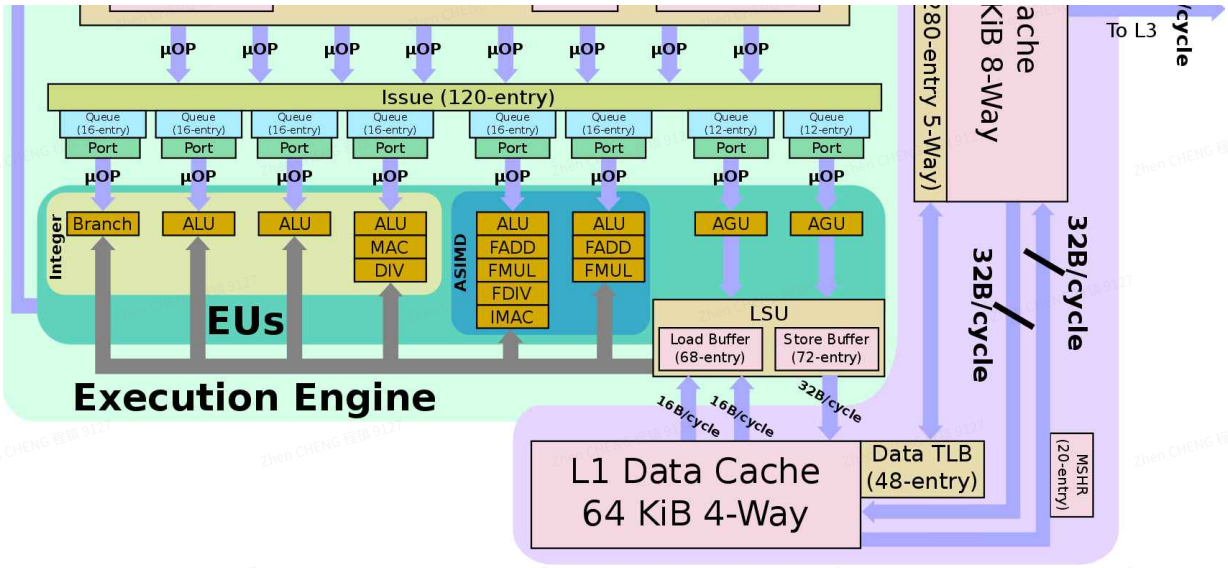




ARMv8体系结构

- ARMv8的一个重要特性是向后兼容，ARMv8架构支持:
- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32, 这与以前版本的ARM架构兼容
- Cortex A76
  - 4-way superscalar out-of-order processor.
  - 8-issue back end.
  - 64 KiB L1D and L1I, configurable L2 Cache, 256KiB-512KiB.





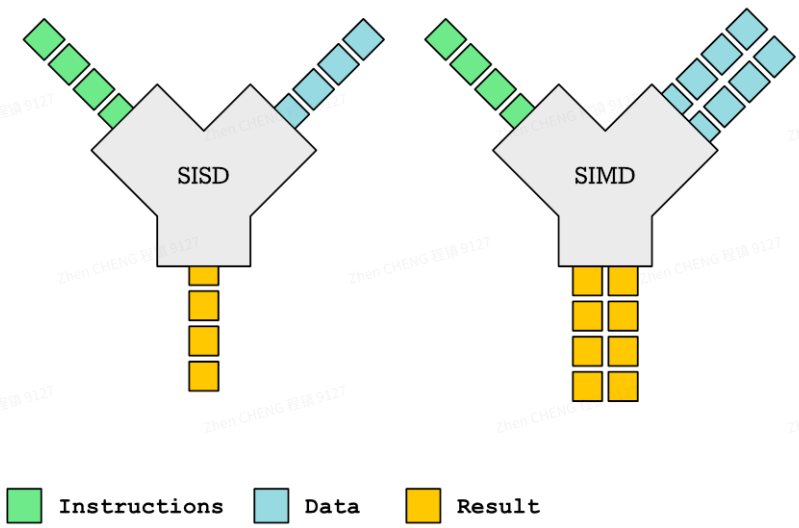
Armv8 Neon

SISD vs. SIMD

- Single Instruction Stream Single Data Stream(SISD)
  - 即单个指令处理单个数据
- Single Instruction Stream Multiple Data Stream(SIMD)
  - 即单个指令处理多条数据。以单精度浮点数加法操为例，一个128bit寄存器能够保存4个float，那么SIMD加法能够同时计算4对float的和。

Assembly language

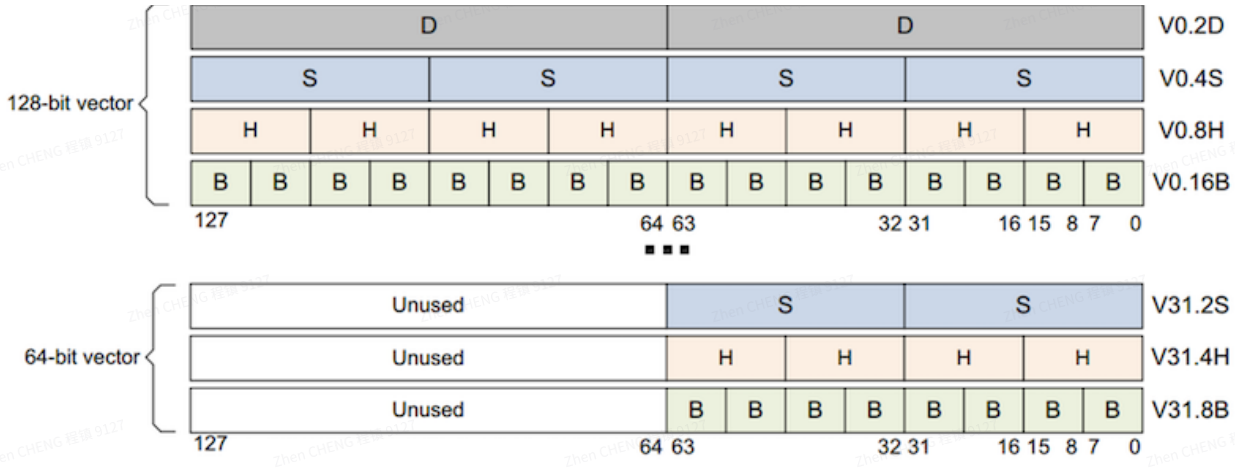
```
FADD V10.4S, V8.4S, V9.4S

// equal to
V10.4S[0] = V9.4S[0] + V8.4S[0]
V10.4S[1] = V9.4S[1] + V8.4S[1]
V10.4S[2] = V9.4S[2] + V8.4S[2]
V10.4S[3] = V9.4S[3] + V8.4S[3]
// for example
v8 = [1, 1, 2, 2]
v9 = [3, 4, 4, 4]
v10 = [4, 5, 6, 6]
```

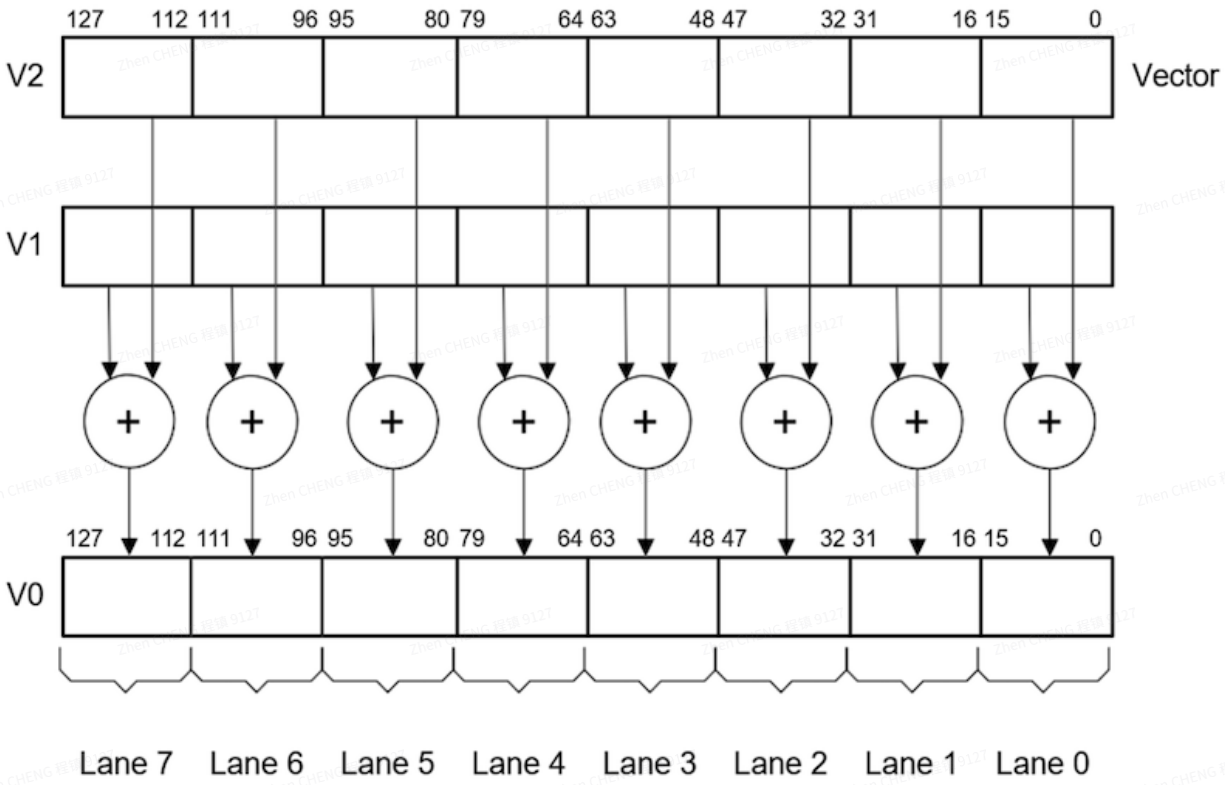


- SIMD特别适用于一些常见的任务，如音频图像处理。大部分现代 CPU 设计都包含了 SIMD 指令，来提高多媒体计算性能。
- ARM Neon是适用于 ARM Cortex-A 和 Cortex-R 系列处理器的一种 Advanced SIMD扩展架构。

硬件实现

- 向量寄存器
  - Neon寄存器可以看作包含相同数据类型的向量，假定该向量一共有n个元素，每条Neon指令都独立且并行作用在这n个元素上。对于128位Neon寄存器而言，如果存放的是 8-bit 元素，则n = 16,或者说这个寄存器被分成了16个lane,每个元素都坐落在一个lane上。
  - 内存中的数据装入Neon寄存器时，是低地址内存数据放入Neon寄存器的低地址上，高地址内存数据放入neon寄存器的高地址上。当Neon寄存器数据装入内存中时，同样是Neon寄存器的低地址数据放入的内存低地址上，Neon寄存器的高地址数据放入的内存高地址上。所以内存中的数据经过Neon处理后，数据的顺序是不会发生变化的。





- 指令Cycles

3.11. FP Data Processing Instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Utilized Pipelines	Notes
FP absolute value	FABS	2	2	V	
FP arithmetic	FADD, FSUB	2	2	V	
FP compare	FCCMP{E}, FCMP{E}	2	1	V0	
FP divide, H-form	FDIV	7	4/7	V0	1
FP divide, S-form	FDIV	7 to 10	4/9 to 4/7	V0	1
FP divide, D-form	FDIV	7 to 15	1/7 to 2/7	V0	1
FP min/max	FMIN, FMINNM, FMAX, FMAXNM	2	2	V	
FP multiply	FMUL, FNMUL	3	2	V	2
FP multiply accumulate	FMADD, FMSUB, FNMADD, FNMSUB	4 (2)	2	V	3
FP negate	FNEG	2	2	V	
FP round to integral	FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ	3	1	V	
FP select	FCSEL	2	2	V	
FP square root, H-form	FSQRT	7	4/7	V0	1
FP square root, S-form	FSQRT	7 to 10	4/9 to 4/7	V0	1
FP square root, D-form	FSQRT	7 to 17	1/8 to 2/7	V0	1

3.16. ASIMD Floating-Point Instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Utilized Pipelines	Notes
ASIMD FP absolute value/difference	FABS, FABD	2	2	V	
ASIMD FP arith, normal	FABD, FADD, FSUB, FADDP	2	2	V	
ASIMD FP compare	FACGE, FACGT, FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT	2	2	V	
ASIMD FP convert, long (F16 to F32)	FCVTL(2)	4	1/2	V0	
ASIMD FP convert, long (F32 to F64)	FCVTL(2)	3	1	V0	
ASIMD FP convert, narrow (F32 to F16)	FCVTN(2)	4	1/2	V0	
ASIMD FP convert, narrow (F64 to F32)	FCVTN(2), FCVTXN(2)	3	1	V0	
ASIMD FP convert, other, D-form F32 and Q-form F64	FCVTAS, FCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU, SCVTF, UCVTF	3	1	V0	
ASIMD FP convert, other, D-form F16 and Q-form F32	FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU, SCVTF, UCVTF	4	1/2	V0	
ASIMD FP convert, other, Q-form F16	FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU, SCVTF, UCVTF	6	1/4	V0	
ASIMD FP divide, D-form, F16	FDIV	7	1/7	V0	3
ASIMD FP divide, D-form, F32	FDIV	7 to 10	2/9 to 2/7	V0	3
ASIMD FP divide, Q-form, F16	FDIV	10 to 13	1/13 to 1/10	V0	3
ASIMD FP divide, Q-form, F32	FDIV	7 to 10	1/9 to 1/7	V0	3

ASIMD FP divide, Q-form, F64	FDIV	7 to 15	1/14 to 1/7	V0	3

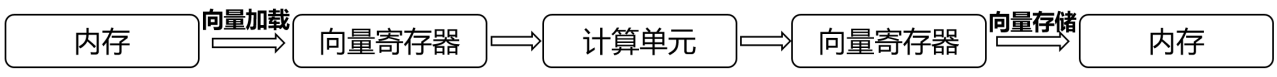
Neon指令

- 常用指令

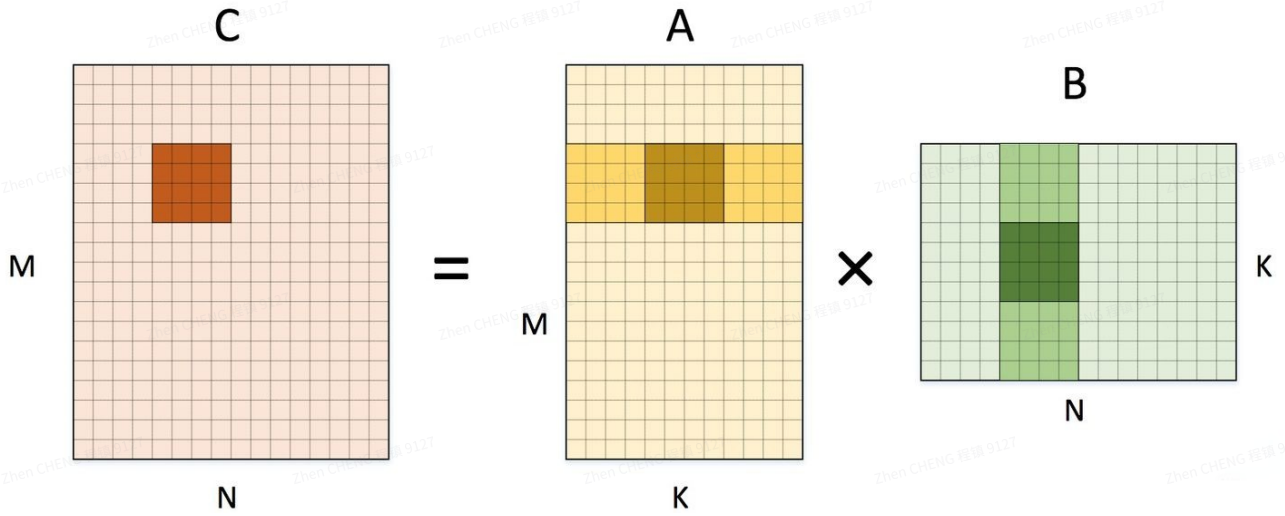
Assembly language

```
// Coding for Neon - Arm Developer

// compute
FADD Vd.4S, Vn.4S, Vm.4S
FADD Vd.2S, Vn.2S, Vm.2S
FMLA Vd.4S, Vn.4S, Vm.4S
FMLA Vd.4S, Vn.4S, Vm.S[2]
FDIV Vd.4S,Vn.4S,Vm.4S // armv8

// load/store
DUP Vd.4S,rn
LD1 {Vt.4S},[Xn]
ST1 {Vt.4S},[Xn]
LD3 {Vt.16B - Vt3.16B},[Xn]
// example
// load [rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb,rgb]
// into
// [r,r,r,r,r,r,r,r,r,r,r,r,r,r,r,r]
// [g,g,g,g,g,g,g,g,g,g,g,g,g,g,g,g]
// [b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b]
"ld3  {v3.16b-v5.16b}, [%[src]], #48      \n"
```

- Neon指令执行流程



实例: 分块矩阵乘



- Naive

C++

```cpp
//A[m][k] 行大小 K
//B[k][n] 行大小 N
//C[m][n] 行大小 N
void BasicSgemm(const float* A, const float* B, float* C, int M, int N, int K) {
  for (int m = 0; m < M; ++m) {
    for (int n = 0; n < N; ++n) {
      float c_mn = 0;
      for (int k = 0; k < K; ++k) {
        c_mn += A[m*K+k] * B[k*N+n]; // C[m][n] += A[m][k] * B[k][n]
      }
      C[m*N+n] = c_mn;
    }
  }
}
#define BLOCK_SIZE_R 8
#define BLOCK_SIZE_C 4
void SgemmBlkNaive(const float* A, const float* B, float* C, int M, int N, int K) {
  for (int m = 0; m < M; m+=BLOCK_SIZE_R) {
    for (int n = 0; n < N; n+=BLOCK_SIZE_R) {
      for (int k = 0; k < K; k+=BLOCK_SIZE_C) {
        do_block(A+m*K+k, B+k*N+n, C+m*N+n, M, N, K);
      }
    }
  }
}
void do_block(const float* A, const float* B, float* C,int M, int N, int K){
    for(int ml=0; ml<BLOCK_SIZE_R; ml++){
        for(int kl=0; kl<BLOCK_SIZE_C; kl++){
            for(int nl=0; nl<BLOCK_SIZE_R; nl++){
                C[ml*N+nl] += A[ml*K+kl] * B[kl*N+nl];
            }
        }
    }
}
```

- Neon

C

```cpp
void do_block_neon_asm(const float* A, const float* B, float* C,int M, int N, int K){
  for(int ml=0; ml<BLOCK_SIZE_R; ml+=4){
    for(int kl=0; kl<BLOCK_SIZE_C; kl+=4){
      for(int nl=0; nl<BLOCK_SIZE_R; nl+=4){
        __asm__ volatile(
```

```
 6              "madd x0, %[ml], %[N], %[nl]         \n" // x0 = c0 = ml*N+nl;
 7              "madd x1, %[ml], %[K], %[kl]         \n" // x1 = a0 = ml*K+kl;
 8              "madd x2, %[kl], %[N], %[nl]         \n" // x2 = b0 = kl*N+nl;
 9              "add x3, x0, %[N]                    \n" // x3 = c1 = (ml+1)*N+nl;
10              "add x4, x1, %[K]                    \n" // x4 = a1 = (ml+1)*K+kl;
11              "add x5, x2, %[N]                    \n" // x5 = b1 = (kl+1)*N+nl;
12              "add x6, x3, %[N]                    \n" // x6 = c2 = (ml+2)*N+nl;
13              "add x7, x4, %[K]                    \n" // x7 = a2 = (ml+1)*K+kl;
14              "add x8, x5, %[N]                    \n" // x8 = b2 = (kl+1)*N+nl;
15              "add x9, x6, %[N]                    \n"
16              "add x10, x7, %[K]                   \n"
17              "add x11, x8, %[N]                   \n"
18              "add x0, %[C], x0, lsl #2            \n" // C+c0
19              "add x1, %[A], x1, lsl #2            \n" // A+a0
20              "add x2, %[B], x2, lsl #2            \n" // B+b0
21              "add x3, %[C], x3, lsl #2            \n" // C+c1
22              "add x4, %[A], x4, lsl #2            \n" // A+a1
23              "add x5, %[B], x5, lsl #2            \n" // B+b1
24              "add x6, %[C], x6, lsl #2            \n" // C+c2
25              "add x7, %[A], x7, lsl #2            \n" // A+a2
26              "add x8, %[B], x8, lsl #2            \n" // B+b2
27              "add x9, %[C], x9, lsl #2            \n" // C+c3
28              "add x10, %[A], x10, lsl #2          \n" // A+a3
29              "add x11, %[B], x11, lsl #2          \n" // B+b3
30
31              "ld1 {v0.4s}, [x0]                   \n" // vc0
32              "ld1 {v1.4s}, [x1]                   \n" // va0
33              "ld1 {v2.4s}, [x2]                   \n" // vb0
34              "ld1 {v3.4s}, [x3]                   \n" // vc1
35              "ld1 {v4.4s}, [x4]                   \n" // va1
36              "ld1 {v5.4s}, [x5]                   \n" // vb1
37              "ld1 {v6.4s}, [x6]                   \n" // vc2
38              "ld1 {v7.4s}, [x7]                   \n" // va2
39              "ld1 {v8.4s}, [x8]                   \n" // vb2
40              "ld1 {v9.4s}, [x9]                   \n" // vc3
41              "ld1 {v10.4s}, [x10]                 \n" // va3
42              "ld1 {v11.4s}, [x11]                 \n" // vb3
43
44              "fmla v0.4s, v2.4s, v1.s[0]          \n" // vc0 =
    vfmaq_laneq_f32(vc0, vb0, va0, 0);
45              "fmla v0.4s, v5.4s, v1.s[1]          \n"
46              "fmla v0.4s, v8.4s, v1.s[2]          \n"
47              "fmla v0.4s, v11.4s, v1.s[3]         \n"
48
49              "fmla v3.4s, v2.4s, v4.s[0]          \n"
50              "fmla v3.4s, v5.4s, v4.s[1]          \n"
51              "fmla v3.4s, v8.4s, v4.s[2]          \n"
52              "fmla v3.4s, v11.4s, v4.s[3]         \n"
```

```
53
54              "fmla v6.4s, v2.4s, v7.s[0]         \n"
55              "fmla v6.4s, v5.4s, v7.s[1]         \n"
56              "fmla v6.4s, v8.4s, v7.s[2]         \n"
57              "fmla v6.4s, v11.4s, v7.s[3]        \n"
58
59              "fmla v9.4s, v2.4s, v10.s[0]        \n"
60              "fmla v9.4s, v5.4s, v10.s[1]        \n"
61              "fmla v9.4s, v8.4s, v10.s[2]        \n"
62              "fmla v9.4s, v11.4s, v10.s[3]       \n"
63
64              "st1 {v0.4s}, [x0]                  \n"
65              "st1 {v3.4s}, [x3]                  \n"
66              "st1 {v6.4s}, [x6]                  \n"
67              "st1 {v9.4s}, [x9]                  \n"
68              :
69              :[A]       "r"(A),
70              [B]        "r"(B),
71              [C]        "r"(C),
72              [M]        "r"(M),
73              [N]        "r"(N),
74              [K]        "r"(K),
75              [ml]        "r"(ml),
76              [nl]        "r"(nl),
77              [kl]        "r"(kl)
78              :"cc", "memory", "x0", "x1", "x2", "x3", "x4", "x5", "x6", "x7",
    "x8", "x9", "x10", "x11",
79              "v0", "v1", "v2", "v3", "v4", "v5","v6", "v7", "v8",
    "v9","v10","v11"
80              );
81          }
82       }
83     }
84 }
```

Neon Intrinsics

- NEON Intrinsics 类似于 C 函数调用，在编译时由编译器替换为相应的汇编指令，使用时需要包含头文件 <arm_neon.h>
- Neon Intrinsics 常用函数

C

```c
// https://developer.arm.com/architectures/instruction-sets/intrinsics/#q=
    FADD Vd.4S, Vn.4S, Vm.4S
    float32x4_t vaddq_f32(float32x4_t a, float32x4_t b)

    FADD Vd.2S, Vn.2S, Vm.2S
    float32x2_t vadd_f32(float32x2_t a, float32x2_t b)

    FMLA Vd.4S, Vn.4S, Vm.4S
    float32x4_t vfmaq_f32(float32x4_t a, float32x4_t b, float32x4_t c)

    FMLA Vd.4S, Vn.4S, Vm.S[2]
    float32x4_t vfmaq_lane_f32(float32x4_t a, float32x4_t b, float32x2_t v, const int lane)

    DUP Vd.4S,rn
    float32x4_t vdupq_n_f32(float32_t value)

    LD1 {Vt.4S},[Xn]
    float32x4_t vld1q_f32(float32_t const * ptr)

    LD3 {Vt.16B - Vt3.16B},[Xn]
    float32x4x3_t vld3q_f32(float32_t const * ptr)

    ST1 {Vt.4S},[Xn]
    void vst1q_f32(float32_t * ptr, float32x4_t val)
```

- 优点
  - Neon Intrinsics 函数提供了一种编写Neon代码的方法，该方法比汇编代码更易于维护，同时仍然可以控制生成的Neon指令。
  - Neon Intrinsic 函数的编写类似于使用这些变量作为参数或返回值的函数调用。编译器做了一些通常与编写汇编语言相关的工作来生成汇编代码，同时可能会做一些优化，例如寄存器分配，重新排序指令等。
- 缺点：性能依赖编译器；不能做到精确控制指令
- 实例: 分块矩阵乘

C++

```cpp
void do_block_neon_intrinsics(const float* A, const float* B, float* C,int M, int N, int K){
    for(int ml=0; ml<BLOCK_SIZE_R; ml+=4){
        for(int kl=0; kl<BLOCK_SIZE_C; kl+=4){
            for(int nl=0; nl<BLOCK_SIZE_R; nl+=4){
                int c0 = ml*N+nl;
                int a0 = ml*K+kl;
```

```
                int b0 = kl*N+nl;
                float32x4_t vc0 = vld1q_f32(C+c0);
                float32x4_t va0 = vld1q_f32(A+a0);
                float32x4_t vb0 = vld1q_f32(B+b0);
                int c1 = (ml+1)*N+nl;
                int a1 = (ml+1)*K+kl;
                int b1 = (kl+1)*N+nl;
                float32x4_t vc1 = vld1q_f32(C+c1);
                float32x4_t va1 = vld1q_f32(A+a1);
                float32x4_t vb1 = vld1q_f32(B+b1);
                int c2 = (ml+2)*N+nl;
                int a2 = (ml+2)*K+kl;
                int b2 = (kl+2)*N+nl;
                float32x4_t vc2 = vld1q_f32(C+c2);
                float32x4_t va2 = vld1q_f32(A+a2);
                float32x4_t vb2 = vld1q_f32(B+b2);
                int c3 = (ml+3)*N+nl;
                int a3 = (ml+3)*K+kl;
                int b3 = (kl+3)*N+nl;
                float32x4_t vc3 = vld1q_f32(C+c3);
                float32x4_t va3 = vld1q_f32(A+a3);
                float32x4_t vb3 = vld1q_f32(B+b3);
                vc0 = vfmaq_laneq_f32(vc0, vb0, va0, 0);
                vc0 = vfmaq_laneq_f32(vc0, vb1, va0, 1);
                vc0 = vfmaq_laneq_f32(vc0, vb2, va0, 2);
                vc0 = vfmaq_laneq_f32(vc0, vb3, va0, 3);
                vc1 = vfmaq_laneq_f32(vc1, vb0, va1, 0);
                vc1 = vfmaq_laneq_f32(vc1, vb1, va1, 1);
                vc1 = vfmaq_laneq_f32(vc1, vb2, va1, 2);
                vc1 = vfmaq_laneq_f32(vc1, vb3, va1, 3);
                vc2 = vfmaq_laneq_f32(vc2, vb0, va2, 0);
                vc2 = vfmaq_laneq_f32(vc2, vb1, va2, 1);
                vc2 = vfmaq_laneq_f32(vc2, vb2, va2, 2);
                vc2 = vfmaq_laneq_f32(vc2, vb3, va2, 3);
                vc3 = vfmaq_laneq_f32(vc3, vb0, va3, 0);
                vc3 = vfmaq_laneq_f32(vc3, vb1, va3, 1);
                vc3 = vfmaq_laneq_f32(vc3, vb2, va3, 2);
                vc3 = vfmaq_laneq_f32(vc3, vb3, va3, 3);
                vst1q_f32(C+c0, vc0);
                vst1q_f32(C+c1, vc1);
                vst1q_f32(C+c2, vc2);
                vst1q_f32(C+c3, vc3);
            }
        }
    }
}
```

- 性能对比
  - gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1) Target: aarch64-linux-gnu
  - N=512; M=512; K=1024;
  - gcc -O3会自动打开 -ftree-vectorize选项。

	A	B	C	D	
1		Navie C	Navie C 4x8	Neon 4x8 Intrinsics	Nec
2	O0	1,232.74	1,356.74	1,223.74	
3	O1	769.92	240.29	31.17	
4	O2	730.24	214.71	30.26	
5	O3	221.23	33.5	29.47	

```
b4c:       d503201f        nop
           C[ml*N+nl] += A[ml*K+kl] * B[kl*N+nl];
b50:       ad404412        ldp     q18, q17, [x0]
b54:       aa0403e5        mov     x5, x4
b58:       3dc20010        ldr     q16, [x0, #2048]
b5c:       aa0303e1        mov     x1, x3
b60:       3dc20407        ldr     q7, [x0, #2064]
b64:       3dc40006        ldr     q6, [x0, #4096]
b68:       3dc40405        ldr     q5, [x0, #4112]
b6c:       3dc60004        ldr     q4, [x0, #6144]
b70:       3dc60403        ldr     q3, [x0, #6160]
b74:       3dc000a2        ldr     q2, [x5]
b78:       914004a5        add     x5, x5, #0x1, lsl #12
b7c:       ad400021        ldp     q1, q0, [x1]
b80:       4f821241        fmla    v1.4s, v18.4s, v2.s[0]
b84:       4f821220        fmla    v0.4s, v17.4s, v2.s[0]
b88:       4fa21201        fmla    v1.4s, v16.4s, v2.s[1]
b8c:       4fa210e0        fmla    v0.4s, v7.4s, v2.s[1]
b90:       4f8218c1        fmla    v1.4s, v6.4s, v2.s[2]
b94:       4f8218a0        fmla    v0.4s, v5.4s, v2.s[2]
b98:       4fa21881        fmla    v1.4s, v4.4s, v2.s[3]
b9c:       4fa21860        fmla    v0.4s, v3.4s, v2.s[3]
```

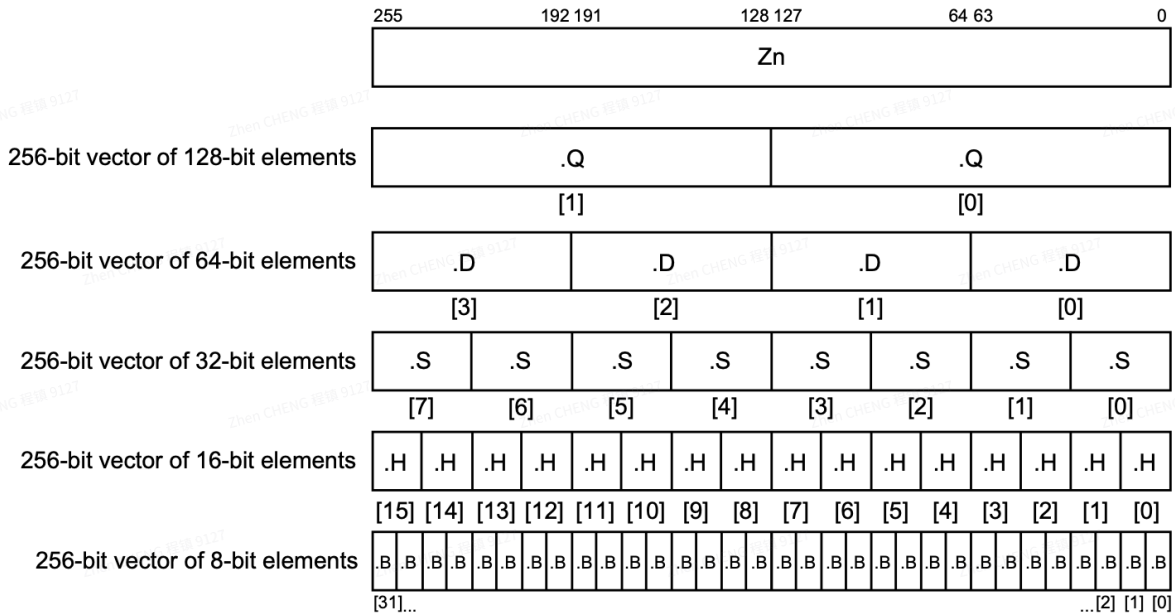
SVE

概念

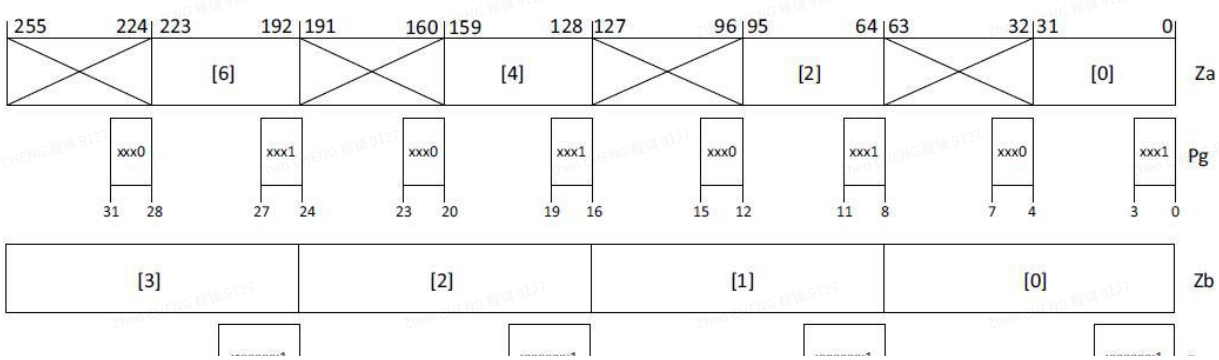
- Arm设计了 Scalable Vector Extension (SVE)作为AArch64的下一代SIMD扩展。SVE允许矢量寄存器长度可以在128到2048之间选择实现。它支持与矢量长度无关的编程模型，允许代码在所有矢量长度上自动运行和缩放，而无需重新编译。
- 设计目标: 加强向量并行处理能力以满足HPC，机器学习等领域的需求。

可变长寄存器

- Scalable vector registers
  - Z0-Z31, 128-2048 bits , 增量步长为128bit
  - 低128bit是和Neon的定长128bit vector(V0-V31)共享的。



- Scalable predicate registers
  - P0-P15，16-256bits，每个Px的长度是Zx的1/8，每一个Px的位对应Zx的字节
  - 可以看作SVE instructions的mask，大大提高了向量操作的灵活性。
  - 假设SVE的矢量长度Z为256bits，predicate register在管理32位和64位操作时，其视图如下。在控制32位数据操作时，如果Pg寄存器的最低为1，则该lane操作为激活状态，该lane操作结果被正常存储到目的寄存器；如果Pg寄存器的最低为0，则该lane操作为未激活状态，该lane操作结果不会被存储到目的寄存器。



- First Fault predicate Register (FFR)
  - 指示load/store指令是否成功
  - 通常是被first-fault load and store instructions赋值
- Configurable vector length
  - scalable vector control registers ZCR_El1, ZCR_El2, and ZCR_El3
  - ZCR_Elx.LEN指明了当前异常等级或者更低的异常等级下矢量寄存器的长度。
- 指令cycles

Table 3-25 SVE floating-point instructions

Instruction Group	SVE Instruction	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Floating point absolute value/difference	FABD, FABS	2	4	V	-
Floating point arithmetic	FADD, FADDP, FNEG, FSUB, FSUBR	2	4	V	-
Floating point associative add, F16	FADDA	10	1/9	V1	-
Floating point associative add, F32	FADDA	6	1/5	V1	-
Floating point associative add, F64	FADDA	4	4	V	-

Table 3-16 AArch64 ASIMD floating-point instructions

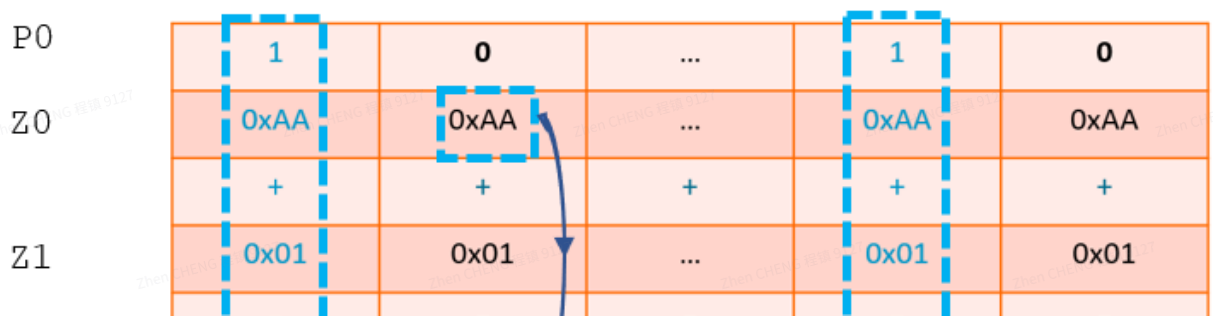
Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP absolute value/difference	FABS, FABD	2	4	V	-
ASIMD FP arith, normal	FADD, FSUB, FADDP	2	4	V	-
ASIMD FP compare	FACGE, FACGT, FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT	2	4	V	-
ASIMD FP complex add	FCADD	2	4	V	-

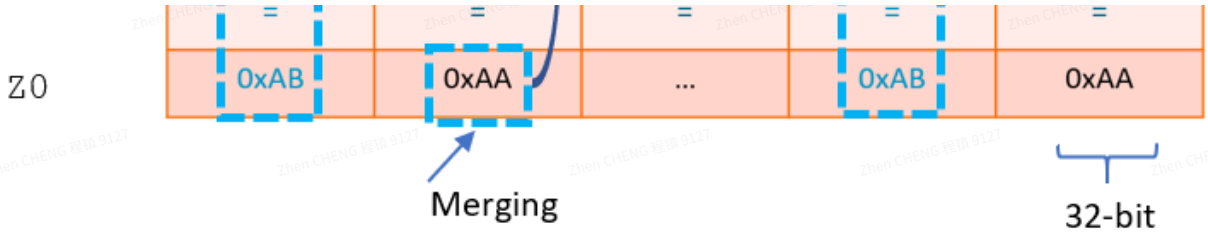
Table 3-11 AArch64 FP data processing instructions

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP absolute value	FABS	2	4	V	-
FP arithmetic	FADD, FSUB	2	4	V	-

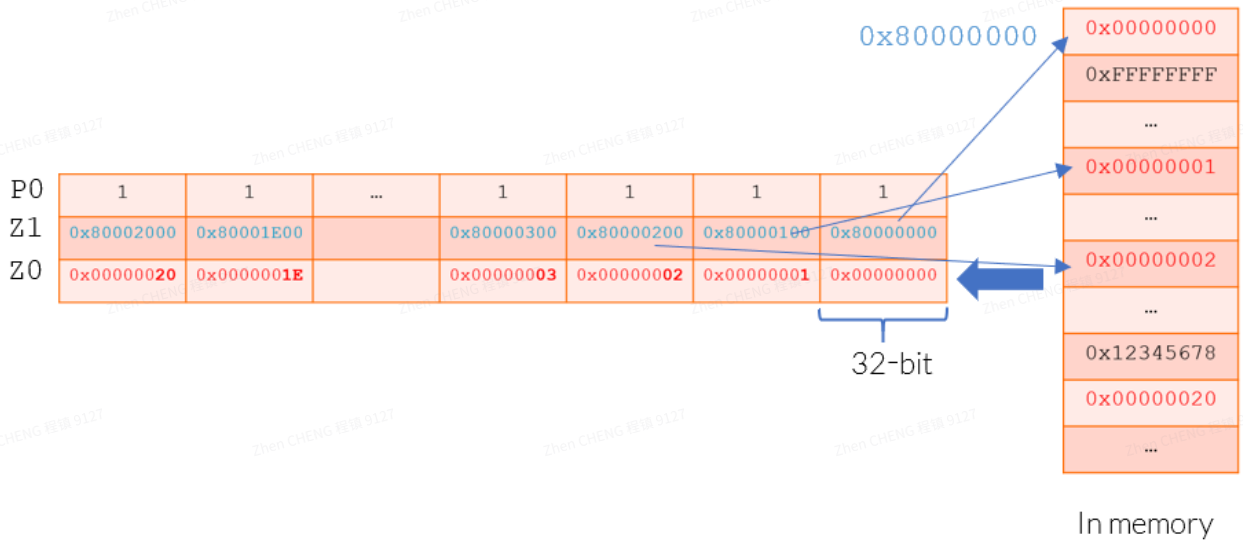
SVE的关键特性

- Per-lane predication
  - 通过使用predicate register来实现lane级操作
  - ADD Z0.S, P0/M, Z0.S, Z1.S





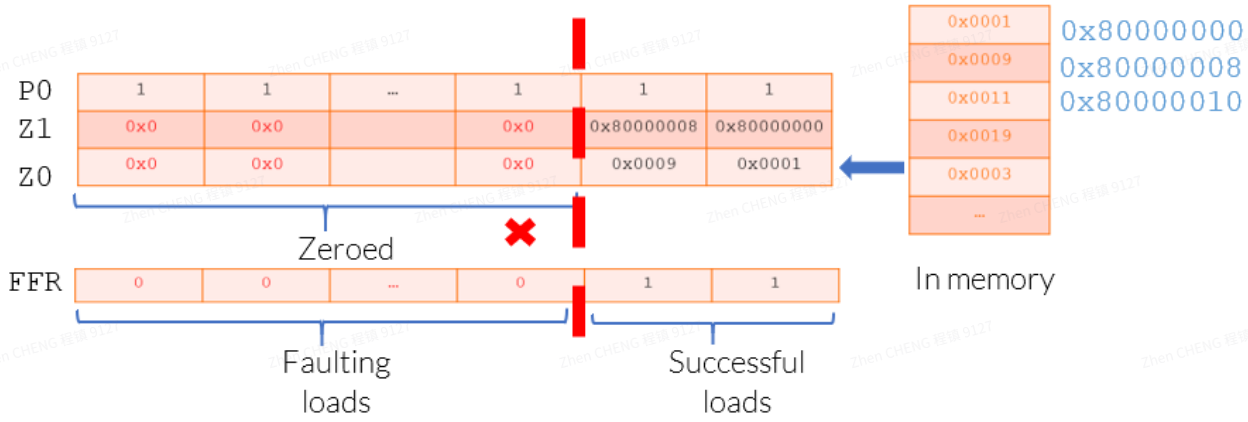
- Gather-load and scatter-store
  - 可以实现非连续内存的读写
  - LD1SB Z0.S, P0/Z, [Z1.S]



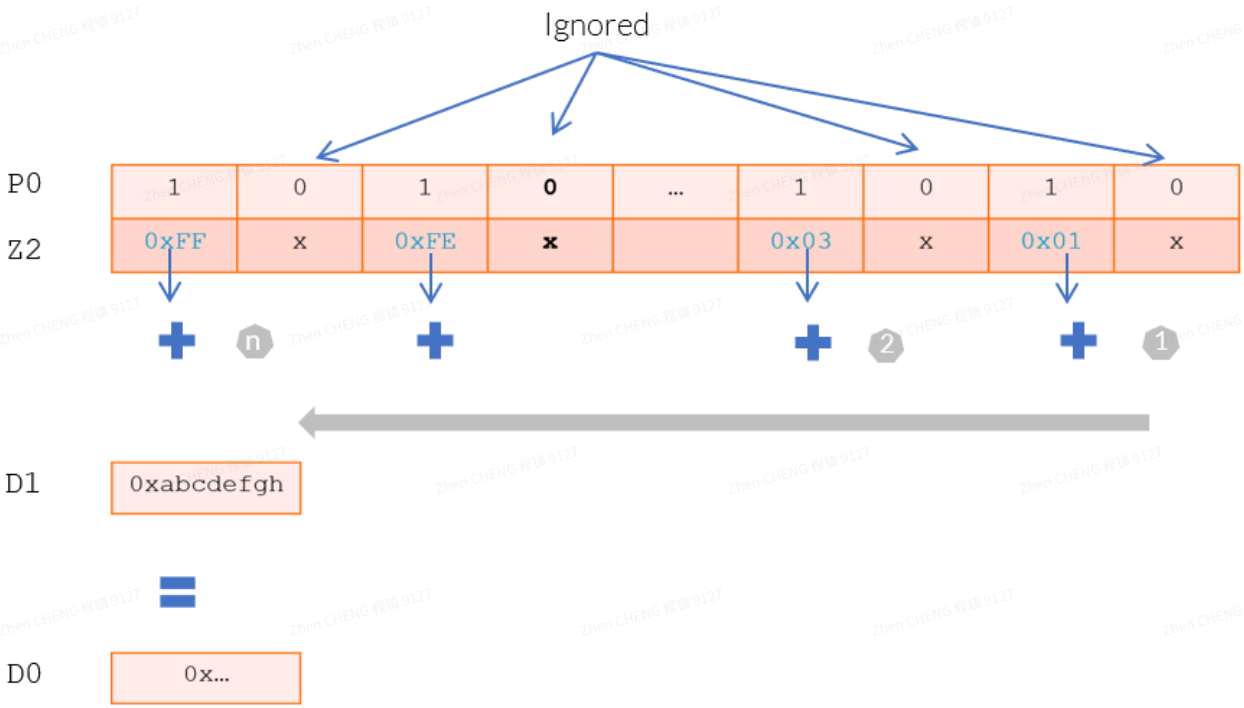
- Predicate-driven loop control and management
  - WHILELO P0.S, x8, x9
  - B.FIRST Loop_start

- Speculative loads

- LDFF1D Z0.D, P0/Z, [Z1.D, #0]



- Enhance floating-point and horizontal reduction
  - FADDA D0, P0/M, D1, Z2.D



实例：向量加权和

- SVE Intrinsics

C++

```cpp
void arrWeightedAvgSVEIntrinsic(const float *arr1,
                                const float arr1Weight,
                                const float *arr2,
                                const float arr2Weight,
                                const int len,
                                float *resultArr) {
    int32_t i = 0;
    svbool_t pg = svwhilelt_b32(i, len);
    svfloat32_t w1 = svdup_n_f32(arr1Weight);
    svfloat32_t w2 = svdup_n_f32(arr2Weight);
    do {
        svfloat32_t arr1_vec = svld1_f32(pg, &arr1[i]);
        svfloat32_t arr2_vec = svld1_f32(pg, &arr2[i]);
        svfloat32_t res_vec = svmul_lane_f32(arr1_vec, w1, 0);
        res_vec = svmla_lane_f32(res_vec, arr2_vec, w2, 0);
        svst1_f32(pg, &resultArr[i], res_vec);
        i += svcntw(); //
        pg = svwhilelt_b32(i, len); // update pg
    }while (svptest_any(svptrue_b32(), pg));
}

//  gcc test_sve.S -o test_sve -g -march=armv8-a+sve
```

Reference

- [SIMD](https://en.wikipedia.org/wiki/Single_instruction,_multiple_data#Hardware)
- [Cortex-A76体系结构](https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a76)
- [Arm® Cortex®-A76 Software Optimization Guide](https://developer.arm.com/documentation/pjdoc466751330-7215/latest/)
- [NEON Programmer's Guide](https://developer.arm.com/documentation/den0018/a/?lang=en)
- [neon intrinsics](https://developer.arm.com/architectures/instruction-sets/intrinsics/)
- [SVE Introduction](https://developer.arm.com/documentation/102476/0001/Introducing-SVE)
- [SVE intrinsics](https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=[sve])
- [The ARM Scalable Vector Extension](https://arxiv.org/pdf/1803.06185.pdf)

- [arm 汇编手册](https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec09_ARMisa.pdf)

附录

分块矩阵乘

向量加权和

RGB_8UC1ToBGR_8UC3