Background and trends of high performance processors

Processor trends:

- Manycore : more and more cores are integrated into a chip
  - ✓ Intel Knights Landing (KNL): 60~72core
- Wide SIMD (Single Instruction Multiple Data): more and more FPU are integrated into a core
  - ✓ AVX-2 (256-bit); Intel Xeon E5 v4 (Broadwell)
  - ✓ AVX512 (512-bit); Intel KNL, Xeon E5 v5 (Skylake)

No program compatibility between different SIMD length

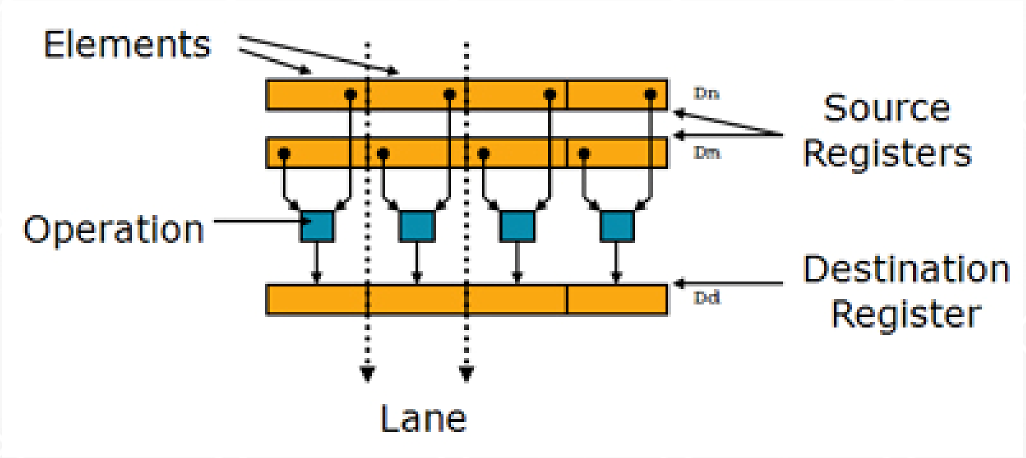
- Re-compile is required between AVX-2 and AVX512

Advanced SIMD (NEON) extension

- **Advanced SIMD (NEON) extension introduced in the ARMv7 ISA.**
  - NEON instructions operate on 64 or 128-bit wide vector data (SIMD data) held in the Advanced SIMD and FP register set.
  - They perform the same operation on all data elements (SIMD)

- **NEON instructions operate on 64 or 128-bit wide vectors with**
  - FX8/FX16/FX32/FX64 or
  - FP16/FP32/FP64

  data elements.



- **Remark**
  - For FP16 data NEON supports only conversions between FP16 and FP32/FP64 data.
  - FP16 data operations are supported only in the Advanced SIMDv2 option.

ARM v8 Scalable Vector Extension (SVE)

- **SVE is a complementary extension that does not replace NEON, and was developed specifically for vectorization of HPC scientific workloads.**
- **The new features and the benefits of SVE comparing to NEON**
  - **Scalable vector length (VL)** : Increased parallelism while allowing implementation choice of VL
  - **VL agnostic (VLA) programming**: Supports a programming paradigm of write-once, run-anywhere scalable vector code
  - **Gather-load & Scatter-store**: Enables vectorization of complex data structures with non-linear access patterns
  - **Per-lane predication**: Enables vectorization of complex, nested control code containing side effects and avoidance of loop heads and tails (particularly for VLA)
  - Predicate-driven loop control and management: Reduces vectorization overhead relative to scalar code
  - Vector partitioning and SW managed speculation: Permits vectorization of uncounted loops with data-dependent exits
  - Extended integer and floating-point horizontal reductions: Allows vectorization of more types of reducible loop-carried dependencies

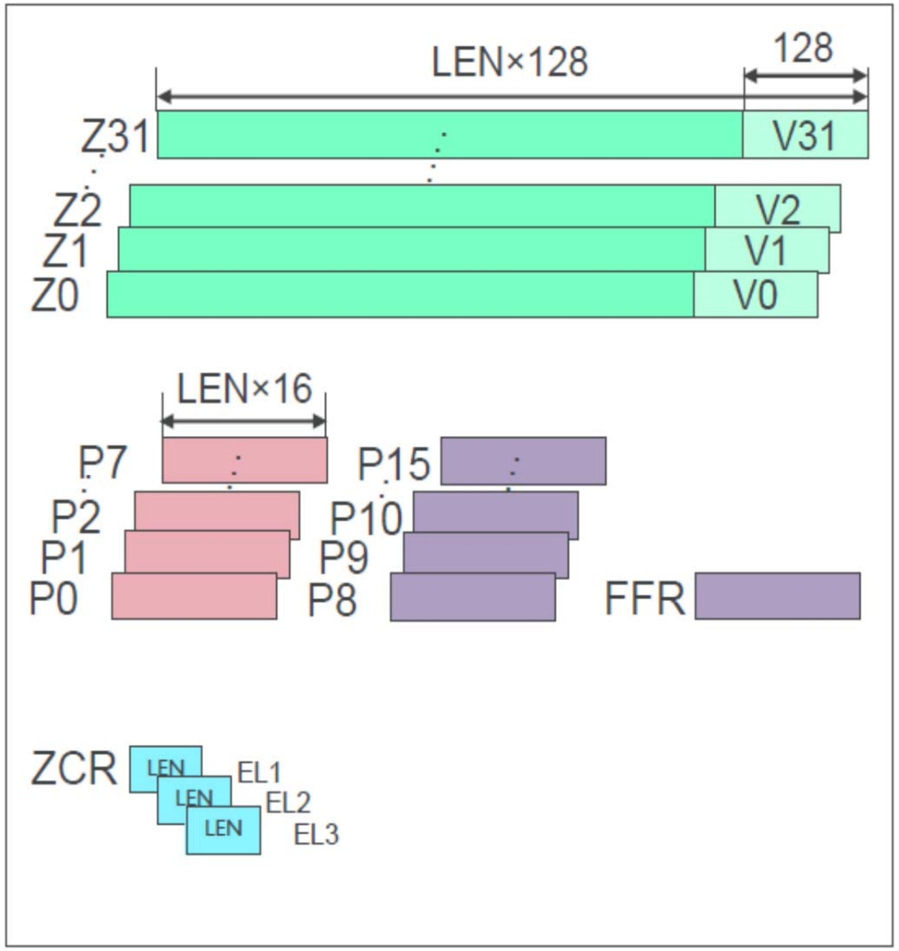
ARM v8 Scalable Vector Extension (SVE)

- **Vector Length Agnostic programming enables same binaries to run on different vector length environment because of programming independent of vector length**

- **Support 128bit～2048bit SIMD**
  - Each processor may support different vector length
  - SVE instructions don’t have vector length information, but refer the value of LEN implicitly.
  - LEN is in system register, that specifies current vector length
  - LEN=1:128bit, 2:256bit, 4:512bit, 8:1024bit, 16:2048bit
  - LEN can be changed by kernel call.

  - Post-K (Fugaku) processor (A64FX) announced to support 512bit.

SVE registers



- Zn is a SVE vector register.
- Vn is a NEON SIMD register, which is overlapped with the SVE vector register.
- When LEN=4(512bit), it is possible to perform the operation to 64bit x8, 32bits x16, 16bit x32, 8bit x64 in parallel.
- The instruction is defined for LEN=1～16, stored in a system register, ZCR, which is used to execute ISA implicitly.
- ZCR contains the vector length for each privileged level.
- Pn is a predicate Register, used as a mask to select active elements for the operation.

Vector length agnostic programming

```
for (int i = 0; i < N; i++)
  y[i] = 3.0 * x[i] + y[i];
```

This code runs with any vector length

Scalar

```
        fmov    d2, 3.0e+0
        mov     x0, 0 // int i
.L2:
        ldr     d0, [x2, x0] // x[i]
        ldr     d1, [x1, x0] // y[i]
        fmadd   d0, d0, d2, d1
        str     d0, [x1, x0] // y[i]
        add     x0, x0, 8 // i++
        cmp     x0, x3 // i < N?
        bne     .L2
```

SVE

```
        fmov    z0.d, #3.00000000
        whilelo p0.d, xzr, x9 // 0 < N?
.LBB0_1:
        ld1d    z1.d, p0/z, [x10, x8, lsl #3]
        ld1d    z2.d, p0/z, [x11, x8, lsl #3]
        fmad    z1.d, p0/m, z0.d, z2.d
        st1d    z1.d, p1, [x11, x8, lsl #3]
        incd    x8 // i+=(# of elements)
        whilelo p1.d, x8, x9 // i < N?
        b.first .LBB0_1 // p0[0] is true ?
```

This SVE code correctly runs for any N iterations, even if N is not the multiple of vector elements.

WHILELO instruction

- **WHILELO instruction generates a predicate vector according the condition on each vector elements**

ex) whilelo p1.d, x8, x9

When loop continues

	x8+7	x8+6	x8+5	x8+4	x8+3	x8+2	x8+1	x8
	<	<	<	<	<	<	<	<
	x9	x9	x9	x9	x9	x9	x9	x9
p1.d	1	1	1	1	1	1	1	1

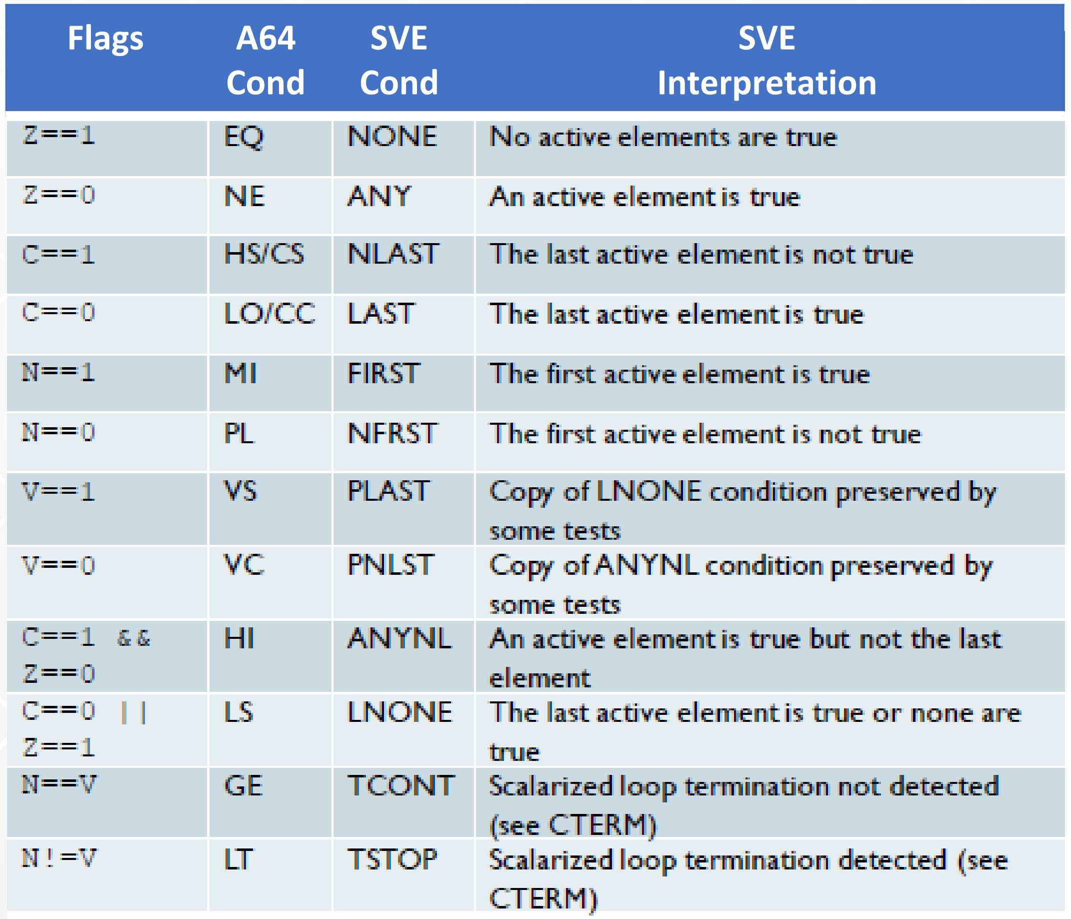
When loop terminates

	x8+7	x8+6	x8+5	x8+4	x8+3	x8+2	x8+1	x8
	>	>	>	=	<	<	<	<
	x9	x9	x9	x9	x9	x9	x9	x9
p1.d	0	0	0	0	1	1	1	1

Vector Loop Control & Branch instructions

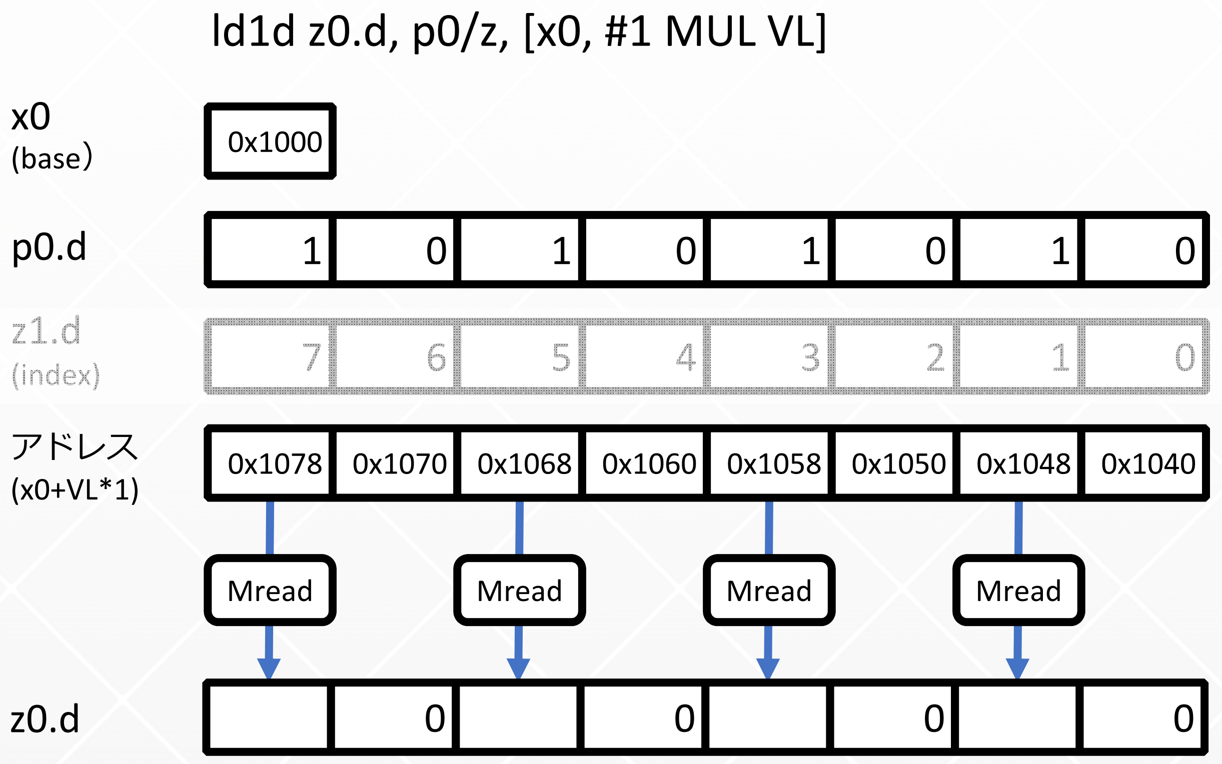
- Overload Pstate.NZCV rather than defining new set of branch instructions
- Set as a result of vector compares and predicate logical arithmetic

Flag	SVE	Set when
N	First	Set if first active predicate element is true
Z	None	Set if no active predicate elements are true
C	!Last	Cleared if last active predicate element is true
V	PLast	Preserves (Last || None) across certain flag-setting operations



Flags	A64 Cond	SVE Cond	SVE Interpretation
Z==1	EQ	NONE	No active elements are true
Z==0	NE	ANY	An active element is true
C==1	HS/CS	NLAST	The last active element is not true
C==0	LO/CC	LAST	The last active element is true
N==1	MI	FIRST	The first active element is true
N==0	PL	NFRST	The first active element is not true
V==1	VS	PLAST	Copy of LNONE condition preserved by some tests
V==0	VC	PNLST	Copy of ANYNL condition preserved by some tests
C==1 && Z==0	HI	ANYNL	An active element is true but not the last element
C==0 || Z==1	LS	LNONE	The last active element is true or none are true
N==V	GE	TCONT	Scalarized loop termination not detected (see CTERM)
N!=V	LT	TSTOP	Scalarized loop termination detected (see CTERM)

Contiguous Load instruction



- VL is 64 in byte when vector length is 512bits.

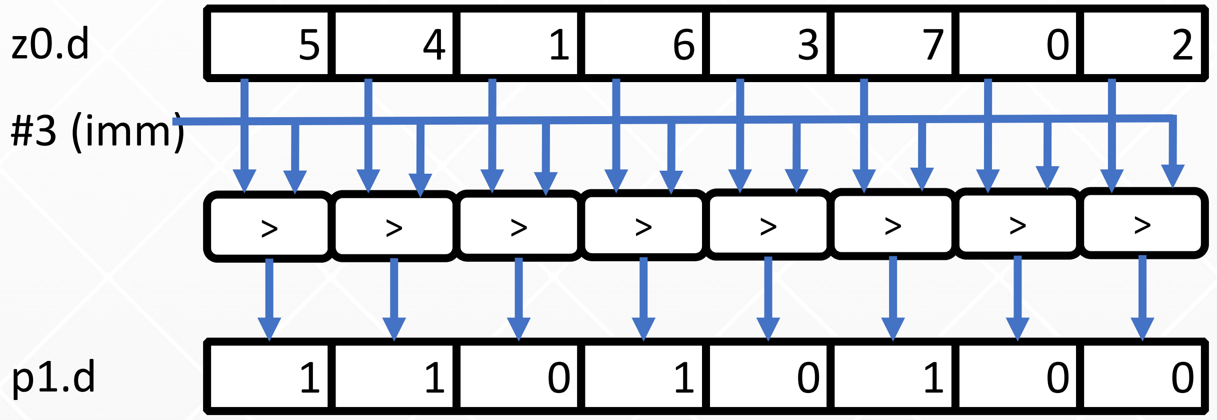
- Read vector data from memory by one instruction. Only active elements are read.
- Otherwise, inactive elements are cleared (/z)

Zeroing / Merging Predication

- **When a lane's predicate bit is "inactive" the destination lane can either be:**
  - **Merged (/m)**: leave the inactive lane unmodified
    - Useful for building a vector result in multiple steps (e.g. if/then/else, accumulation, serialised loops)
    - Previous value of destination register is an implicit read dependency
    - In a destructive encoding the previous value is already a dependency (ADD Zds1, Zds1, Zs2)
  - **Zeroed (/z)** : set the inactive lane to zero
    - Avoids dependency on destination for constructive 3-op encodings, but value is really "don't care"
    - A merged or computed value can also be "don't care", if no additional cost and no side effects
- **So:**
  - Most arithmetic instructions are destructive 2-op encodings with Merging predication
  - Commonly-used arithmetic instructions also have constructive 3-op Unpredicated forms
  - Most predicate-generating instructions (e.g. compare) have option for Merging or Zeroing

Predicate-setting instructions

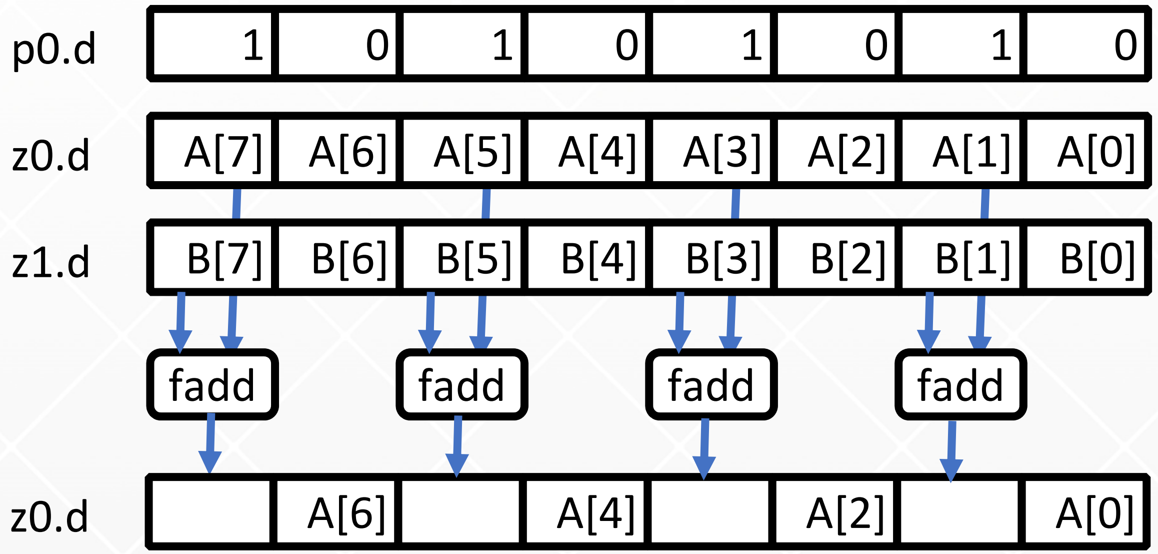
```
ptrue p0.d
cmpgt p1.d p0/z z0.d #3
```



- Ptrue sets all predicates “true”.
- Pflase sets all predicates “false”.
- Cmpgt compares each elements in vector with elements in the second vector or immediate value and set each flag of predicate.
- CMP+ EQ, NE, （signed） GT, GE, LT, LE, （unsiged） HI, HS, LO, LS

Arithmetic operation with predicate mask

fadd z0.d, p0/m, z0.d, z1.d



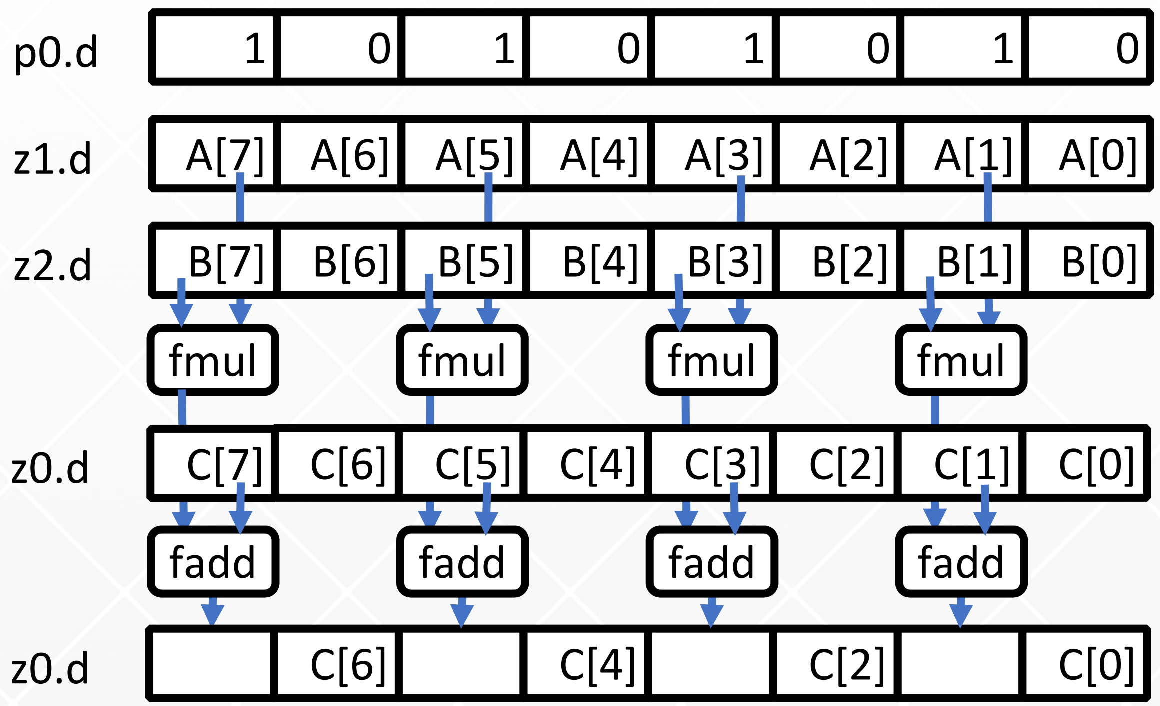
This example uses "double precision". There are other types; "single precision", "half precision", integer （64bit, 32bit, 16bit, 8bit）

- According to each value in the predicate, only active elements are calculated.
- If "/m" specified, inactive elements are unmodified. ("/z" for zero clear)
- The destination register must be the same as the first operand, that is "destructive" operation

fadd z2.d, z0.d, z1.d
"Constructive" 3 operands format if "un-predicated"

FMA(floating-point multiply-add) instruction

fmla z0.d, p0/m, z1.d, z2.d [Zda = Zda + Zn * Zm]



Both mul and add are executed in one instruction.

Other operations are provided:

FMAD [Zdn = Za + Zdn * Zm]

FMLS [Zda = Zda + -Zn * Zm]

FMSB [Zdn = Za + -Zdn * Zm]

FNMAD[Zdn = -Za + -Zdn * Zm]

FNMLA[Zda = -Zda + -Zn * Zm]

FNMLS[Zda = -Zda + Zn * Zm]

FNMSB[Zdn = -Za + Zdn * Zm]

These are “destructive”

- Only active elements are calculated
- Inactive elements remains (/m)
- Destination register are the same as 1st source operand, so that destructive!

Destructive vs Constructive (MOVPRFX instruction)

- **Predicated arithmetic instructions are generally destructive**

```
ADD Zds1, Pg/M, Zds1, Zs2 => Pg: Zds1 += Zs2
```

  - Constructivity plus predication is expensive for ISA encoding space
- **Constructive operations may be achieved with instruction pair, prefix move instruction "movprfx"**
  - The operation is the same as "move" instruction
  - The prefix move must immediately precede the instruction and it can be merged (the same destination, predicate, the size and type) , then the hardware can execute these operations as one construction operations

```
MOVPRFX Zd, Pg/M, Zs1
ADD Zd, Pg/M, Zd, Zs2
   =>  ADD Zd, Pg/M, Zs1, Zs2
       (so long as d != s2)
```

```
movprfx z4, p0/z, z0
fmla z4.d, p0/z, z1.d, z2.d
  => [Zda = Zda + Zn * Zm]
```

4 operand instruction!

Uses of Predication

- **Loops with control flow divergence (vector if-conversion)**
  - Each iteration of a scalar loop and hence each lane of a vector may follow different control path
  - Predication controls which elements take part in an operation (the active elements)
  - Prevents side-effects from unsafe values in inactive elements (floating-point, load/store, accumulation)
- **Loop heads/tails**
  - Avoid extra tests and scalar code for loop heads (alignment) and tails (non-VL multiples)
  - Predicate operations can set condition flags for loop control flow instructions
- **Vector partitioning**
  - Iterating through portions of a vector in response to speculative faults or serialised loops
  - Vector-length agnosticism is a special case of vector partitioning

IF-conversion with masking

C

```c
void ifcvt() {
  long i;
  for (i=0; i<256; i++){
    if (b[i] != 0.0) {
      if (c[i] == 0.0) {
        d[i] = a[i] + b[i];
      }
      c[i] = d[i] + b[i];
    }
  }
}
```

SVE (masking, omitting loop)

```
// z0.d = {i, i+1, i+2,}; p0 = partition
ld1d      z1.d, p0/z, [b, z0.d]     // load b[i]
fcmne     p1, p0/z, z1.d, #0.0      // test b[i]!=0.0
ld1d      z2.d, p1/z, [c, z0.d]     // load c[i] under p1
fcmeq     p2, p1/z, z2 .d, #0.0     // test c[i]==0.0
ld1d      z3.d, p2/z, [a, z0.d]     // load a[i] under p2
fadd      z3.d, p2/m, z3.d, z1.d    // a[i]+b[i] under p2
st1d      z3.d, p2, [d, z0.d]       // store d[i] under mask
bic       p3, p1, p2                // remaining d lanes in mem
ld1d      z4.d, p3/z, [d, z0.d]     // load org d[i] under p3
fadd      z1.d, p2/m, z1.d, z3.d    // b[i]+d[i] under p2
fadd      z1.d, p3/m, z1.d, z4.d    // b[i]+org d[i] under p3
st1d      z1.d, p1, [c, z0.d]       // store c[i] under p1
```

Load instruction: Addressing mode

4 kinds of addressing modes：

Contiguous Load

1. Scalar plus immediate

   ld1d z0.d, p0/z, [x0, #1, MUL VL]

2. Scalar plus scalar

   ld1d z1.d, p0/z, [x0, x1, LSL #3]

Gather load

3. Scalar plus vector
   a. 32bit unpacked scaled offset

      ld1d z0.d, p0/z, [x0, z1.d, sxtw #3]

   b. 32bit unpacked unscaled offset

      ld1d z0.d, p0/z, [x0, z1.d, sxtw]

   c. 64bit scaled offset

      ld1d z0.d, p0/z, [x0, z1.d, lsl #3]

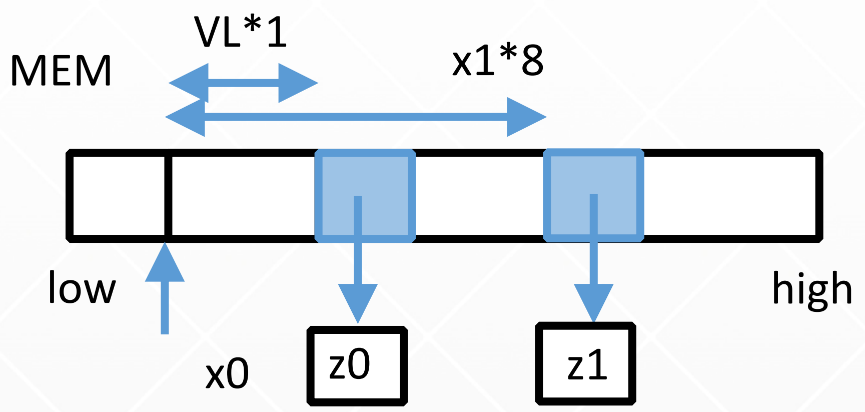
   d. 64bit unscaled offset

      ld1d z0.d, p0/z, [x0, z1.d]

4. Vector plus immediate

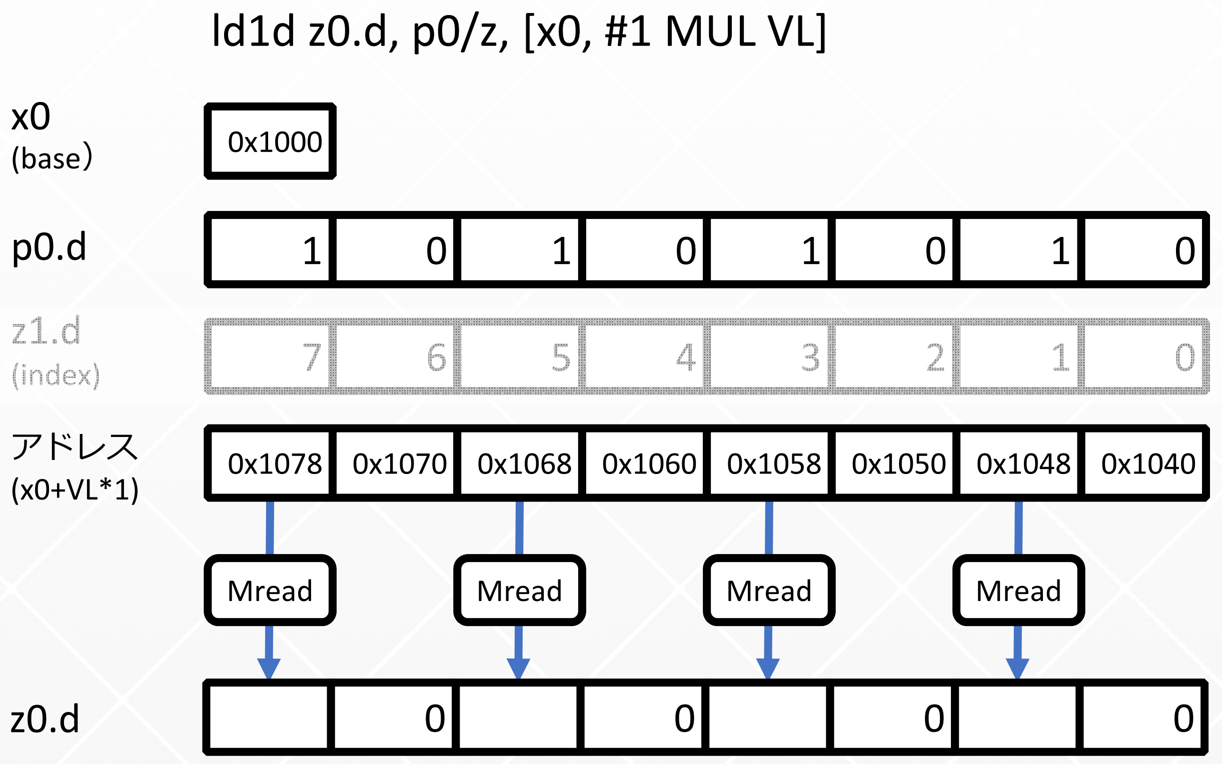
   ld1d z0.d, p0/z, [z1.d, #8]

Example: Contiguous load



- Instruction **mnemonic :** ld1 + [b,h,w,d] for each data size.
- Ld1 read memory data without sign extension. Ld1s+[b,h,w,d] is load instruction with sign-extension.

Contiguous Load instruction

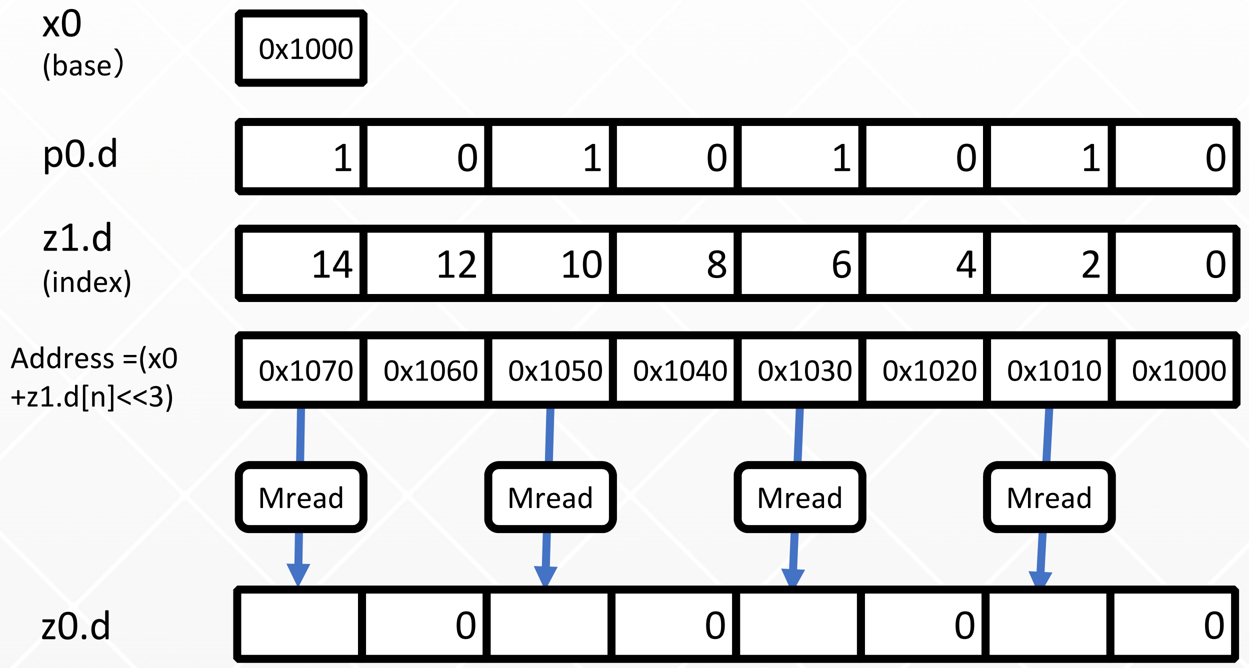


- VL is 64 in byte when vector length is 512bits.

- Read vector data from memory by one instruction. Only active elements are read.
- Otherwise, inactive elements are cleared (/z)

Gather load instruction

ld1d z0.d, p0/z, [x0, z1.d, lsl #3] // Scalar plus vector



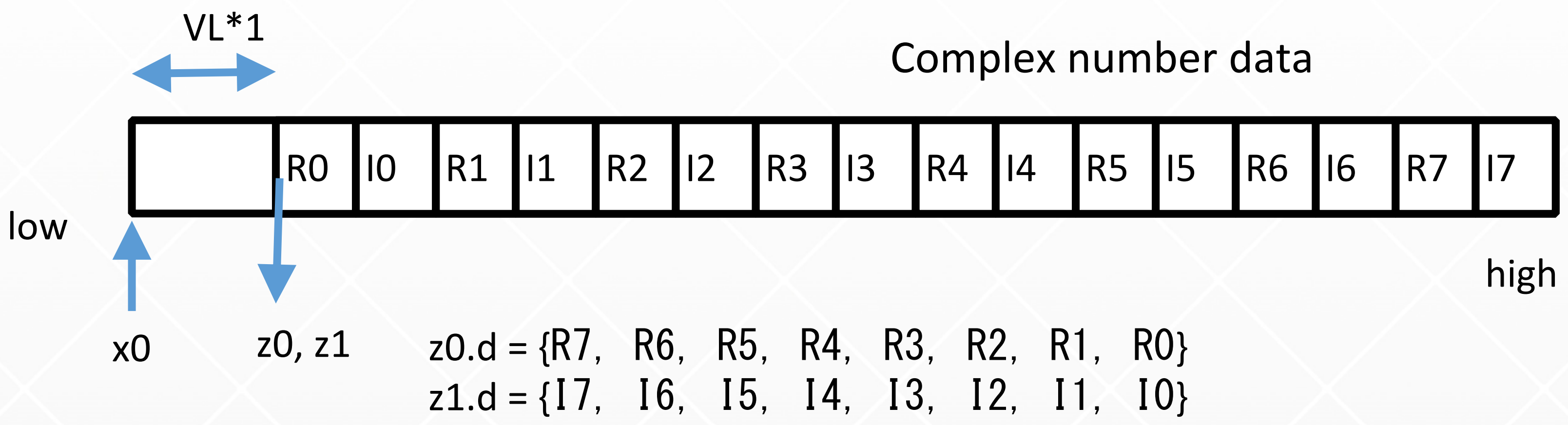
- A set of indices are specified by value in vector register (z1)
- Only active elements are read from memory.
- Otherwise, inactive elements are cleared (/z)

ld1d z0.d, p0/z, [z1.d, lsl #8]
// Vector plus immediate

The value of each element are used as base address, and plus constant offset.

Structure load instruction

ld2d {z0.d, z1.d} p0/z, [x0, #1, MUL VL]



VL*1
Complex number data
R0
I0
R1
I1
R2
I2
R3
I3
R4
I4
R5
I5
R6
I6
R7
I7
low
high
x0
z0, z1
z0.d = {R7, R6, R5, R4, R3, R2, R1, R0}
z1.d = {I7, I6, I5, I4, I3, I2, I1, I0}

- Structure load, LD2, LD3, LD4, and store, ST2, ST3, ST4, instructions transfer two, three, or four vector registers from or to contiguous structures of two, three, or four fields in memory.
- The read memory must be starting from an address that is defined by a scalar base register plus a scalar index register or an immediate index
- Instruction mnemonic : ld2 + [b,h,w,d] for each data size.

Another example

```
// ----------------------------------------
// subroutine saxpy(x,y,a,n)
// real*4 x(n),y(n),a
// do i = 1,n
// y(i) = a*x(i) + y(i)
// enddo
// ----------------------------------------
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy_:
        ldrsw x3, [x3]                  // x3=*n
        mov x4, #0                      // x4=i=0
        whilelt p0.s, x4, x3           // p0=while(i++<n)
        ld1rw z0.s, p0/z, [x2]         // p0:z0=bcast(*a)
.loop:
        ld1w z1.s, p0/z, [x0,x4,lsl 2] // p0:z1=x[i]
        ld1w z2.s, p0/z, [x1,x4,lsl 2] // p0:z2=y[i]
        fmla z2.s, p0/m, z1.s, z0.s    // p0?z2+=x[i]*a
        st1w z2.s, p0, [x1,x4,lsl 2]   // p0?y[i]=z2
        incw x4                        // i+=(VL/32)
.latch:
        whilelt p0.s, x4, x3           // p0=while(i++<n)
        b.first .loop                  // more to do?
        ret
```

Whilelt: generate predicate by comparison with index and limit

incw: increment the operand with vector size

b.first: branch if the first active element is true.

SVE instructions (1)

- Vector Set and Copy (CPY, DUP, FCPY, FDUP, FMOV, INDEX, SEL)
- Constructive Prefix (MOVPRFX)
- Integer Arithmetic (ABS, ADD, CNOT, MAD, MLA, MLS, MSB, MUL, NEG, SABD, SDIV, SDIVR, SMAX, SMIN, SMULH, SQADD, SQSUB, SUB, SUBR, SXT{B/H/W/D}, UABD, UDIV, UDIVR, UMAX, UMIN, UMULH, UQADD, UQSUB, UXT{B/H/W/D}
- Integer Dot Product (SDOT, UDOT)
- Integer Comparisons (CMP{EQ/GE/GT/HI/HS/LE/LO/LS/LT/NE})
- Loop Control (WHILE{LE/LO/LS/LT}, BRK{A/AS/B/BS/N/NS/PA/PAS/PB/PBS})
- Bitwise Logical Operations (AND, BIC, DUPM, EON, EOR, MOV, NOT, ORN, ORR)
- Bitwise Shift, Permute and Count (ASR, ASRD, ASRR, CLS, CLZ, CNT, LSL, LSLR, LSR, LSRR, RBIT)
- Floating-Point Arithmetic (FABD, FABS, FADD, FDIV, FDIVR, FMAD, FMAX, FMAXNM, FMIN, FMINNM, FMLA, FMLS, FMSB, FMUL, FMULX, FNEG, FNMAD, FNMLA, FNMLS, FNMSB, FRECP{E/S/X}, FRSQRT{E/S}, FSCALE, FSQRT, FSUB, FSUBR)
- Floating-Point Indexed Multiplies (FMLA, FMLS, FMUL)
- Floating-Point Complex Arithmetic (FCADD, FCMLA)
- Floating-Point Conversion (FCVT, FCVTZ{S/U}, FRINT{A/I/M/N/P/X/Z}, SCVTF, UCVTF)
- Floating-Point Comparison (FAC{GE/GT/LE/LT}, FCM{EQ/GE/GT/LE/LT/NE/UO}
- Floating-Point Transcendental Acceleration (FTMAD, MTMAD, FTSMUL, FTSSEL)
- Exponential (FEXPA)

SVE instructions (2)

- Predicate Set and Copy (MOV, MOVS, PFALSE, PFIRST, PTRUE, PTRUES, SEL)
- Predicate Logical Operations (AND, ANDS, BIC, BICS, EOR, EORS, NAND, NANDS, N
  OR, NORS, NOT, NOTS, ORN, ORNS, ORR, ORRS, PTEST)
- Predicate Partitioning (RDFFR, RDFFRS, SETFFR, WRFFR)
- Predicate Counts (CNT{B/D/H/P/W}, DEC{B/D/H/P/W}, INC{B/D/H/P/W}, SQDEC{B
  /D/H/P/W}, SQINC{B/D/H/P/W}, UPDEC{B/D/H/P/W}, UPINC{B/D/H/P/W})
- Permute and Shuffle (CLASTA, CLASTB, LASTA, LASTB, COMPACT, SPLICE, TBL, DUP, EXT, INSR, MOV, REV, REV{B/H/W}, SUNPK{HI/LO}, TRN{1/2}, UUNPK{HI/LO}, UZP
  {1/2}, ZIP{1/2}, PUNPK{HI/LO})
- Horizontal Reduction (ANDV, EORV, FADDA, FADDV, FMAXNMV, FMAXV, FMINNM
  V, FMINV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, UMINV)
- Serialized Operations (PNEXT, CTERM{EQ/NE})
- Vector Address Calculations (ADDPL, ADDVL, ADR, RDVL)
- Vector Load/Store/Prefetch (LD1{B/SB/H/SH/W/SW/D}, LDFF1{B/SB/H/SH/W/SW/
  D}, LDNF1{B/SB/H/SH/W/SW/D}, LDNT1{B/H/W/D}, PRF{B/H/W/D}, ST1{B/H/W/
  D}, STNT1{B/H/W/D}, LD2{B/H/W/D}, LD3{B/H/W/D}, LD4{B/H/W/D}, ST2{B/H/W/
  D}, ST3{B/H/W/D}, ST4{B/H/W/D}, LD1R{B/H/W/D}, LD1RQ{B/H/W/D}, LDR, STR,

Information about ARM v8 and SVE

- **Specification of ARMv8 and ARM SVE is available at:**
  - https://developer.arm.com/
  - ARM SVE architecture reference manual (ARM® Architecture Reference Manual Supplement, The Scalable Vector Extension (SVE), for ARMv8-A)
    - https://developer.arm.com/products/architecture/a-profile/docs
  - Other reference: a presentation slide at Hot Chips 2016
    - https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-01-20-49/ARMv8_2D00_A-SVE-technology-Hot-Chips-v12.pdf

- **Compiler for ARM SVE**
  - https://github.com/ARM-software/LLVM-SVE