RISC-V SIMD ARCHITECTURE

Dr. Girish H

Professor

Department of ECE

Cambridge Institute of Technology

&

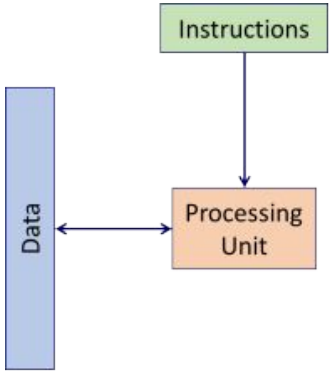
Kavinesh

Research Staff

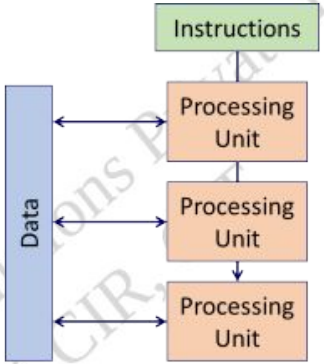
CCCIR

Cambridge Institute of Technology

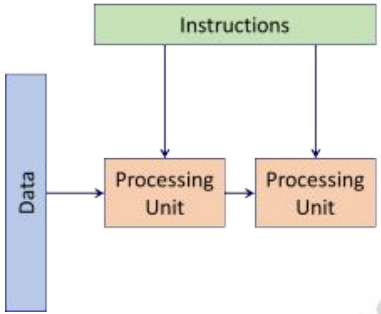
Flynn Taxonomy Recap



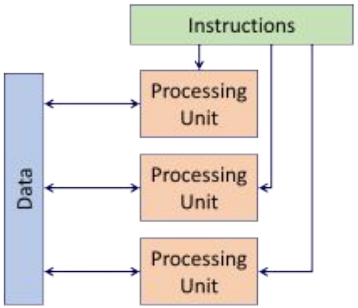
Single-Instruction
Single-Data
(Single-Core Processors)



Single-Instruction
Multi-Data
(GPUs, Intel SIMD Extensions)



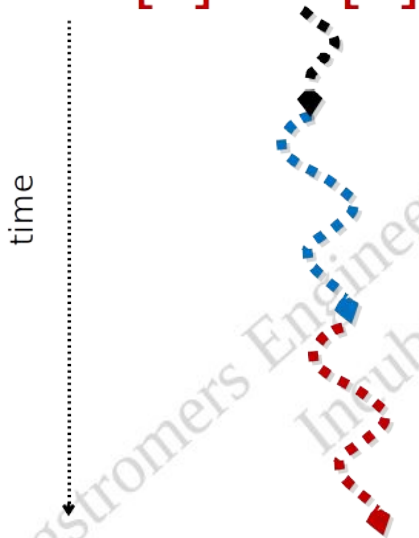
Multi-Instruction
Single-Data
(Systolic Arrays,...)



Multi-Instruction
Multi-Data
(Parallel Computers)

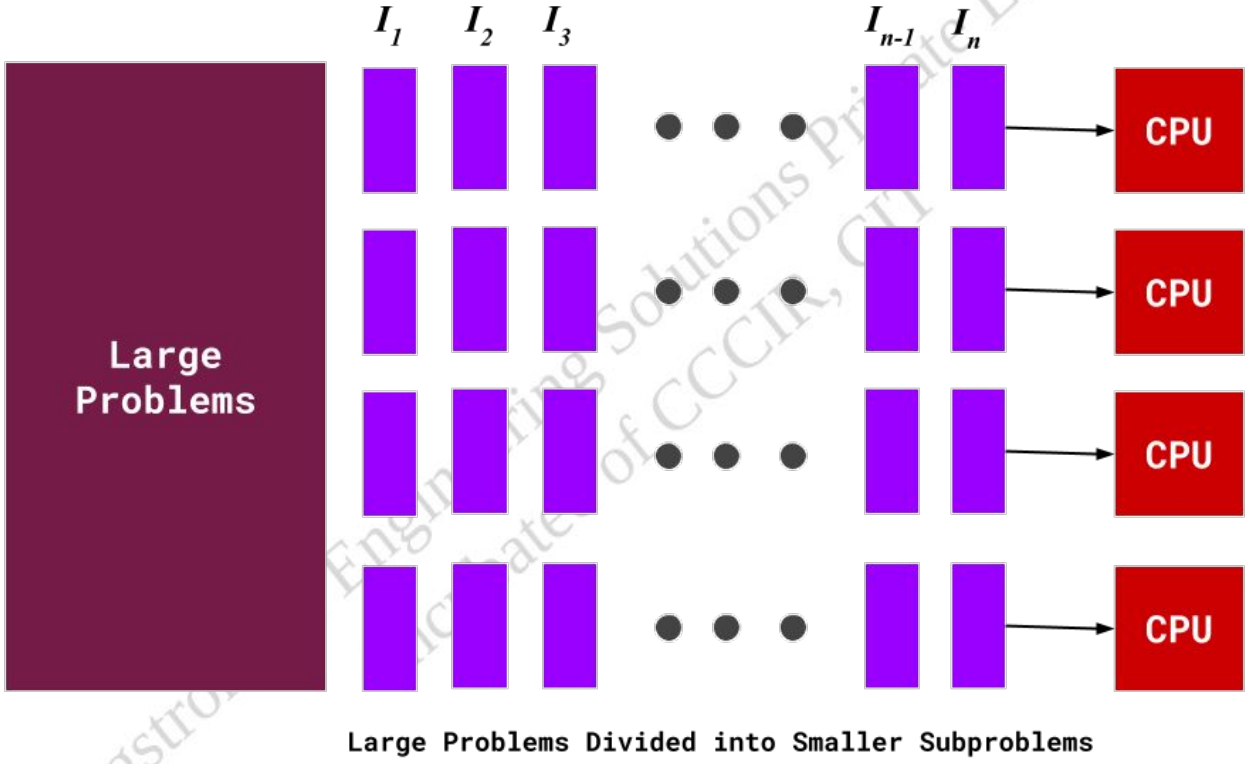
Sequential Execution Model / SISD

```
int a[N]; // N is large
for (i =0; i < N; i++)
  a[i] = a[i] * fade;
```



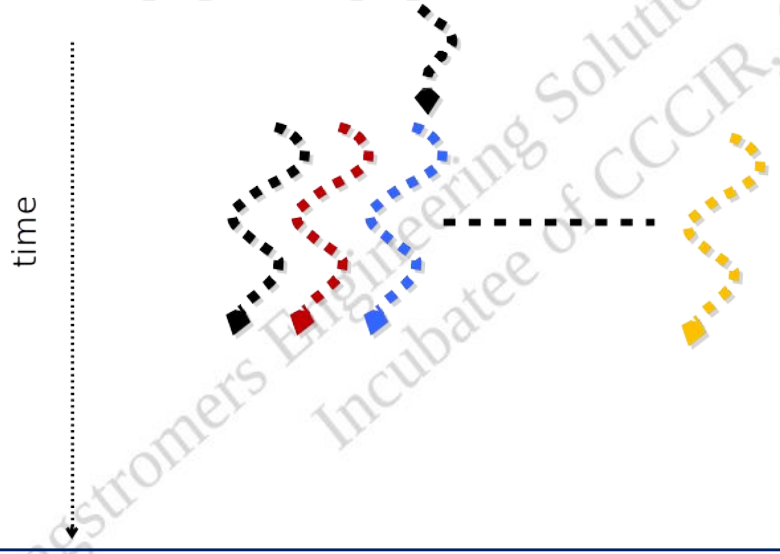
Flow of control / Thread
One instruction at the time
Optimizations possible at the machine level

SISD is inefficient to solve for large data



Data Parallel Execution Model / SIMD

```
int a[N]; // N is large
for all elements do in parallel
  a[i] = a[i] * fade;
```



Introduction to SIMD Architecture

SIMD architectures can exploit significant data-level parallelism for:

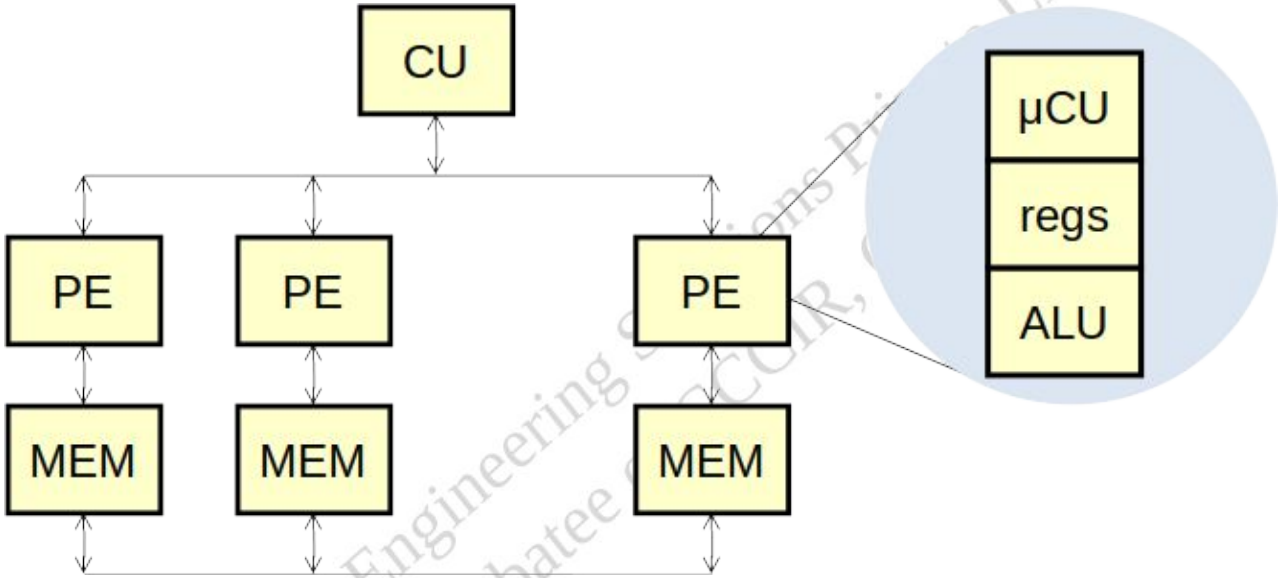
- matrix-oriented scientific computing
- media-oriented image and sound processors

SIMD is more energy efficient than MIMD

- Only needs to fetch one instruction per data operation
- Makes SIMD attractive for personal mobile devices

SIMD allows programmer to continue to think sequentially

SIMD Architecture



- Replicate Datapath, not the control and all PEs work in tandem
- Control Unit orchestrates operations

SIMD Architecture

Basic idea:

- Read sets of data elements into “vector registers”
- Operate on those registers
- Disperse the results back into memory

Registers are controlled by compiler

- Used to hide memory latency
- Leverage memory bandwidth

SIMD Variations/Configuration

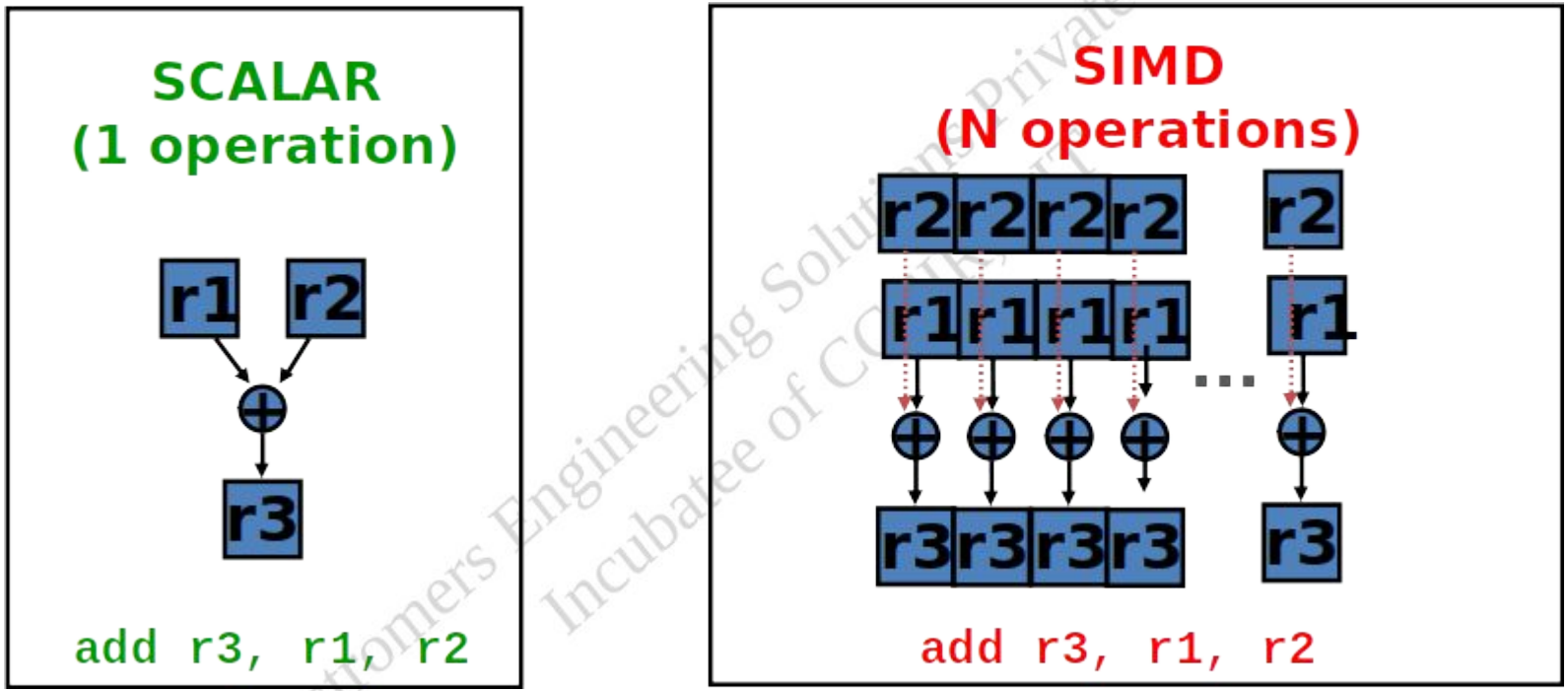
Vector architectures

- SIMD extensions
- MMX: Multimedia Extensions (1996)
- SSE: Streaming SIMD Extensions
- AVX: Advanced Vector Extension (2010)
- Graphics Processor Units (GPUs)

Multimedia / Scientific Applications

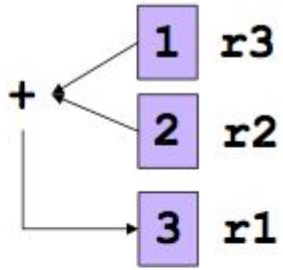
- Graphics : 3D games, movies
- Image recognition
- Video encoding/decoding : JPEG, MPEG4
- Sound Encoding/decoding: IP phone, MP3
- Speech recognition Digital signal processing: Cell phones
- Scientific applications
- Double precision Matrix-Matrix multiplication (DGEMM)
- Y[] = a*X[] + Y[] (SAXPY)

SIMD processing

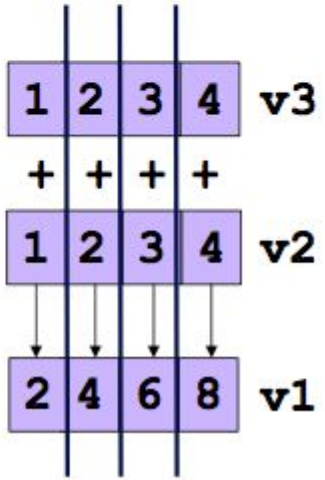


SISD vs SIMD

```
Scalar: add r1,r2,r3
```



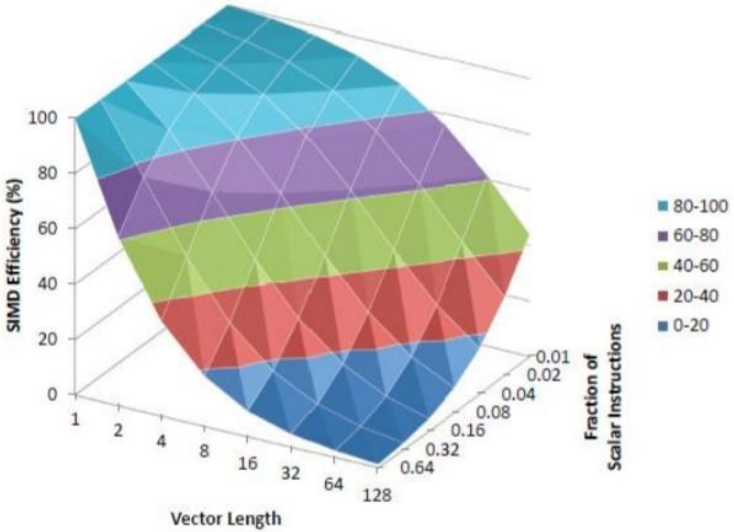

```
SIMD: vadd<sws> v1,v2,v3
```

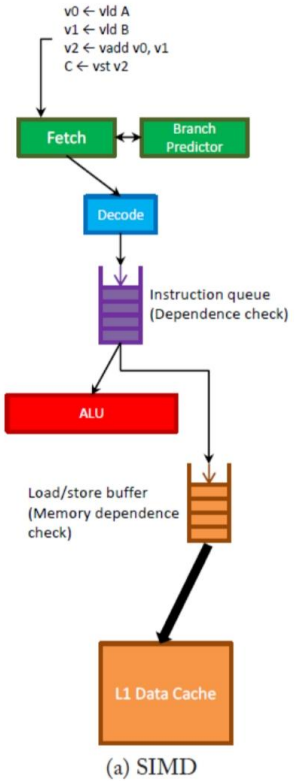


SIMD Efficiency

From [Hughes, SIMD Synthesis Lecture]

$$SIMD\ efficiency = \frac{\frac{instructions_{scalar}}{instructions_{SIMD}}}{vector\ \ length}$$



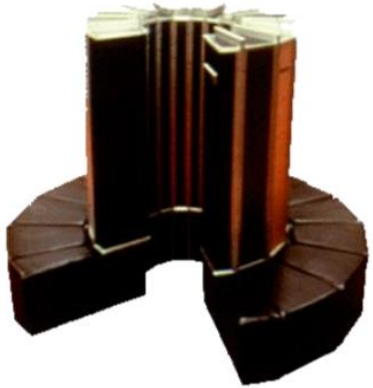


(a) SIMD

- Amdahl's Law…

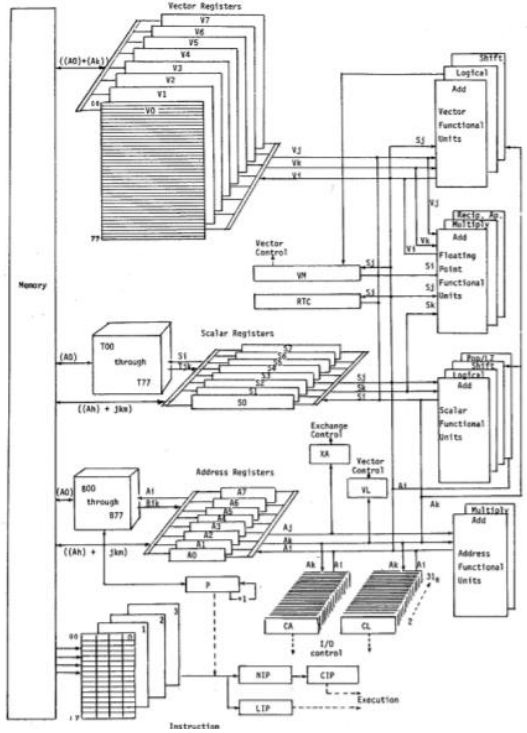
Cray-1 Architecture

- Circa 1976
- 80 MHz clock
  - When high performance mainframes were 20 MHz
- Scalar instruction set
  - 16/32 bit instruction sizes
    - Otherwise conventional RISC
  - 8 S register (64-bits)
  - 8 A registers (24-bits)
- In-order pipeline
  - Issue in order
  - Can complete out of order (no precise traps)



Cray-1 Vector ISA

- 8 vector registers
  - 64 elements
  - 64 bits per element (word length)
  - Vector length (VL) register
- RISC format
  - Vi ← Vj OP Vk
  - Vi ← mem(Aj, disp)
- Conditionals via vector mask (VM) register
  - VM ← Vi pred Vj
  - Vi ← V2 conditional on VM



Case study: Sony Playstation 2000

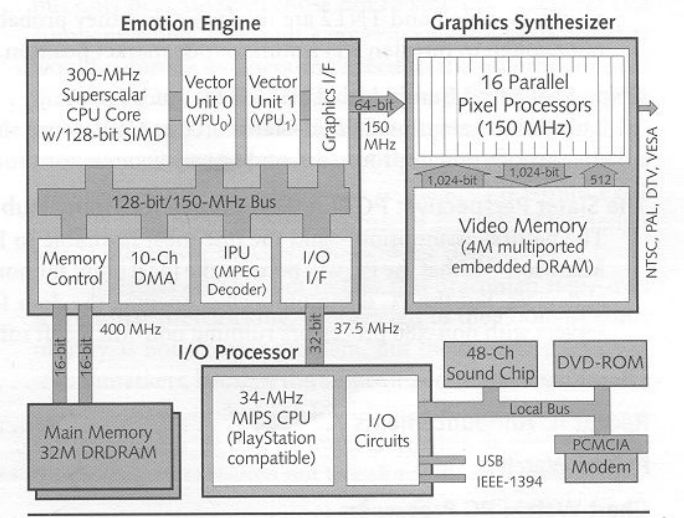


Figure 1. PlayStation 2000 employs an unprecedented level of parallelism to achieve workstation-class 3D performance.

Figure 2. PlayStation 2000 screenshot. (Source: Namco)

Emotion Engine: 6.2 GFLOPS, 75 million polygons per second.Graphics Synthesizer: 2.4 Billion pixels per second

Claim: Toy Story realism brought to games!

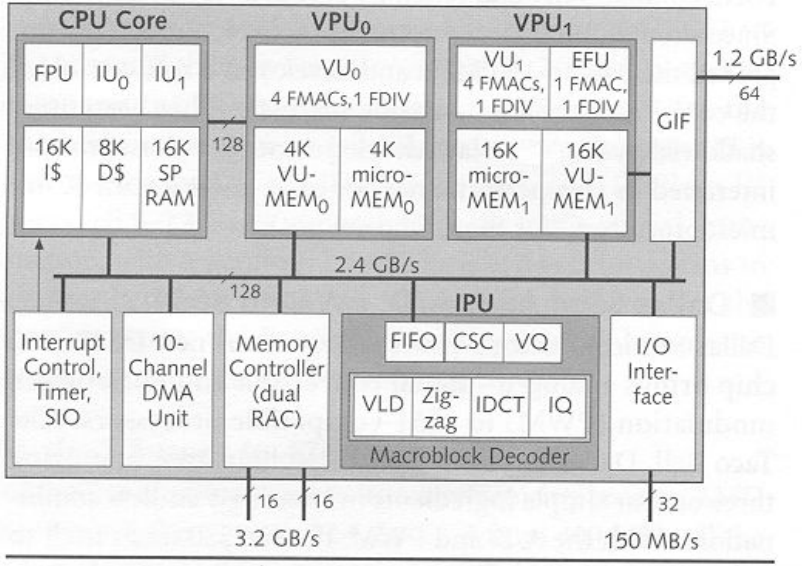


Figure 3. The PSX2's Emotion Engine provides ten floating-point multiplier-accumulators, four floating-point dividers, and an MPEG-2 decoder to deliver killer multimedia performance.

Emotion Engine:

Superscalar MIPS core

Vector Coprocessor Pipelines

RAMBUS DRAM interface

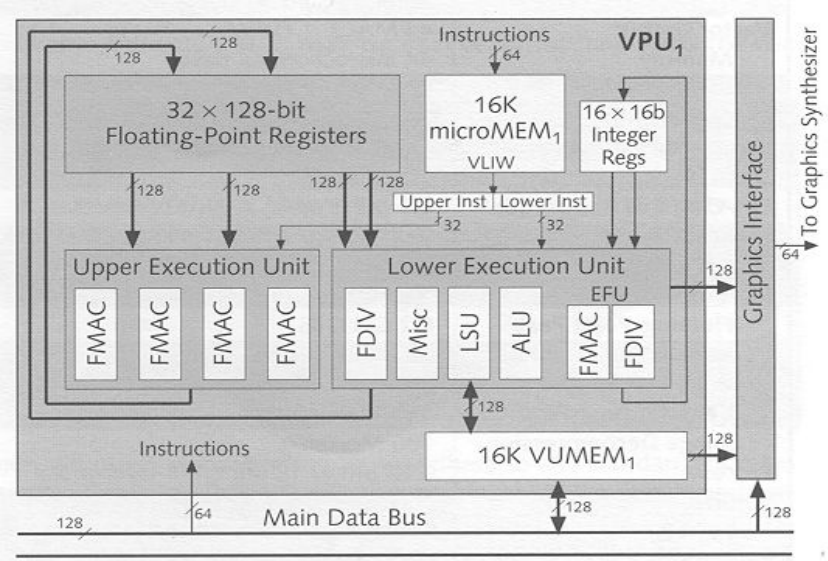


Figure 5. Each vector unit has enough parallelism to complete a vertex operation (19 mul-adds + 1 divide) every seven cycles.

Sample Vector Unit

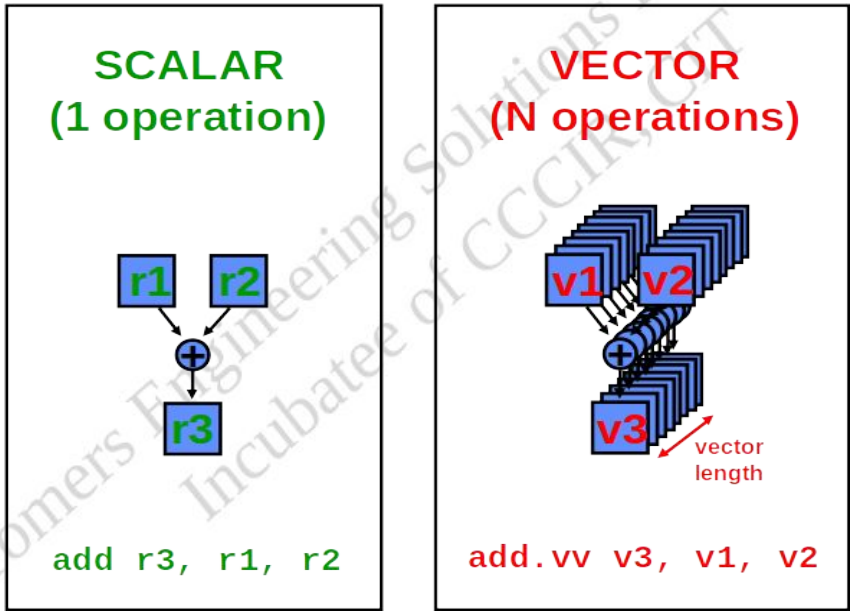
2-wide VLIW

Includes Microcode Memory

High-level instructions like matrix-multiply

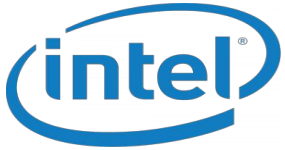
Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



Why a Vector Extension?

Vector ISA Goodness	RISC-V Vector ISA	Domain
• Reduced instruction bandwidth • Reduced memory bandwidth • Lower energy • Exposes DLP • Masked executio	• Scalar, Vector & Matrix • Typed registers • Reconfigurable • Mixed-type instructions • Common Vector/SIMD programming model • Fixed-point support • Easily Extensible	• Machine Learning • Graphics • DSP • Cryptography • Structural analysis • Weather prediction • Drug design



x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register-memory
Encoding	Variable (1 to 15 bytes)
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)
x86 Instruction Set

ARM architectures

Designer	ARM Holdings
Bits	32-bit, 64-bit
Introduced	1985; 31 years ago
Design	RISC
Type	Register-Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
Endianness	Bi (little as default)

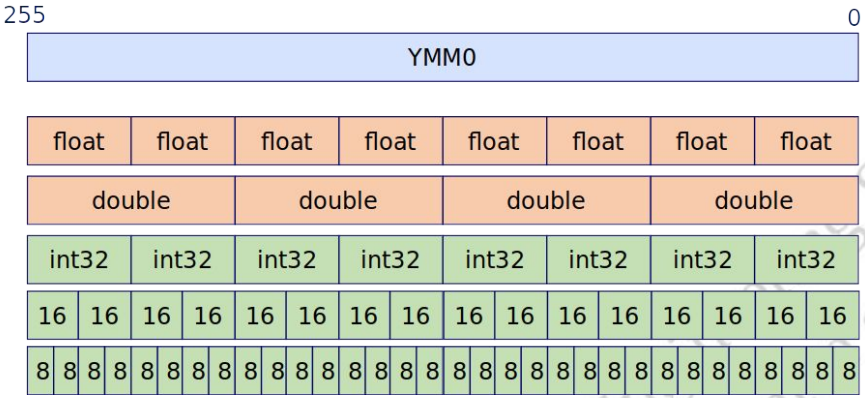
Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

RISC-V

Designer	University of California, Berkeley
Bits	32, 64, 128
Introduced	2010
Version	2.2
Design	RISC
Type	Load-store
Encoding	Variable
Branching	Compare-and-branch
Endianness	Little

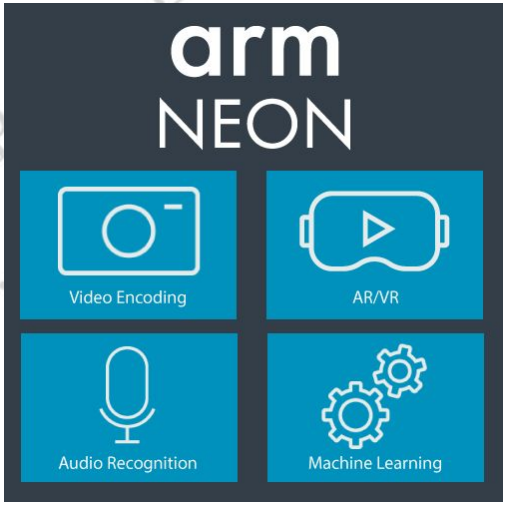
Versatile and open-source designed for cloud computing, small embedded sys.

Intel SIMD Registers (AVX-512)



- YMM0 – YMM15
  256-bit registers AVX, AVX2
- Operation on 32 8-bit values in one instruction!

ARM NEON Intrinsic



- Arm Neon Intrinsics is for the Advanced SIMD architecture extension

	RVV	ARM SVE	x86 AVX512
Spec supports multiple vector sizes	Yes	Yes	No
VL register	Yes	No	No
Masks	Yes (1)	Yes (16)	Yes (8)
Narrowing/Widening	Yes, LMUL	Yes, 2 instructions	Yes, 2 instructions
Mixed Datatype Vectorization	Uses LMUL	Pack/Unpack	Pack/Unpack
Hardware Unrolling (LMUL)	Yes	No	No
Polymorphic Encoding	Yes, vtype	No	No
Strided Memory Access	Yes	No	No
Gather/Scatter	Yes	Yes	Yes
Structured/Segmented Loads	Yes	Yes	No
Forward Progress on gather/scatter	Vstart	Repeat full instruction	Mask state
Fixed Point Support	Yes	Partial	No
Complex Support	No	Yes	Partial
Reductions	Yes	Yes	No
Register Gather	Yes	Yes	No
Tail Element control	Merge, Agnostic	Merge, Zeroing	Merge, Zeroing[9]
Destructive destination	No	Yes	No

RISC-V Vectors vs Traditional SIMD

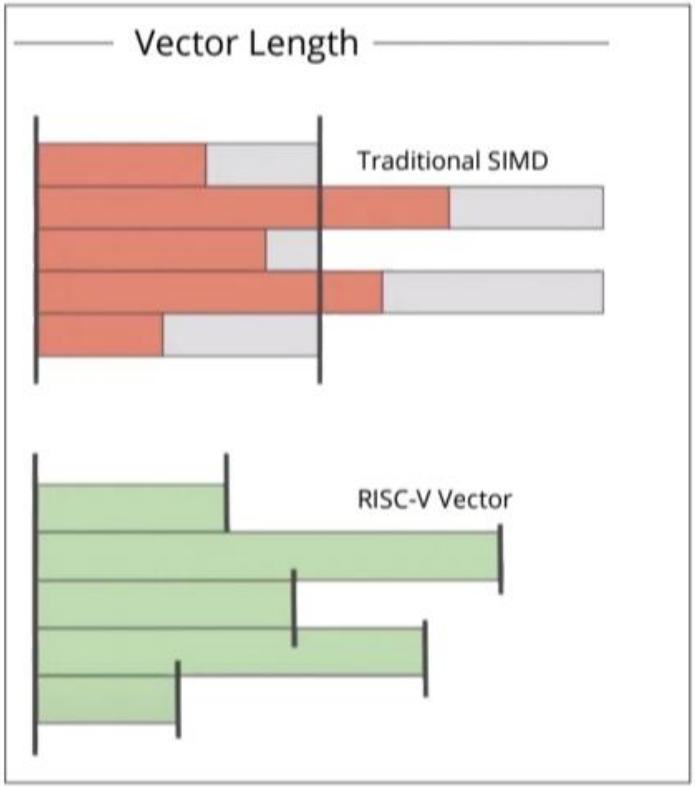
Variable Vector Register Length

- ○ Vector width configurable at runtime
- ○ Up to maximum hardware length (VLEN)
- ○ Vector ISA is agnostic to VLEN

● Unified Code Base

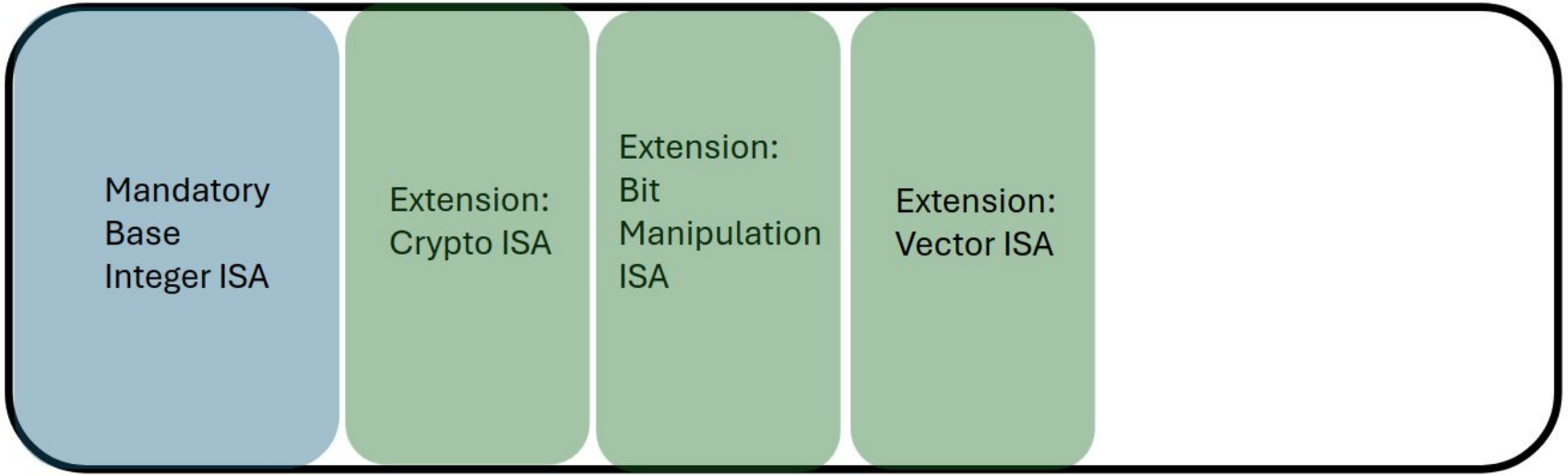
- ○ ARM has two different ISAs (Helium & Neon)
- ○ IA-32 vector instructions were an order of magnitude larger when supporting SIMD vector lengths

● No requirement for dedicated vector data memory



RISC-V Vector Extension RVV

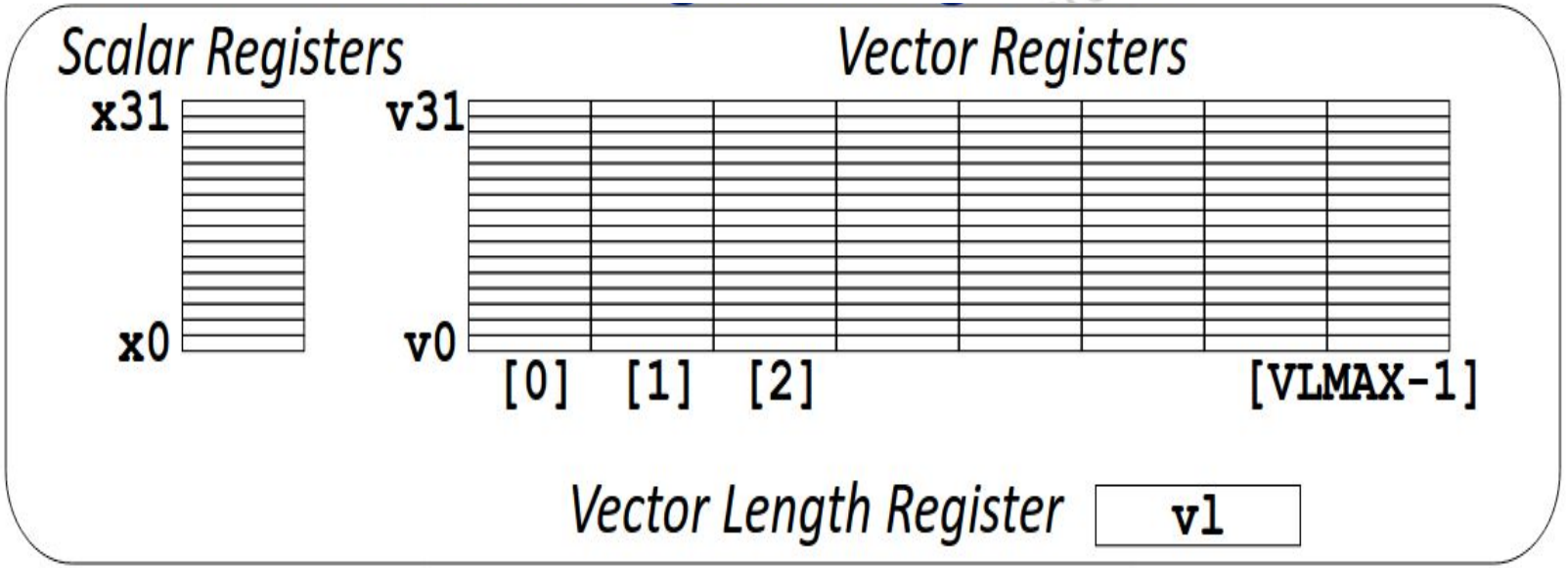
The RISC-V V vector extension (RVV) resembles a traditional vector ISA, with some key distinctions.



What are specified in a Vector Instruction Set Architecture?

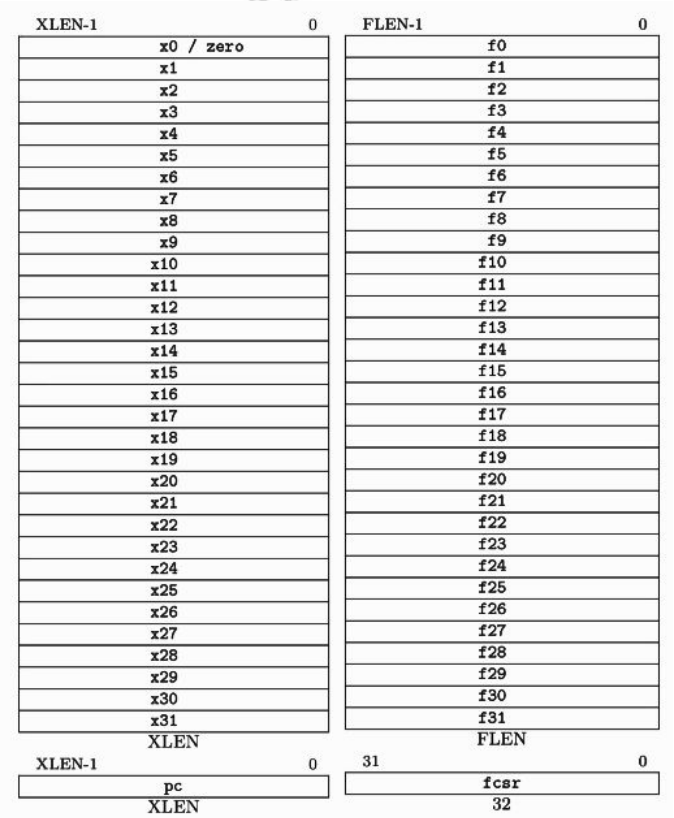
- ISA in general: Operations, Data types, Format, Accessible Storage, Addressing Modes, Exceptional Conditions
- Vectors :Operations, Data types (Float, int, V op V, S op V), format
- Source and Destination Operands : Memory?, register?
- Length
- Successor (consecutive, stride, indexed, gather/scatter, …)
- Conditional operations
- Exceptions

Vector Programming model



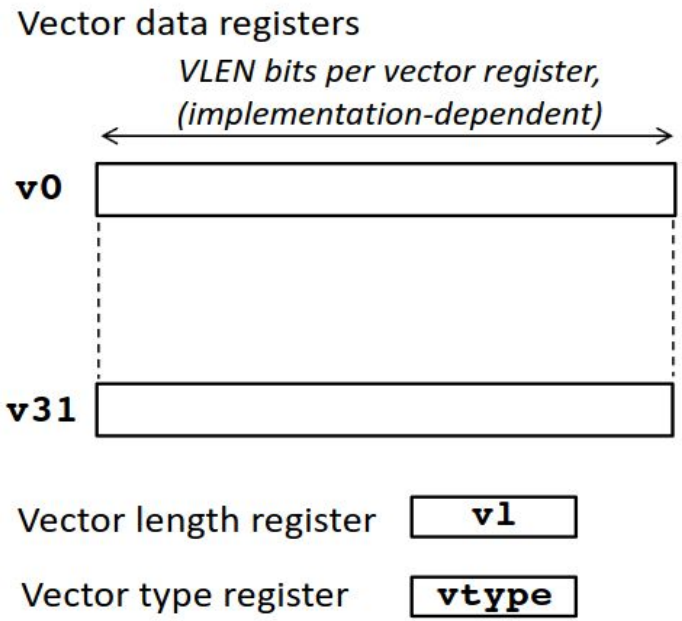
RISC-V Scalar State

- Program counter (pc)
- 32x32/64-bit integer registers (x0-x31)
- Floating-point (FP), adds 32 registers (f0-f3
- ISA string options:
- RV32I (XLEN=32)
- RV32IF (XLEN=32, FLEN=32)
- RV32ID (XLEN=32, FLEN=64)
- RV64I (XLEN=64)
- RV64IF (XLEN=64, FLEN=32)
- RV64ID (XLEN=64, FLEN=64)

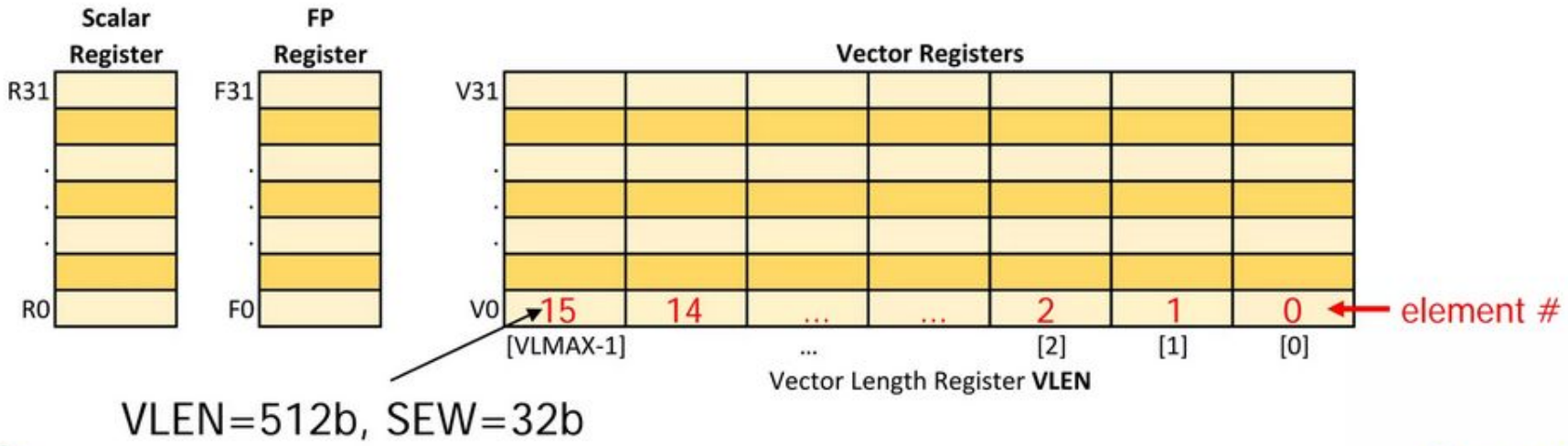


Vector Extension

- 32 vector data registers, v0-v31 , each VLEN bits long



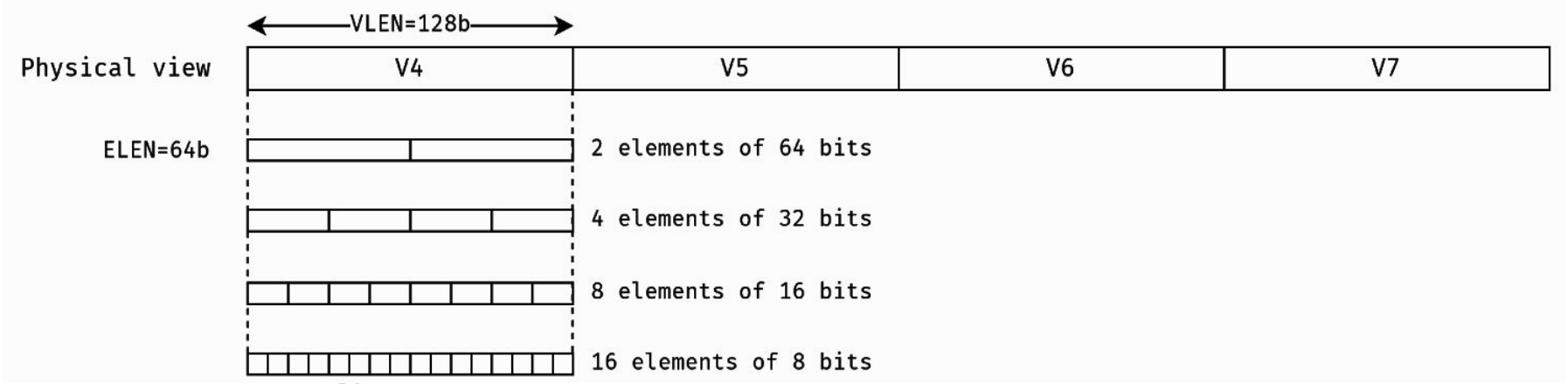
Vector Register 32 bit elements



RVV Parameters

RVV are affected by global parameters: VLEN, LMUL, SEW.

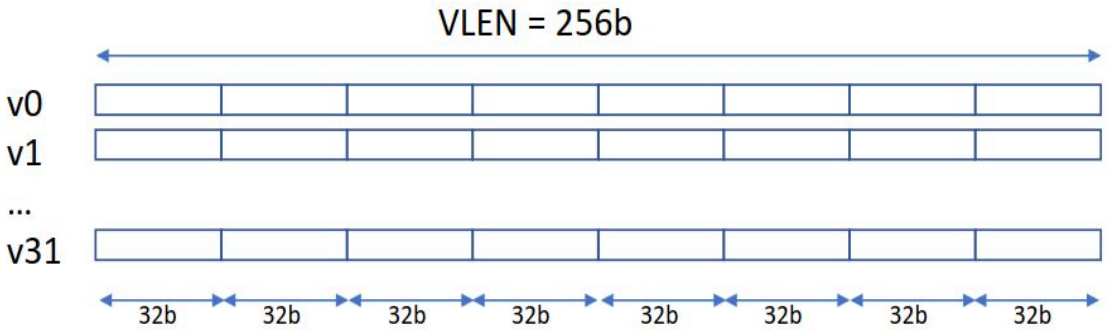
- RVV defines 32 vector registers of size VLEN bits named : v0 to v31.
- Each register is VLEN-bits wide, VLEN is chosen by implementation, must be power of 2.



Selected Element Width (SEW):

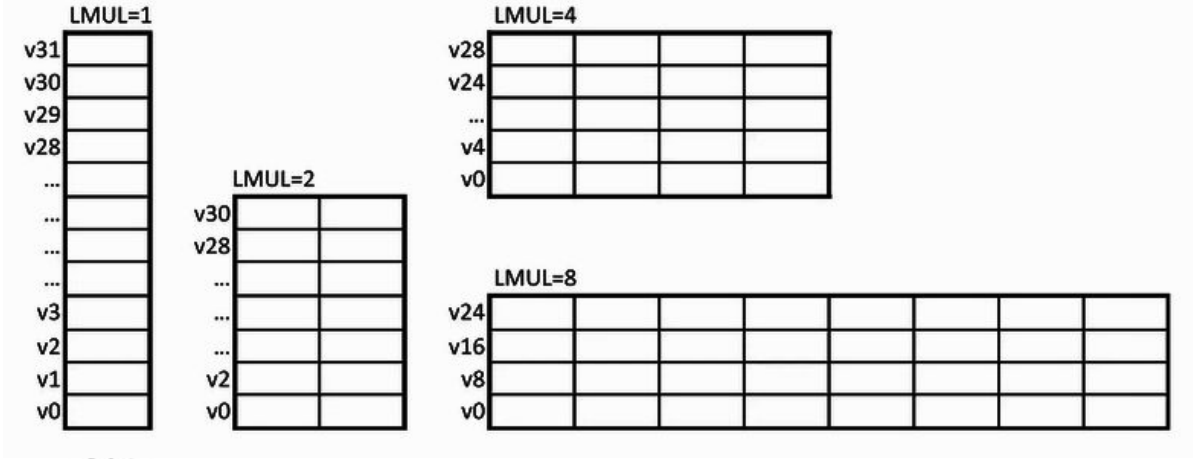
- SEW determines number of elements per vector.
- It represents the width of the currently selected element within the vector register.
- It acts as a divider, breaking down the vector register into multiple elements

Example: VLEN=256b, vsew='010, SEW=32b, elements = VLEN/SEW = 8

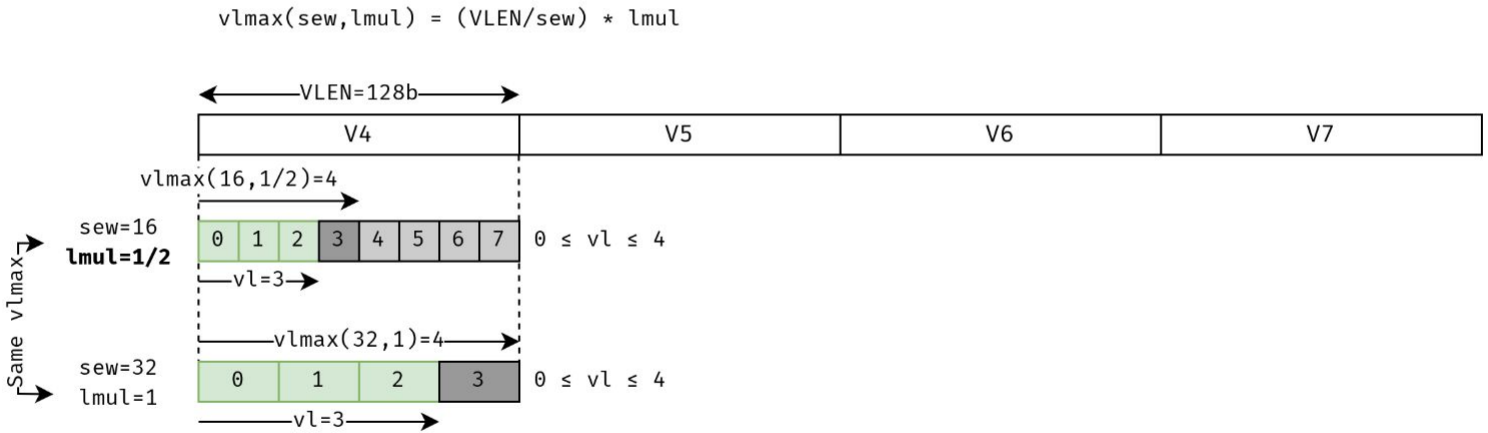


Vector Register Group Multiplier (LMUL)

- LMUL is a multiplier that allows you to pack multiple vector registers into a group.
- With LMUL=2, vector registers vn and vn + 1 are operated on as one vector with twice the maximum vector length.



- The LMUL is an integer power of 2 (from 1 to 8) that controls how many consecutive vector registers are grouped together to form longer vectors.
- RVV allows the two scenarios via the length multiplier
- When lmul = 1 we can operate up to all the elements of a vector register
- When lmul < 1 we can operate up to a fraction of all the elements of a vec



Vector Integer Instructions

operation	instructions
add	vadd, vaddi, vaddw, vaddiw
subtract	vsub, vsubw
multiply	vmul, vmulh, vmulhsu, vmulhu
widening multiply	vmulwdn
divide	vdiv, vdivu, vrem, vremu
shift	vsll, vslli, vsra, vsrai, vsrl, vsrli
logical	vand, vandi, vor, vori, vxor, vxori
compare	vseq, vslt, vsltu
fixed point	vclipb, vclipbu, vcliph, vcliphu, vclipw, vclipwu

Vector Floating Point Instructions

operation	instructions
add	vfadd.h, vfadd.s, vfadd.d
subtract	vfsub.h, vfsub.s, vfsub.d
multiply	vfmul.h, vfmul.s, vfmul.d
divide	vfdiv.h, vfdiv.s, vfdiv.d
sign	vfsgn{j,jn,jx}.h, vfsgn{j,jn,jx}.s, vfsgn{j,jn,jx}.d
max	vfmax.h, vfmax.s, vfmax.d
min	vfmin.h, vfmin.s, vfmin.d
compare	vfeq.h, vfeq.s, vfeq.d, vltq.h, vflt.s, vflt.d, vfle.h, vfle.s, vfle.d
sqrt	vfsqrt.h, vfsqrt.s, vfsqrt.d

Vector Data Movement

operation	instructions	action
insert gpr into vector	vins vd, rs1, rs2	vd[rs2] = rs1
insert fp into vector	vins vd, fs1, rs2	vd[rs2] = fs1
extract velem to gpr	vext rd, vs1, rs2	rd = vs1[rs2]
extract velem to fp	vext fd, vs1, rs2	fd = vs1[rs2]
vector-vector merge	vmerge vd, vs1, vs2, vm	mask picks src
vector-gpr merge	vmergex vd, rs1, vs2, vm	mask picks src
vector-fp merge	vmergef vd, fs1, vs2, vm	mask picks src
vector register gather	vrgather vd, vs1, vs2, vm	vd[i] = vs1[vs2[i]]
Gpr splat/bcast	vsplatx vd, rs1	Vd[0..MAXVL] = rs1
fpr splat/bcast	vsplatf vd, fs1	Vd[0..MAXVL] = fs1
vector slide down	vslidedwn vd, vs1, rs2, vm	vd[i] = vs1[rs2+i]
vector slide up	vslideup vd, vs1, rs2, vm	vd[rs2+i] = vs1[i]

Vector Floating Point Multiply Add

operation	instructions
add	vfmadd.h, vfmadd.s, vfmadd.d
sub	vfmsub.h, vfmsub.s, vfmsub.d
widening add	vfmaddwdn.h, vfmaddwdn.s, vfmaddwdn.d
widening sub	vfmsubwdn.h, vfmsubwdn.s, vfmsubwdn.d

Vector Convert

From Integer to Float	
To Half	vfcvt.h.i, vfcvt.h.u
To Single	vfcvt.s.i, vfcvt.s.u
To Double	vfcvt.d.i, vfcvt.d.u
From Float to Vemaxw Integer	
To Signed	vfcvt.i.h, vfcvt.i.s, vfcvt.i.d
To Unsigned	vfcvt.u.h, vfcvt.u.s, vfcvt.u.d
From Float to Float	
To Half	vfcvt.h.s, vfcvt.h.d
To Single	vfcvt.s.h, vfcvt.s.d
To Double	vfcvt.d.h, vfcvt.d.s

Introduction

- The PULP Ara is a 64-bit Vector Unit which is compatible with the RISC-V Vector extension version 1.0. working as a coprocessor to CORE-V's CVA6.

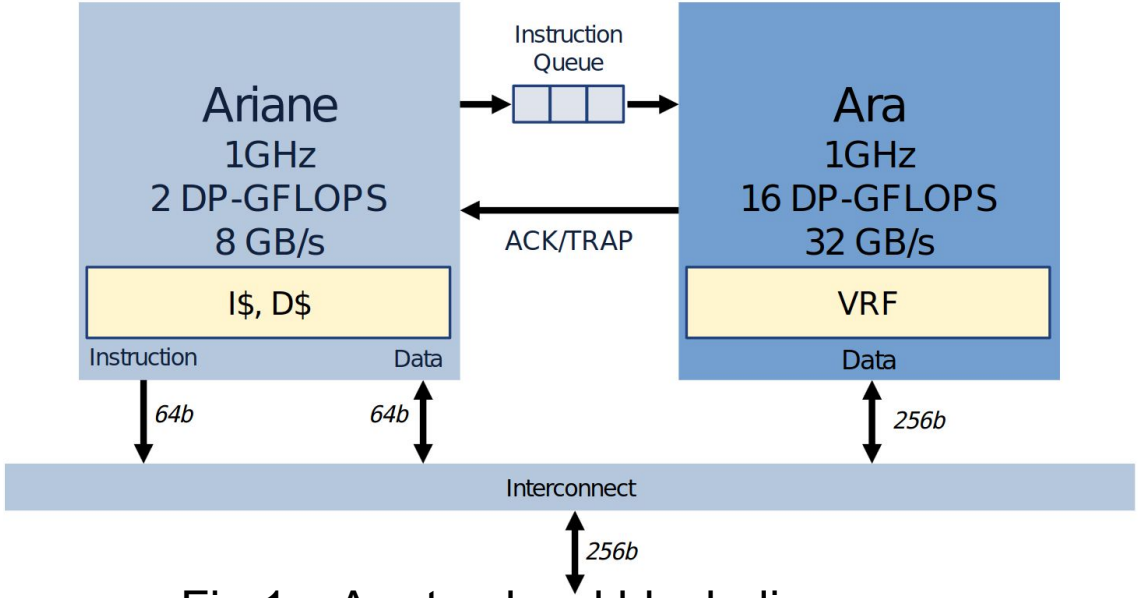


Fig 1a. Ara top level block diagram

Significance of vector architecture

- In an instruction-based programmable architectures, the key challenge is how to mitigate the Von Neumann Bottleneck (VNB).
- Each core tends to execute the same instruction many times, resulting in waste in terms of b

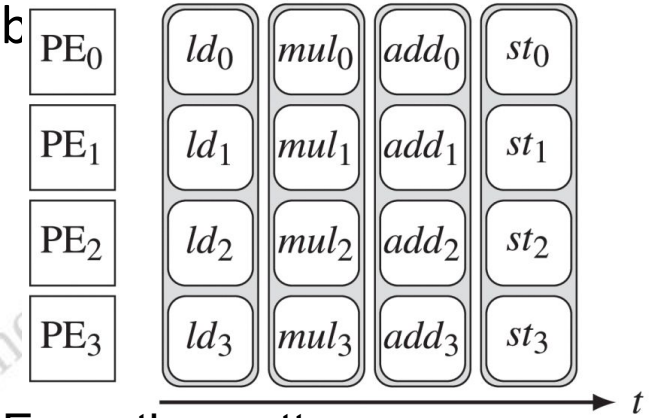


Fig 1b. Execution pattern on a array processor

- The need for extreme energy efficiency in data-parallel execution shifted the interest in vector architectures.
- Such systems tackle the VNB very effectively, providing better energy efficiency than a general-purpose processor for applications.

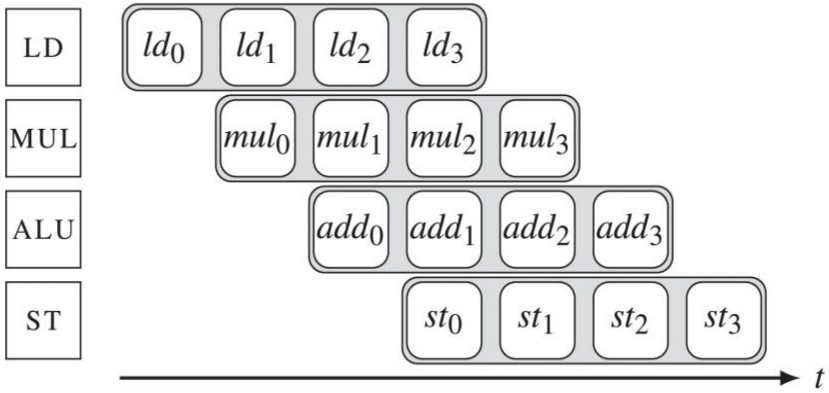


Fig 1c. Execution pattern on a vector processor

- The vector machines tackle the VNB through vector instructions, which encode a series of micro-operations within a single instruction.
- This renewed interest in vector processing lead to introduction of vector instruction extensions in all popular ISA, such as Arm's with its SVE, and RISC-V with the V extension.

Fig 1d. Execution cycle

Top Level Block Diagram

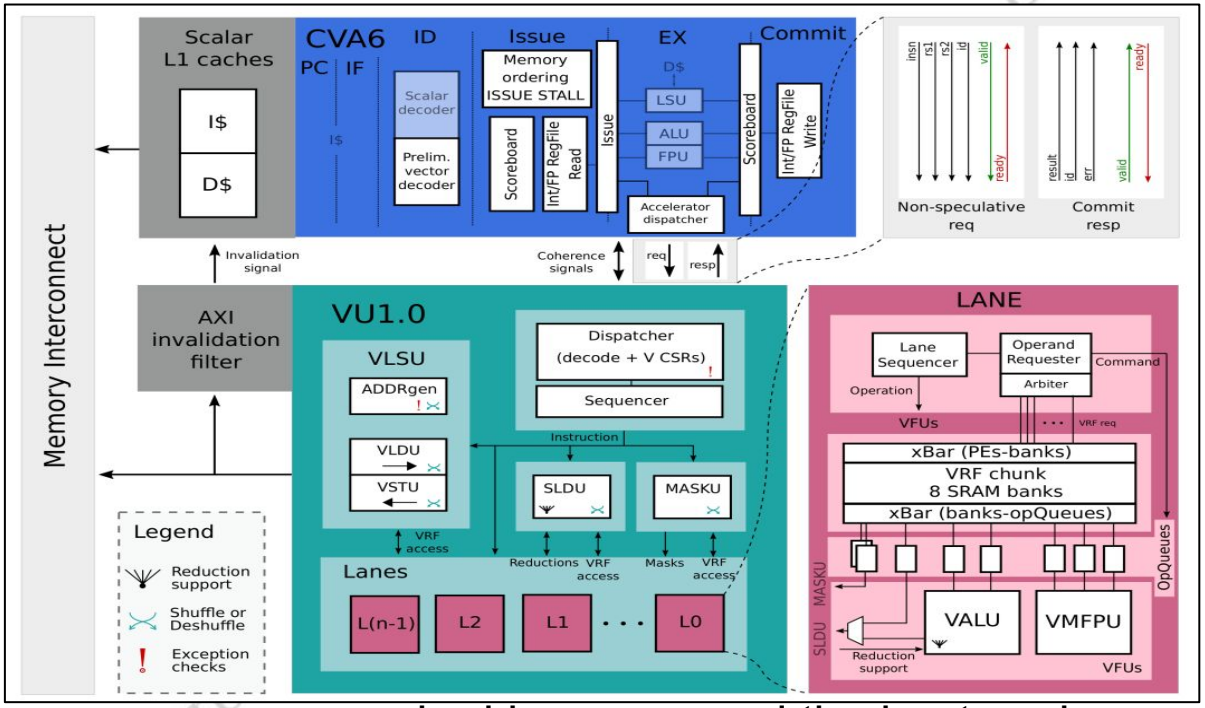


Fig 1e.Vector coprocessor marked in green, and the host scalar core CVA6 in blue

Microarchitecture

- Ara has a scalable microarchitecture and is composed of a set of identical lanes.
- Each lane contain part of the processor’s vector register file and functional units.
- The main data path elements of Ara are ALU, MUL and FPU
- The floating point unit supports FP64, FP32, FP16, bfloat16 with Independent pipelines for each data type where each has a different latency.

Vector Unit 1.0

- The VU1.0 is a flexible architecture with parametric VLEN which helps to obtain high performance and efficiency on a vast range of vector lengths.
- The vector unit 1.0 is mainly composed of:
- Vector Load Store Unit
- Lane
- Dispatcher and Sequencer
- Slide Unit
- Mask Unit

Vector Register File

Each lane of the vector processor has 3 functional units:

1. Vector Floating-point Unit (VFPU)
2. Vector Multiplier (VMUL)
3. Vector Integer ALU (VALU)

- A Vector Load & Store unit (VLSU) for Lector Load and Store instructions
- Slide unit (SLDU) for Vector Slide instructions are present as separate units outside the lanes.

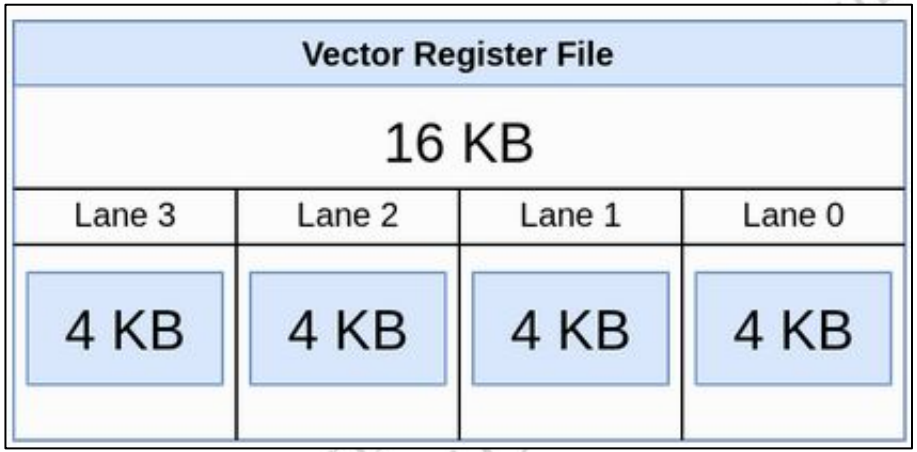


Fig 1f. Vector register file

- There are eight banks per lane resulting in eight single ported memory banks per lane.
- Size of Vector register (VLEN) = 4096 bit
- Size of Vector Register File = 4096 * 32 = 217 bits = 16 KB

- Number of Lanes in Ara can vary from 1 to 16 (implementation dependent).
- In a 4 lane system the VRF will be divided into 4 lanes with 4KiB of memory per lane.

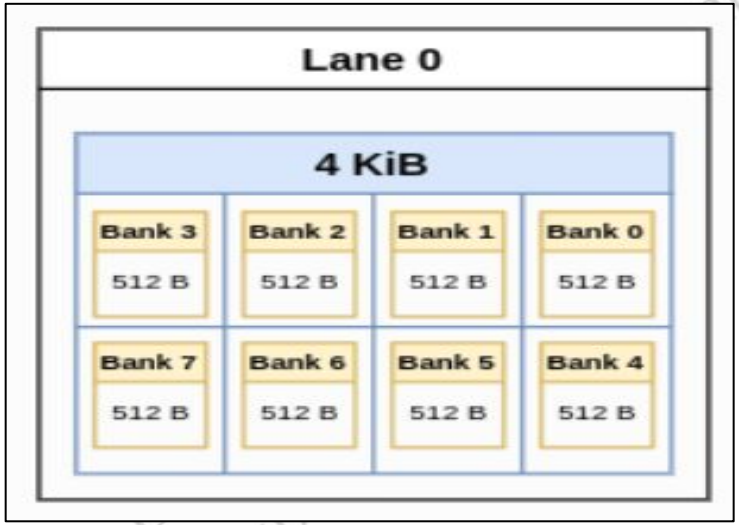
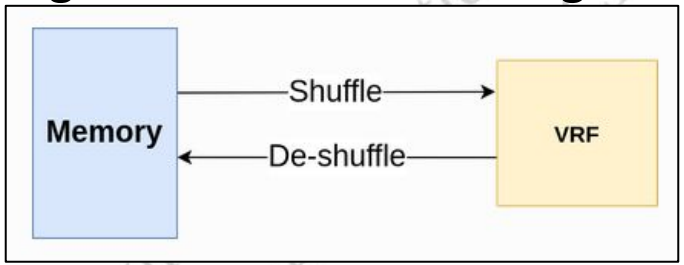


Fig 1g. Inside a lane

- Each lane will as such have 8 banks of 512B (4096B / 8 = 512B) memory.

Shuffle/ De-shuffle:

- The shuffle/de-shuffle logic sits between the memory subsystem and the VRF.
- When data is moved from memory to the VRF it gets shuffled from the Natural Packing arrangement to the Lane Organization arrangement.

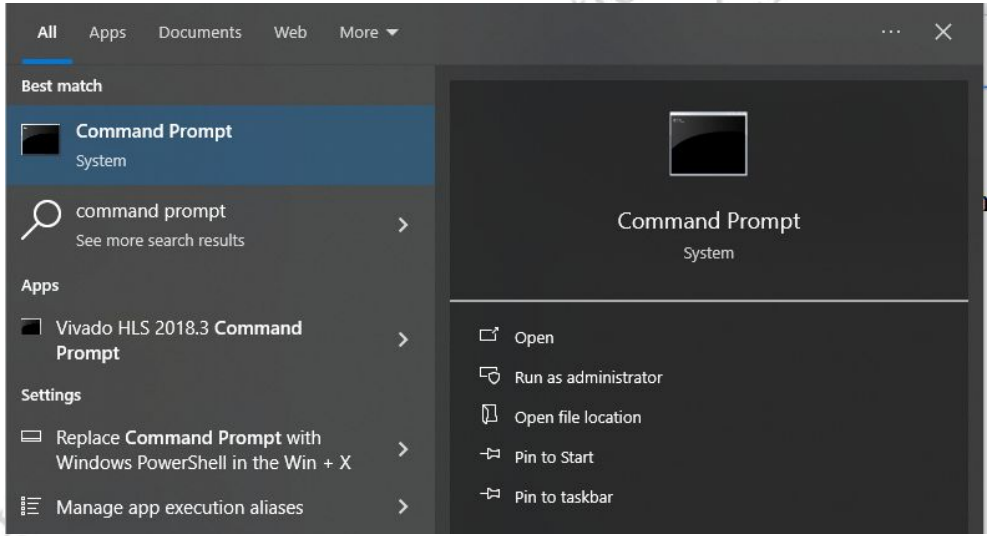


- Similarly When data is moved from VRF to memory (via a store instruction) it gets de-shuffled from the Lane Organization to the Natural Packing arrangement.

RISC-V Programming

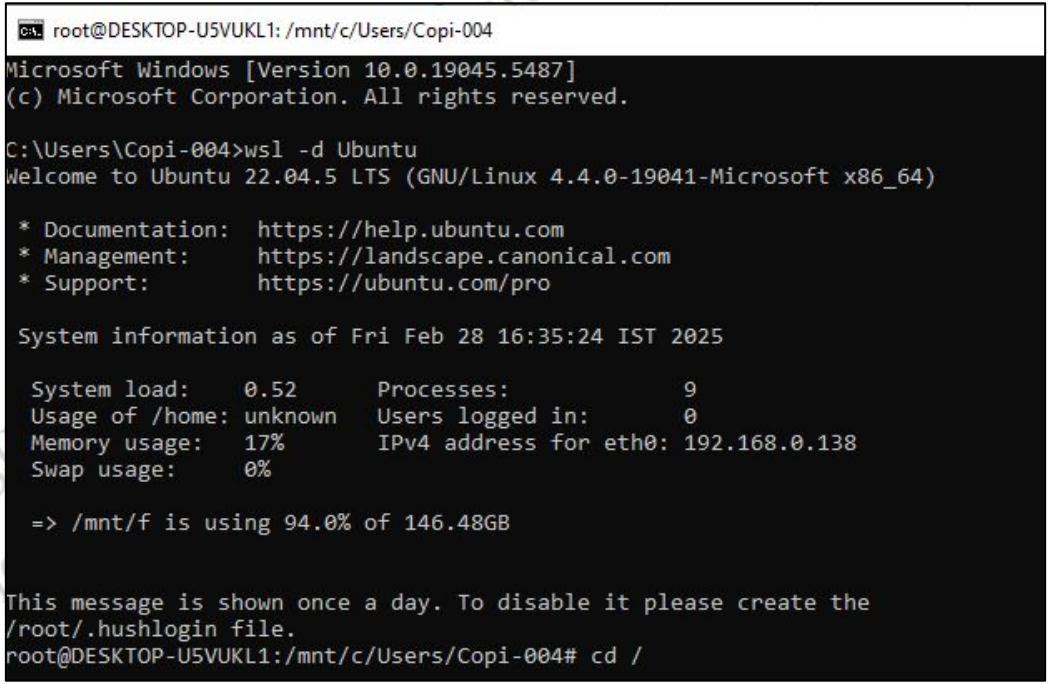
Launch the WSL software

- Search for command prompt in windows and type the following command



Initial Configuration

- Launch command prompt using wsl -d Ubuntu
- To change from directory cd / to the main working directory



Code Compilation

- To create a new folder in the directory type the command mkdir followed by the folder name, in which the saved program will be store. Example : folder_name = Program_name = vadd

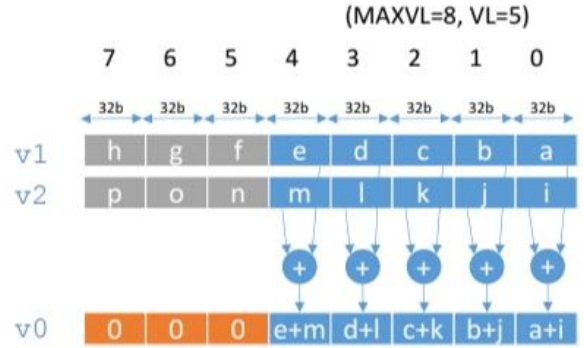
```
copi-001@copi001-OptiPlex-7050:~/ara$ cd apps
copi-001@copi001-OptiPlex-7050:~/ara/apps$ mkdir folder_name
copi-001@copi001-OptiPlex-7050:~/ara/apps$ cd folder_name/
```

- To create a new file to write the c code using editor type command nano followed by .c extension.

```
copi-001@copi001-OptiPlex-7050:~/ara/apps/folder_name$ nano main.c
copi-001@copi001-OptiPlex-7050:~/ara/apps/folder_name$ cd ../..
copi-001@copi001-OptiPlex-7050:~/ara$ make -C apps bin/folder_name
```

vfadd.s v0, v1, v2

```
for (i = 0; i < vl; i++ )
{
   v0[i] = v1[i] +F32 v2[i]
}
for (i = vl; i < MAXVL; i++ )
{
   v0[i] = 0
}
```



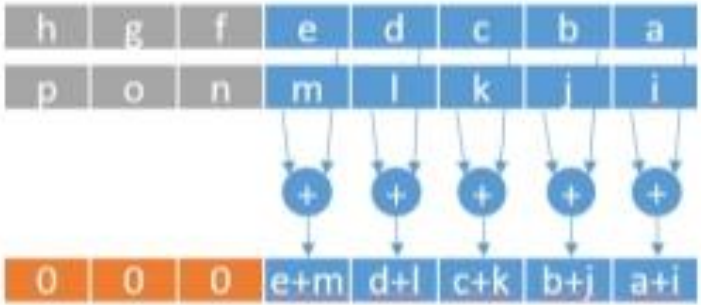
- When VL is zero, dest register is fully cleared
- Operations past ‘vl’ shall not raise exceptions
- Destination can be same as source

RISC_V vector add instruction

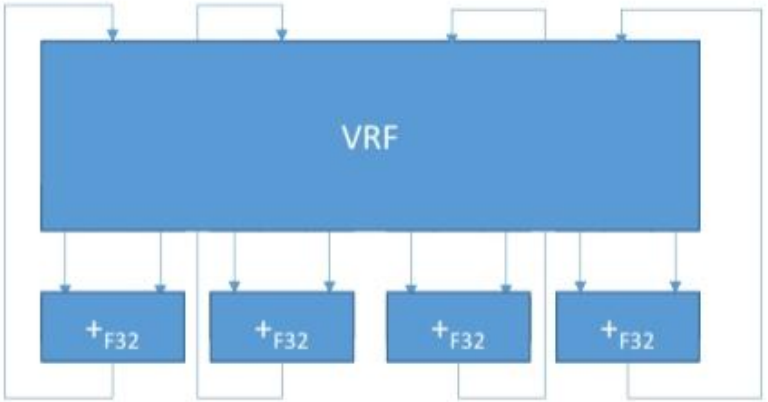
```
#include <stdio.h>
#include <riscv_vector.h>
#include <stddef.h>
#include <printf.h>

void vector_add(float *dst, float *lhs, float *rhs, size_t avl) {
   for (size_t vl; avl > 0; avl -= vl, lhs += vl, rhs += vl, dst += vl) {
      vl = vsetvl_e32m1(avl);
      vfloat32m1_t vec_src_lhs = vle32_v_f32m1(lhs, vl);
      vfloat32m1_t vec_src_rhs = vle32_v_f32m1(rhs, vl);
      vfloat32m1_t vec_acc = vfadd_vv_f32m1(vec_src_lhs, vec_src_rhs,
vl);
      vse32_v_f32m1(dst, vec_acc, vl);
   }
}
#ifndef ARRAY_SIZE
#define ARRAY_SIZE 1024
#endif
```

```
float lhs[ARRAY_SIZE];
float rhs[ARRAY_SIZE];
float dst[ARRAY_SIZE] = {0.f};
int main(void) {
   for (int i = 0; i < ARRAY_SIZE; ++i) {
  lhs[i] = (float)i / ARRAY_SIZE;
 rhs[i] = (float)(ARRAY_SIZE - i) / ARRAY_SIZE;
   }
   vector_add(dst, lhs, rhs, ARRAY_SIZE);
   for (int i = 0; i < ARRAY_SIZE; ++i) {
      printf("dst[%d] = %f\n", i, dst[i]);
   }
   return 0;
}
```

Vector Instruction Processing



4-lane implementation



1st clock: a+i, b+j, c+k, d+l

2nd clock: e+m, 0, 0, 0

Enable Vectorisation in Pulp ara

```
copi001-OptiPlex-7050:~/ara$ make -C apps bin/binary_search
ering directory '/home/copi-001/ara/apps'
_search && if [ -d script ]; then python3 script/gen_data.py
i-001/ara/install/riscv-llvm/bin/clang -march=rv64gcv_zfh_zvf
vm -scalable-vectorization=on -mllvm -riscv-v-vector-bits-min
me/copi-001/ara/apps/common -std=gnu99 -O3 -ffast-math -fno-c
nused-command-line-argument  -ffunction-sections -fdata-secti
/home/copi-001/ara/apps/common/script/align_sections.sh
me/copi-001/ara/apps/common/link.ld && cp /home/copi-001/ara/
i-001/ara/apps/common/script/align_sections.sh 4 /home/copi-0
i-001/ara/install/riscv-llvm/bin/clang -march=rv64gcv_zfh_zvf
vm -scalable-vectorization=on -mllvm -riscv-v-vector-bits-min
me/copi-001/ara/apps/common -std=gnu99 -O3 -ffast-math -fno-c
nused-command-line-argument  -ffunction-sections -fdata-secti
i-001/ara/install/riscv-llvm/bin/clang -march=rv64gcv_zfh_zvf
vm -scalable-vectorization=on -mllvm -riscv-v-vector-bits-min
me/copi-001/ara/apps/common -std=gnu99 -O3 -ffast-math -fno-c
```

Enable Vectorisation in Pulp ara

Enter command : nano apps/common/runtime.mk

Modify the following parameters

Scalable vectorisation=on

riscv-v-vector-bits-min=128

-fvectorise

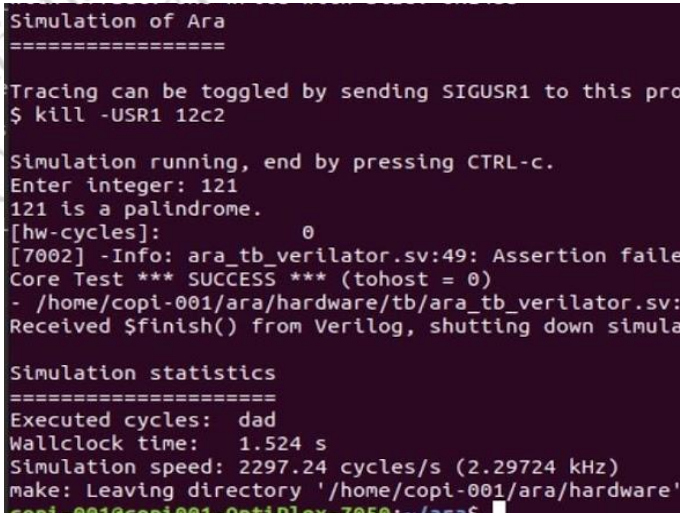
Simulation on ARA

For simulation command :

make -C hardware simv app=folder_name

Where, folder_name is the program.

folder_name = vadd



RISC-V Vector matrix multiplication

```
#include <stdio.h>
#include <riscv_vector.h>
#include <printf.h>
void print_matrix(const char *name, double *matrix, int rows, int cols) {
   printf("%s:\n", name);
   for (int i = 0; i < rows; ++i) {
      for (int j = 0; j < cols; ++j) {
         printf("%06.2f\t", matrix[i * cols + j]);
      }
      printf("\n");    }
   printf("\n");    }
void matmul(double *a, double *b, double *c, int n, int m, int o) {
   size_t vlmax = vsetvlmax_e64m1();  // Set max vector length
   for (int i = 0; i < n; ++i) {
      double *ptr_a_row = &a[i * o];
      for (int j = 0; j < m; ++j) {
         double *ptr_a = ptr_a_row;
         double *ptr_b = &b[j * o];
         int k = o;
         vfloat64m1_t vec_s = vfmv_v_f_f64m1(0, vlmax);  // Initialize vector sum
         while (k > 0) {
            size_t vl = vsetvl_e64m1(k);  // Set actual vector length for the current chunk
            vfloat64m1_t vec_a = vle64_v_f64m1(ptr_a, vl);  // Load vector from matrix A
            vfloat64m1_t vec_b = vle64_v_f64m1(ptr_b, vl);  // Load vector from matrix B
            vec_s = vfmacc_vv_f64m1(vec_s, vec_a, vec_b, vl);
            k -= vl;
            ptr_a += vl;
            ptr_b += vl;
         }
         double sum = 0;
            double vec_sum_array[vlmax];
            vse64_v_f64m1(vec_sum_array, vec_s, vlmax);
            for (size_t vi = 0; vi < vlmax; vi++) {
               sum += vec_sum_array[vi];
            }
            c[i * m + j] = sum;
      }
   }
}
int main() {
   const int N = 3;  // Size of the matrix
   const int M = 3;
   const int O = 3;
   double A[N * O] = {
      1, 2, 3,
      2, 3, 4,
      3, 4, 5
   };
   double B[O * M] = {
      1, 2, 3,
      2, 3, 4,
      3, 4, 5
   };
   double C[N * M] = {0.0};
   print_matrix("Matrix A", A, N, O);
   print_matrix("Matrix B", B, O, M);
   matmul(A, B, C, N, M, O);
   print_matrix("Matrix C", C, N, M);
   return 0;
}
```

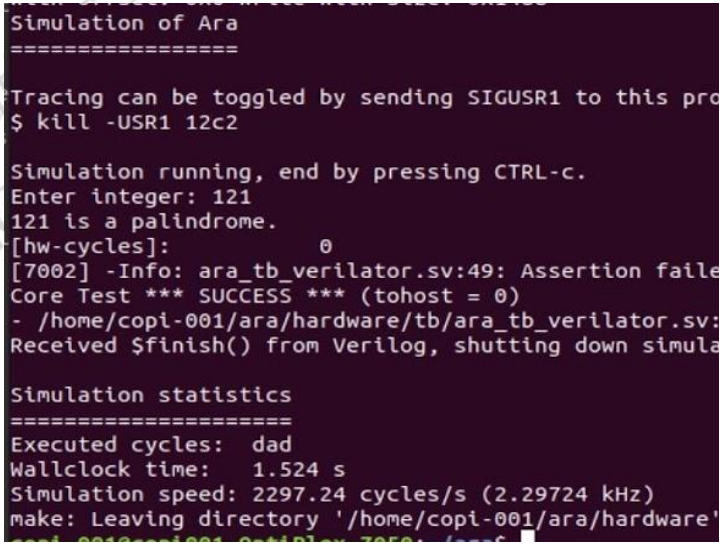
Simulation on ARA

For simulation command :

make -C hardware simv app=folder_name

Where, folder_name is the program.

folder_name = vmatrix



RISC-V Vector Subtraction

```
#include <stdio.h>
#include <riscv_vector.h>
#include <printf.h>
typedef float float32_t;
#define SIZE 16  // Number of elements in each tensor
struct onnx_tensor_t {
   void *datas;
   size_t ndata;
};
struct onnx_node_t {
   struct onnx_tensor_t **inputs;
   struct onnx_tensor_t **outputs;
};
void Sub_float32_rvv(struct onnx_node_t *n)
{
   struct onnx_tensor_t *y = n->outputs[0];
   struct onnx_tensor_t *a = n->inputs[0];
   struct onnx_tensor_t *b = n->inputs[1];
   float32_t *py = (float32_t *)y->datas;
   float32_t *pa = (float32_t *)a->datas;
   float32_t *pb = (float32_t *)b->datas;
   size_t blkCnt = y->ndata;
   size_t l;
   while (blkCnt > 0)
   {
      l = vsetvl_e32m8(blkCnt);
      vfloat32m8_t va = vle32_v_f32m8(pa, l);
      vfloat32m8_t vb = vle32_v_f32m8(pb, l);
      vfloat32m8_t vc = vfsub_vv_f32m8(va, vb, l);
      vse32_v_f32m8(py, vc, l);
      pa += l;
      pb += l;
      py += l;
      blkCnt -= l;
   }  }
int main()
{
   float32_t a_data[SIZE];
   float32_t b_data[SIZE];
   float32_t y_data[SIZE];
   for (size_t i = 0; i < SIZE; i++) {
      a_data[i] = (float32_t)i;
      b_data[i] = (float32_t)(SIZE - i);
   }
   struct onnx_tensor_t a = { a_data, SIZE };
   struct onnx_tensor_t b = { b_data, SIZE };
   struct onnx_tensor_t y = { y_data, SIZE };
   struct onnx_tensor_t *inputs[] = { &a, &b };
   struct onnx_tensor_t *outputs[] = { &y };
   struct onnx_node_t node = { inputs, outputs };
   Sub_float32_rvv(&node);
   printf("Result of a - b:\n");
   for (size_t i = 0; i < SIZE; i++) {
      printf("y[%zu] = %f\n", i, y_data[i]);
   }
   return 0;
}
```

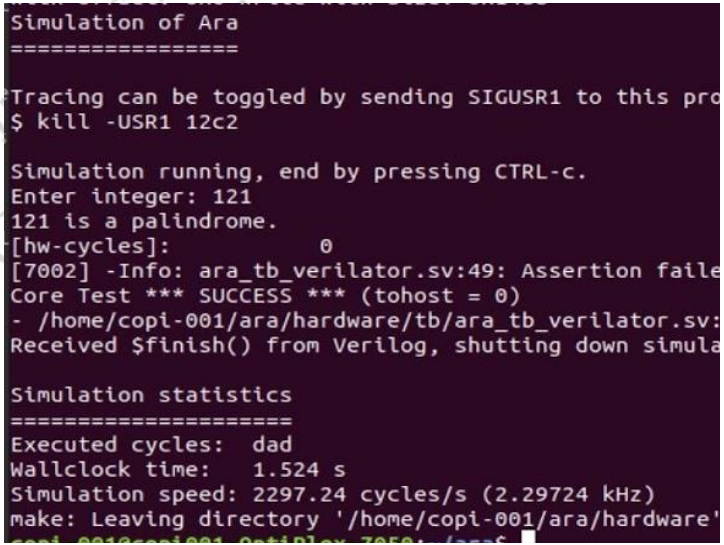
Simulation on ARA

For simulation command :

make -C hardware simv app=folder_name

Where, folder_name is the program.

folder_name = vmatrix



RISC-V Vector Reciprocal

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <riscv_vector.h>
typedef float float32_t;
#define MAX_SIZE 16
struct onnx_tensor_t {
   float32_t *datas;
   size_t ndata;  // number of elements
};
struct onnx_node_t {
   struct onnx_tensor_t *inputs[1];
   struct onnx_tensor_t *outputs[1];
};
void Reciprocal_float32_scalar(struct onnx_node_t *n)
{
   struct onnx_tensor_t *x = n->inputs[0];
   struct onnx_tensor_t *y = n->outputs[0];
   float32_t *px = x->datas;
   float32_t *py = y->datas;
   for (size_t i = 0; i < x->ndata; ++i)
   {
      py[i] = 1.0f / px[i];
   }
}
```

```
 while (vblkCnt > 0) {
      vl = vsetvl_e32m8(vblkCnt);
      vx = vle32_v_f32m8(px, vl);
      vy = vfrdiv_vf_f32m8(vx, 1.0f, vl);  // Reciprocal: 1.0 / x
      vse32_v_f32m8(py, vy, vl);

px += vl;
      py += vl;
      vblkCnt -= vl;
   } }
int main()
{
   float32_t input[MAX_SIZE]  = {1.0f, 2.0f, 4.0f, 0.5f, 10.0f, 0.25f, 100.0f, 0.1f, 8.0f, 0.05f, 0.2f, 16.0f,
32.0f, 64.0f, 128.0f, 0.01f};
   float32_t output[MAX_SIZE] = {0};  // Output buffer
   struct onnx_tensor_t x = { .datas = input,  .ndata = MAX_SIZE };
  struct onnx_tensor_t y = { .datas = output, .ndata = MAX_SIZE };
   struct onnx_node_t node = { .inputs = { &x }, .outputs = { &y } };
   Reciprocal_float32_rvv(&node);
   printf("Input:\t\tReciprocal:\n");
   for (size_t i = 0; i < MAX_SIZE; ++i) {
      printf("%f\t%f\n", input[i], output[i]);
   }
   return 0;
}
```

Simulation on ARA

For simulation command :

make -C hardware simv app=folder_name

Where, folder_name is the program.

folder_name = vreciprocal