Vector Extension Proposal

Colin Schmidt, Albert Ou, Yunsup Lee, Krste Asanović
krste@eecs.berkeley.edu
http://www.riscv.org

2nd RISC-V Workshop, Berkeley, CA
June 29, 2015

Goals for RISC-V Standard V Extension

- Efficient and scalable to all reasonable design points
  - Low-cost or high-performance
  - In-order, decoupled, or out-of-order microarchitectures
  - Integer, fixed-point, and/or floating-point data types
- Good compiler target
- Support both implicit auto-vectorization (OpenMP) and explicit SPMD (OpenCL) programming models
- Work with virtualization layers
- Fit into standard fixed 32-bit encoding space
- Be base for future vector++ extensions

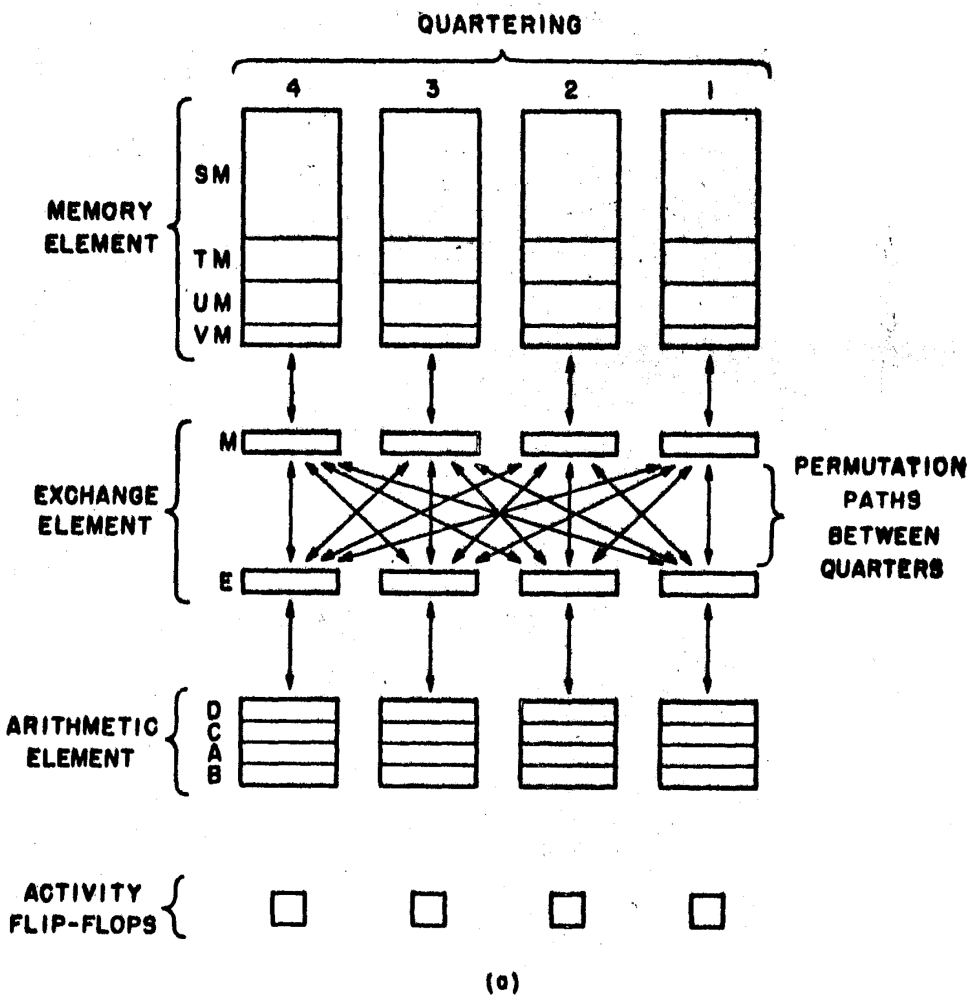
- Non-goal: Look like everyone's Packed-SIMD
- Non-goal: Look like a GPU

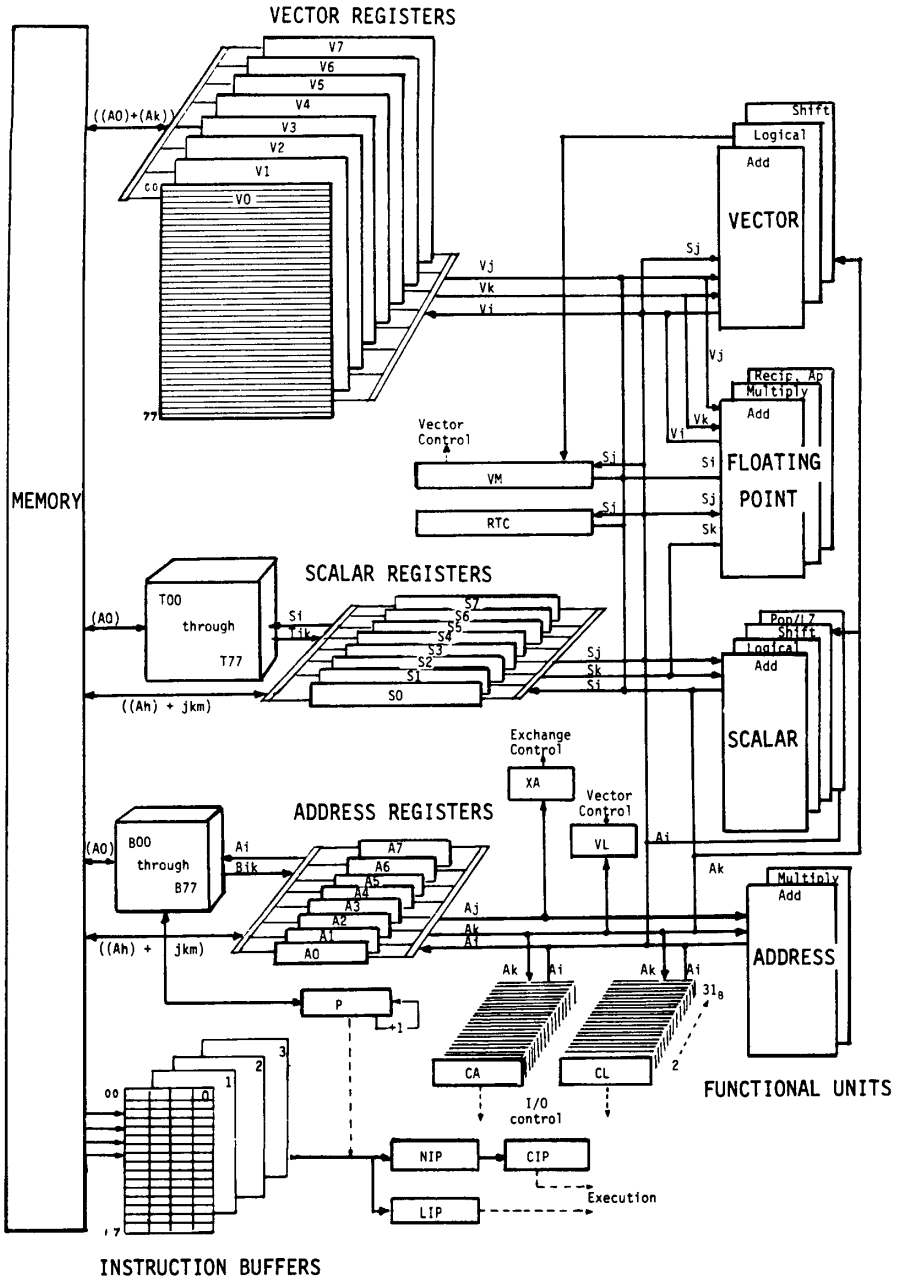
Packed-SIMD versus Traditional Vectors

- 1950s' Packed-SIMD

- 1970s' Vectors

Packed-SIMD versus Traditional Vectors





GPU versus Traditional Vector

- GPU
  - Discrete accelerator running in own memory space
    - More recently support sharing of physical, or pinned virtual, but still in different memory hierarchy and no page faults
  - Terrible at scalar code
  - Only effective on very large data-parallel tasks (>10,000s)
  - ISA/microarchitecture evolved from graphics shader needs, not general compute
  - Only seems efficient compared to out-of-order scalar core
- Traditional vector
  - Coprocessor running in same memory hierarchy as scalar
  - Tightly coupled to scalar core
  - Effective with loop counts of 2 or more
  - ISA evolved from general computing needs
  - Very efficient

Rest of Talk Outline

- Why traditional vectors are better than Packed-SIMD or GPUs
- What we're proposing for V extension

Autovectorization Programming Model

SAXPY

```
for (i=0; i<n; i++) {
  y[i] = a*x[i] + y[i];
}
```

Autovectorization
Programming Model

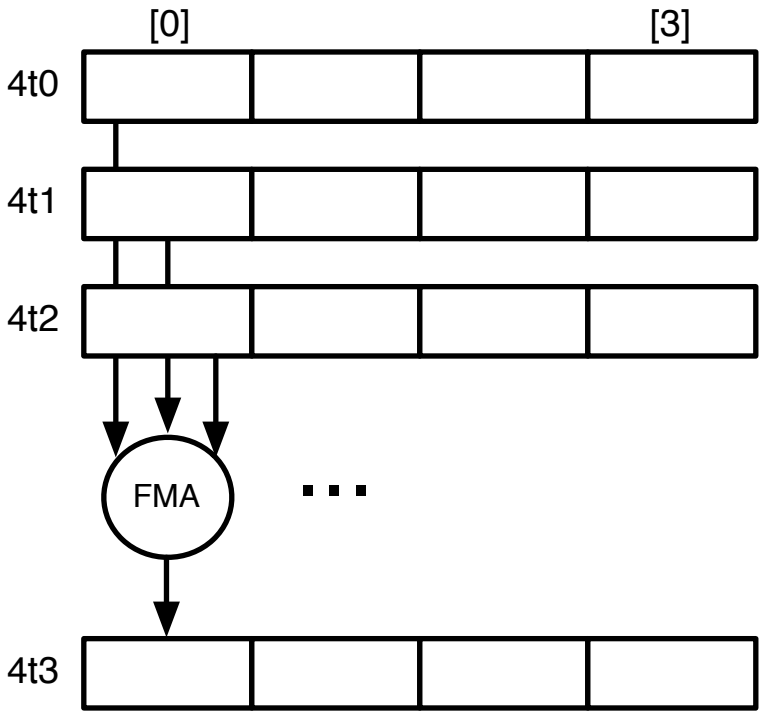
- Mostly automatic, possibly some restructuring from the application writer

SAXPY on Packed-SIMD Architecture

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

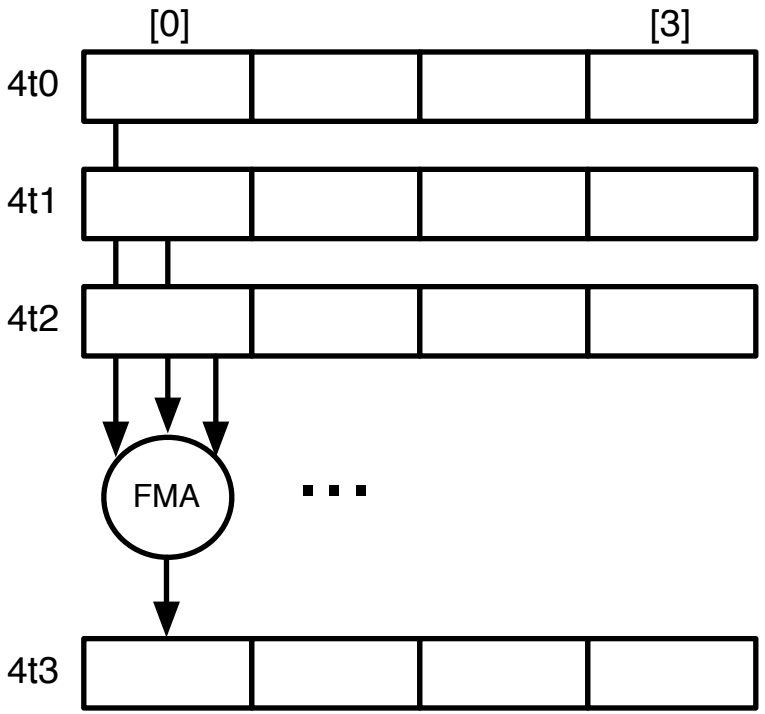


SAXPY Function Arguments

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

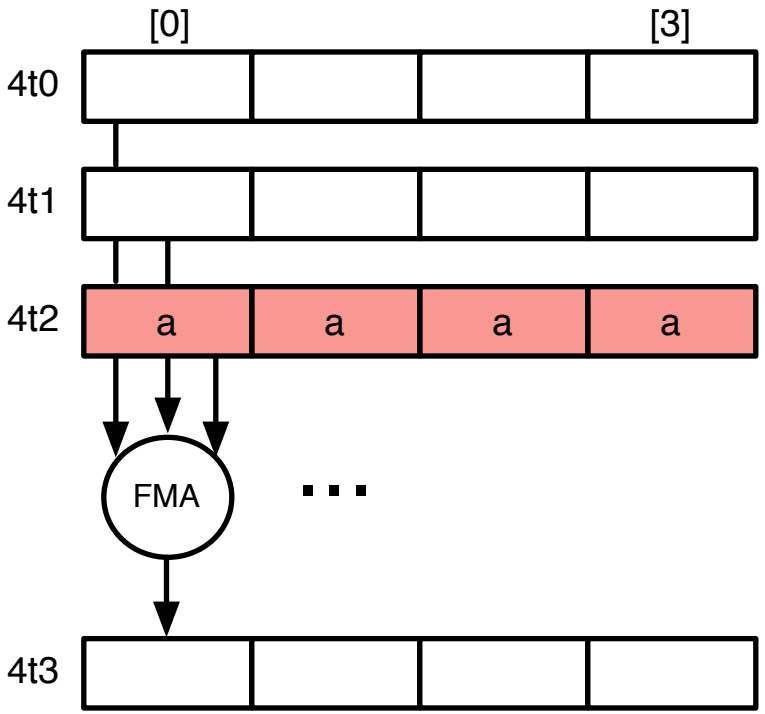


Convert Scalar to Vector

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

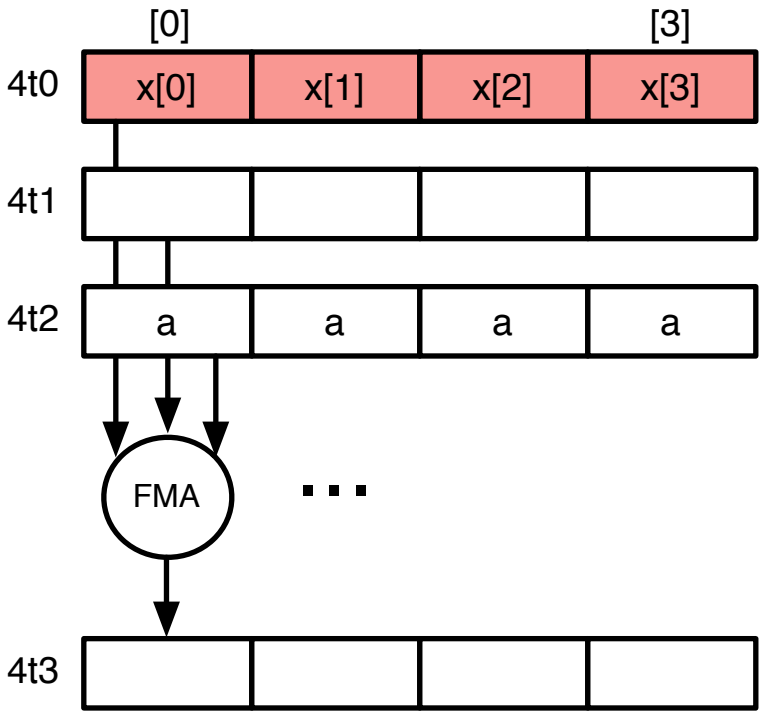


Load X Vector

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

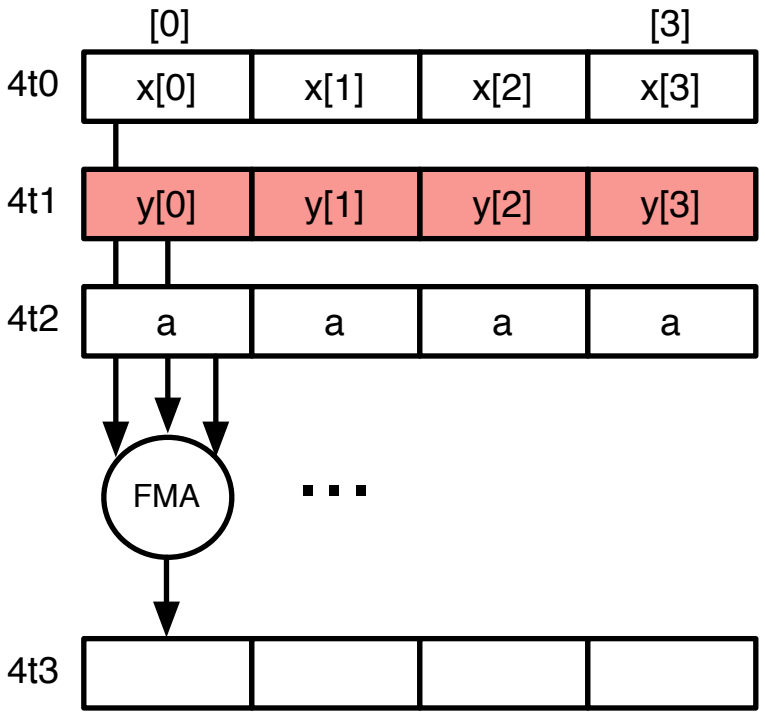


Load Y Vector

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

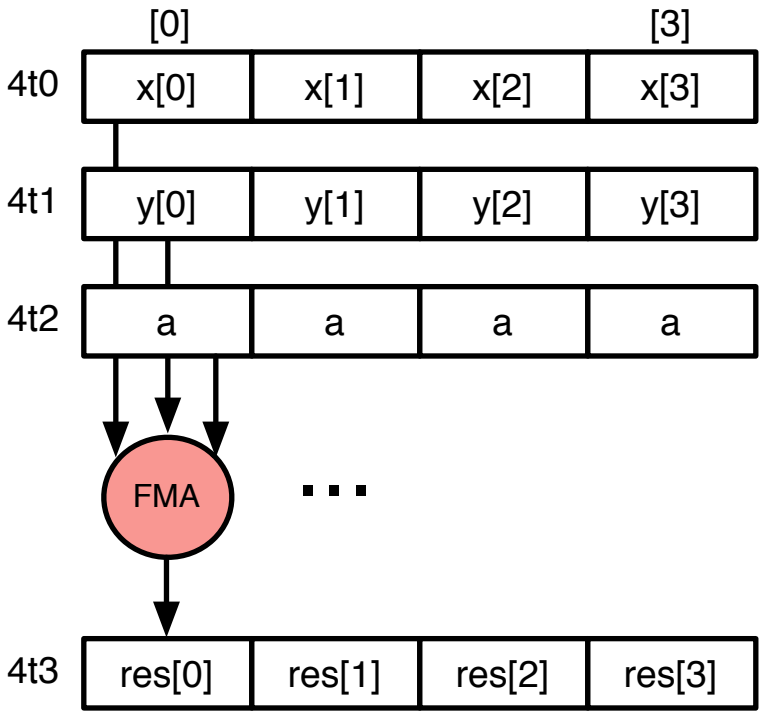


Compute Multiply-Add Result

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

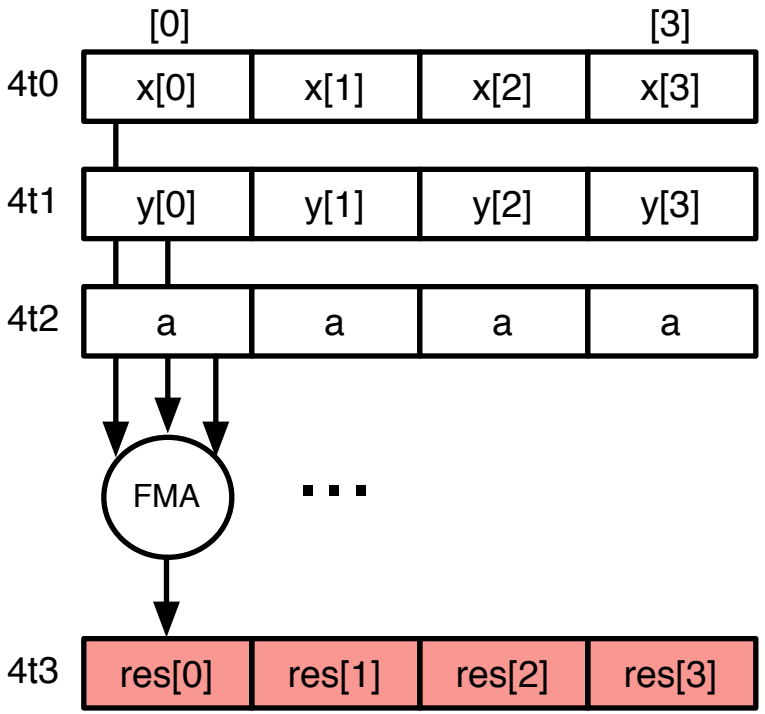


Vector Store of Results

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

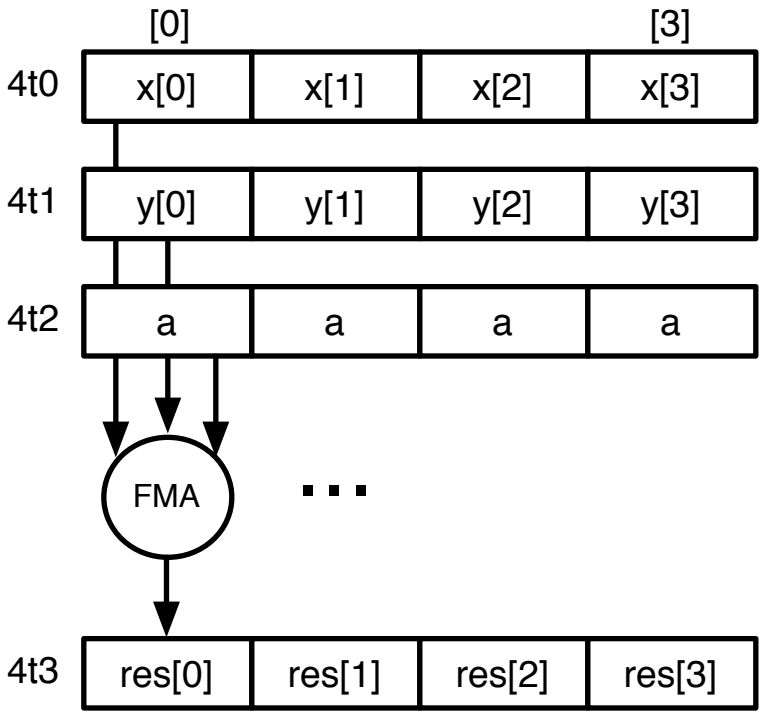


Increment Address Pointers

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

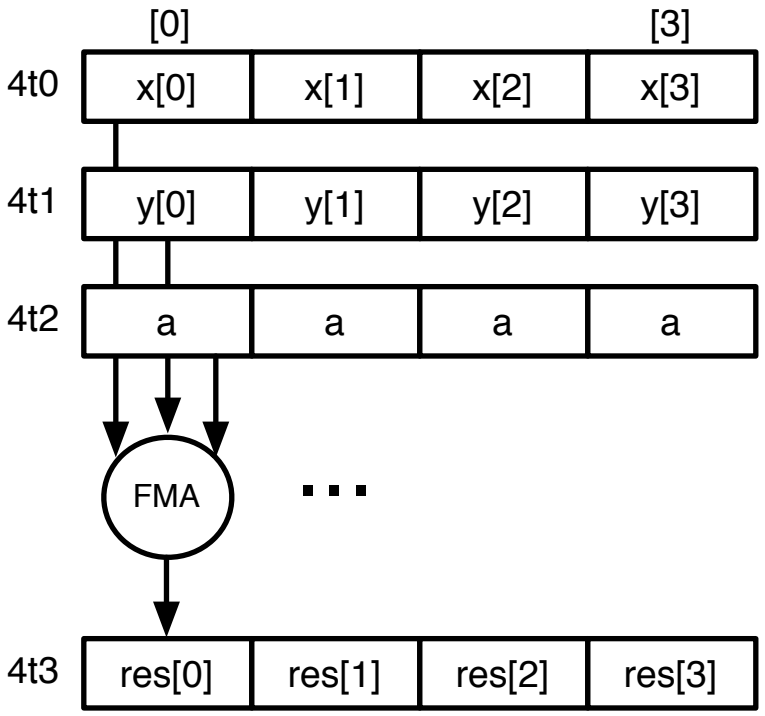


Stripmine Loop Checks

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

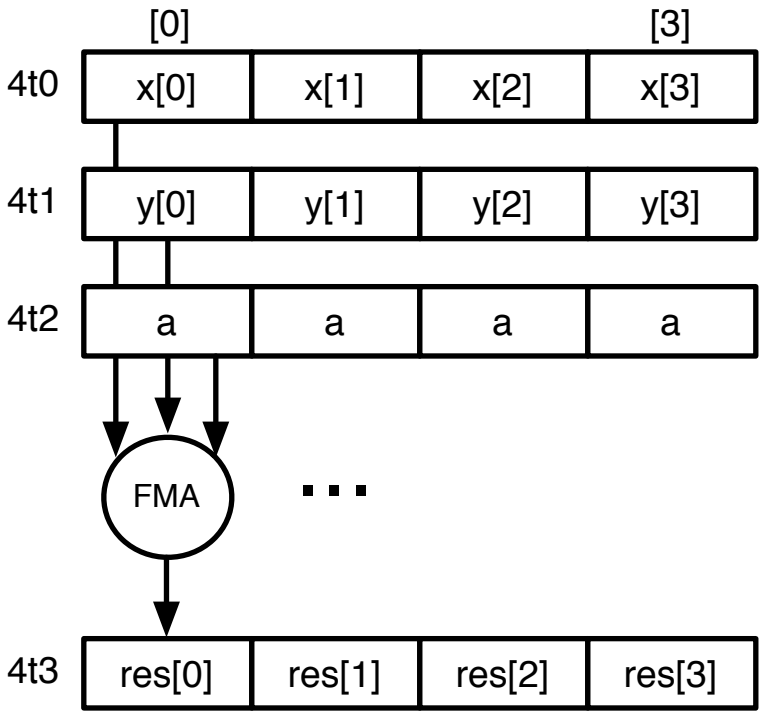


Handle Remainder Strip

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD



Port from 4-wide to 8-wide Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat8 8t2, a1
stripmine:
  vlw8 8t0, a2
  vlw8 8t1, a3
  vfma8 8t3, 8t2, 8t0, 8t1
  vsw8 8t3, a3
  addi a2, a2, 8<<2
  addi a3, a3, 8<<2
  sub a0, a0, 8
  bgte a0, 8, stripmine
  . . .
  handle even more edge cases
```

New and Improved Packed-SIMD

Packed-SIMD vs. Traditional Vectors

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Set Vector Length Register
min(app_length, hardware_length)

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Vector Loads

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Vector Multiply-Add Compute (vector and scalar operands)

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Vector Store Results

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Increment Address Pointers

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Stripmine Loop (note, no edge cases!)

```
a0: n, a1: a, a2: *x, a3: *y

  vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
  handle edge cases
```

Packed-SIMD

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

SPMD Programming Model

```
Kernel(int n, float a,
       float* x, float* y) {
  if (tid < n) {
    y[tid] = a*x[tid]+y[tid];
  }
}

Kernel<<<(n+31)/32*32>>>
  (n, a, x, y);
```

SPMD
Programming Model

- Classic model reappears in CUDA/OpenCL
- Same restructuring as autovectorization, plus more on top to get performance

Explicit Thread (Element) Identifier

```
Kernel(int n, float a,
       float* x, float* y) {
  if (tid < n) {
    y[tid] = a*x[tid]+y[tid];
  }
}

Kernel<<<(n+31)/32*32>>>
  (n, a, x, y);
```

SPMD
Programming Model

Explicit Check if Thread Active

```
Kernel(int n, float a,
       float* x, float* y) {
  if (tid < n) {
    y[tid] = a*x[tid]+y[tid];
  }
}

Kernel<<<(n+31)/32*32>>>
  (n, a, x, y);
```

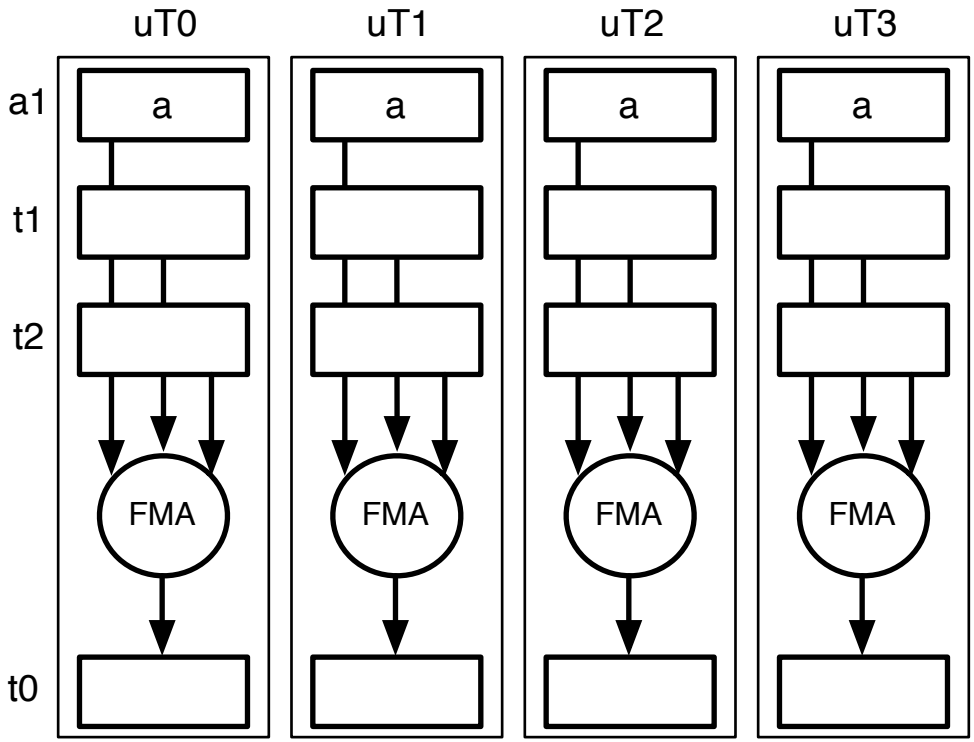
SPMD
Programming Model

GPU/SIMT Architecture

```
a0: n, a1: a,
 a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a1, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

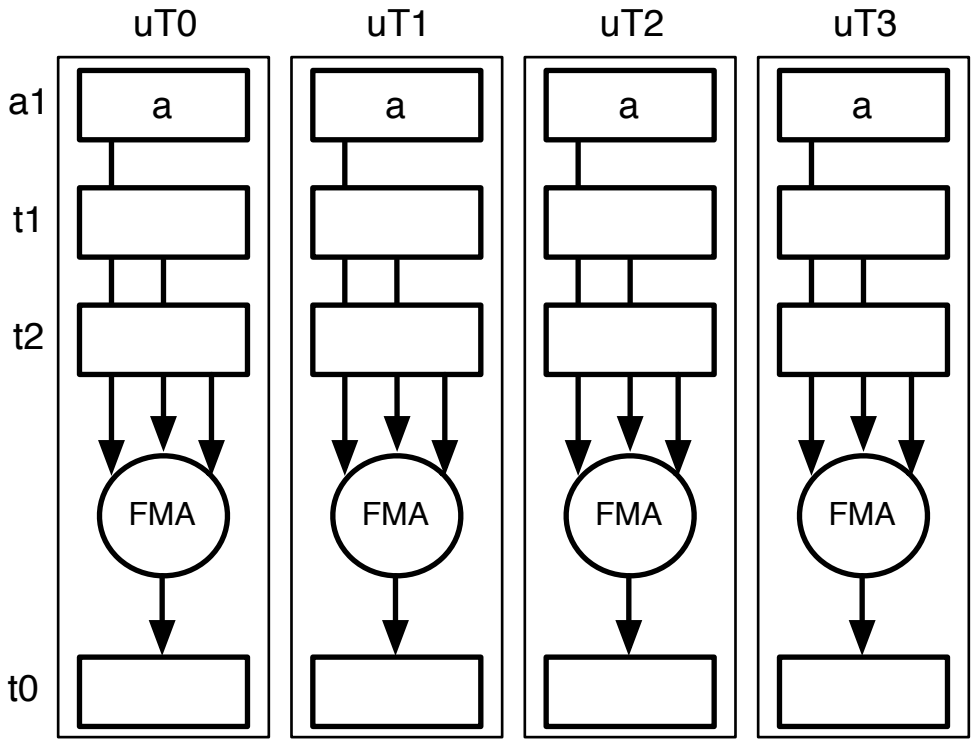


Check if active

```
a0: n, a1: a,
 a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a1, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

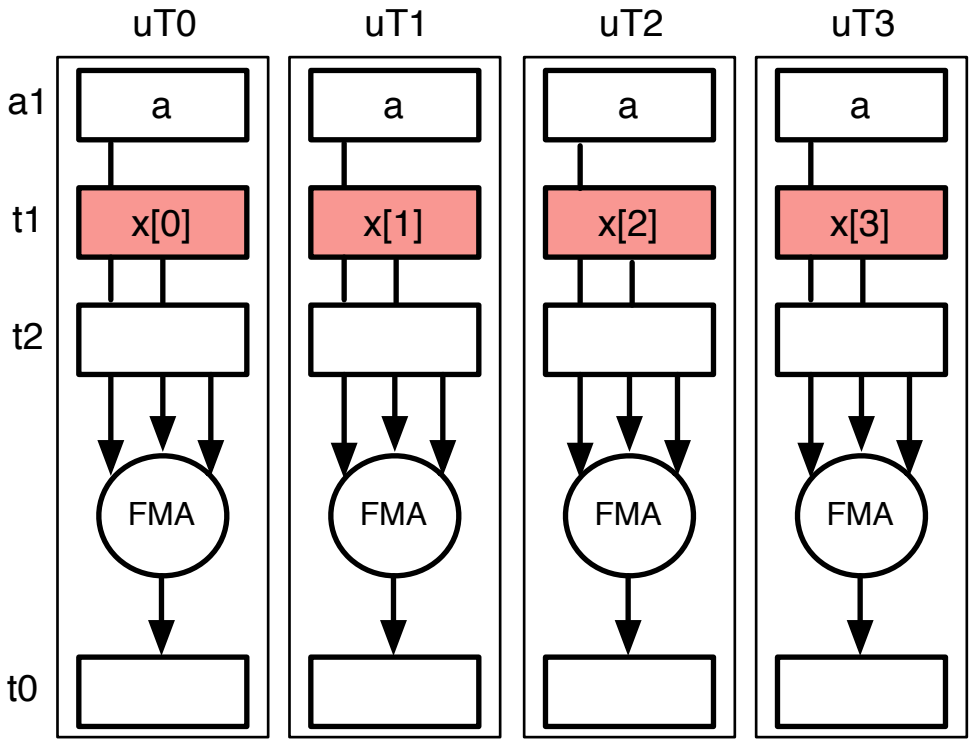


Calculate address and load X elements

```
a0: n, a1: a,
 a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a1, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

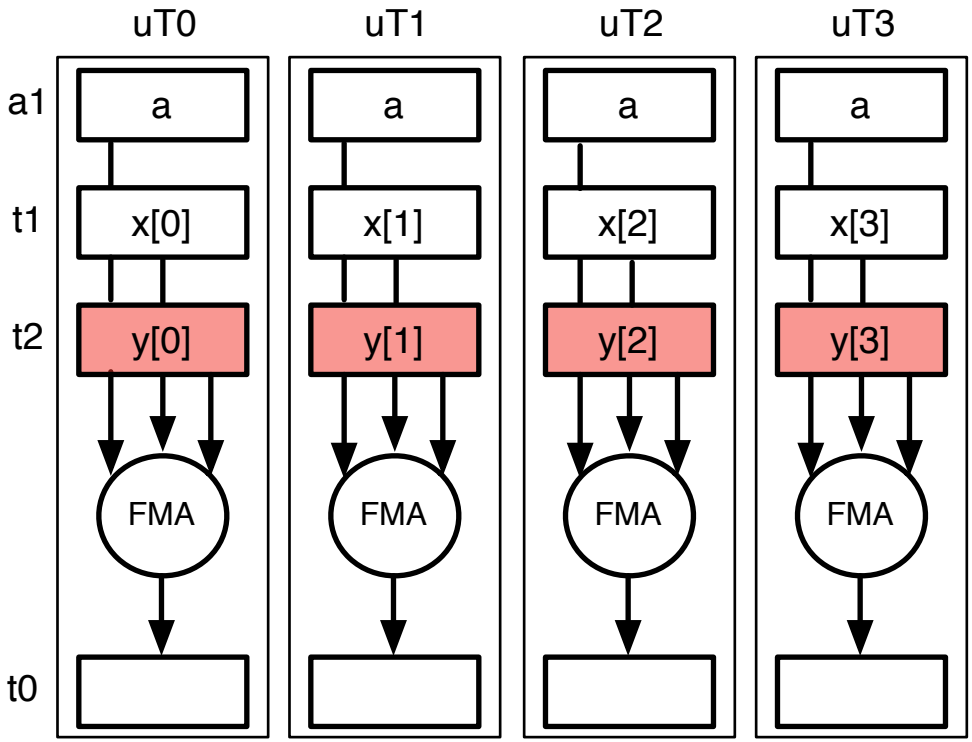


Address calc and load Y elements

```
a0: n, a1: a,
 a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a1, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

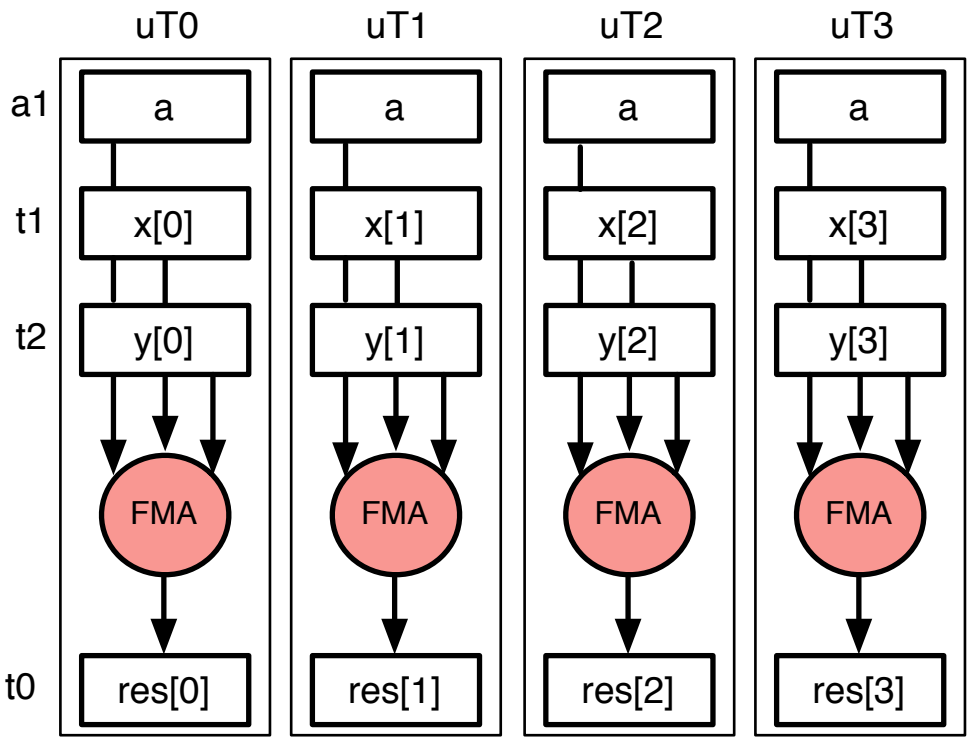


Compute Element Results

```
a0: n, a1: a,
 a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a1, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

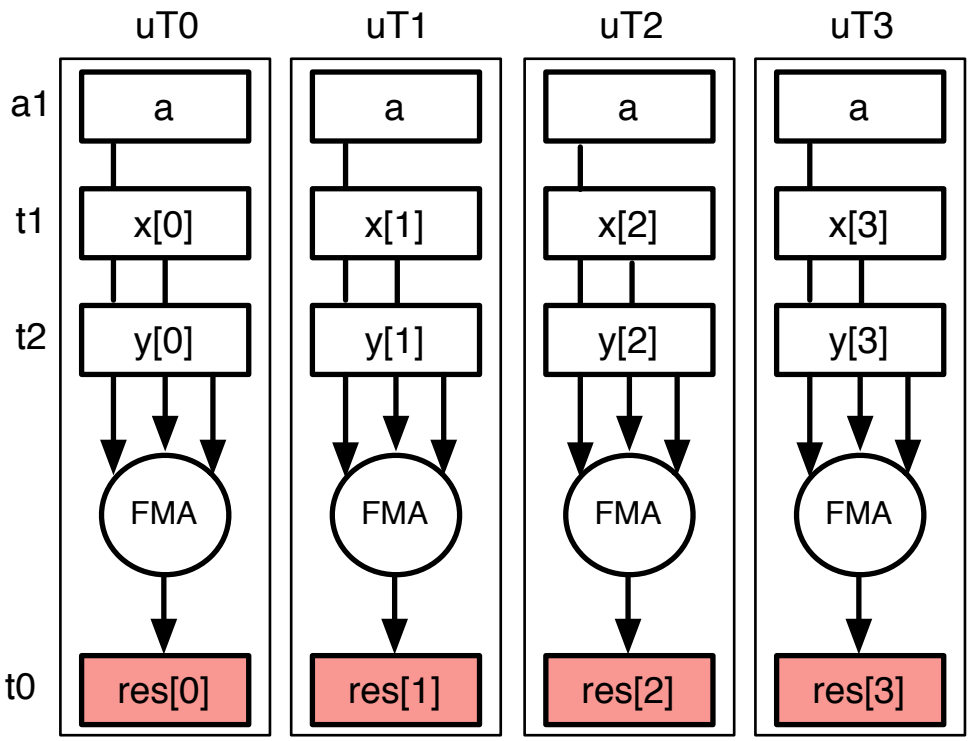


Store Result Elements

```
a0: n, a1: a,
 a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a1, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT



GPU/SIMT vs. Traditional Vectors

```
a0: n, a1: a, a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a0, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

Vector Length avoids Activity Branch

```
a0: n, a1: a, a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a0, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

GPU/SIMT replicates scalar operands

```
a0: n, a1: a, a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a0, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vectors

GPU/SIMT redundant address calculations, dynamic memory coalescing

```
a0: n, a1: a, a2: *x, a3: *y

  mv t0, tid
  bge t0, a0, skip
  slli t0, t0, 2
  add a2, a2, t0
  add a3, a3, t0
  lw t1, 0(a2)
  lw t2, 0(a3)
  fma.s t0, a0, t1, t2
  sw t0, 0(a3)
skip:
  stop
```

SIMT

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

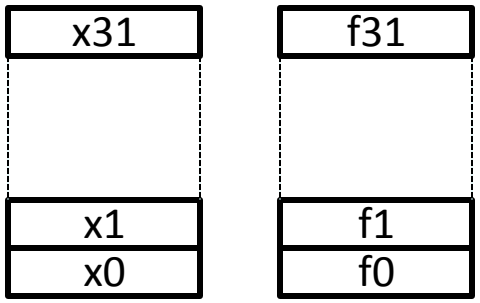
Traditional Vectors

Don’t need SIMT for SPMD

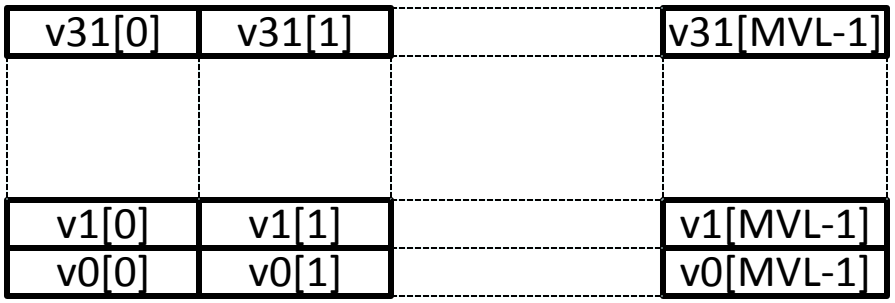
- Original CUDA model developed for NVIDIA SIMT engines, but don’t need SIMT hardware to run SPMD (CUDA/OpenCL) programs well
  - Check out Yunsup’s MICRO-2014 paper and upcoming thesis

Proposed V Extension State

Standard RISC-V scalar x and f registers



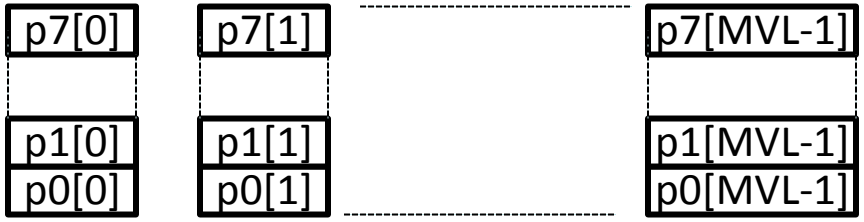
Up to 32 vector data registers, v0-v31, of at least 4 elements each, with variable bits/element (8,16,32,64,128)



MVL is maximum vector length, implementation and configuration dependent, but MVL >= 4

Vector configuration CSR

Vector length CSR



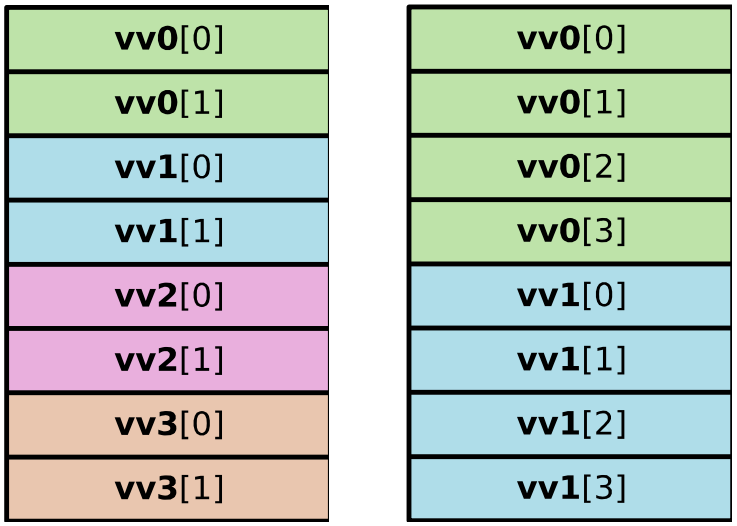
8 vector predicate registers, with 1 bit per element

V Extension Features

- Reconfigurable vector registers
  - Exchange unused architectural registers for longer vectors
- Mixed-precision support
  - From 8-bit to MAX(XLEN,FLEN) in powers of 2
- Integer, fixed-point, floating-point arithmetic
  - Floating-point requires corresponding scalar extension
  - Fixed-point to include rounding, saturation, scaling
- Unit-stride, Strided, Indexed Load/Stores
- Predication

Reconfigurable Vector Register File

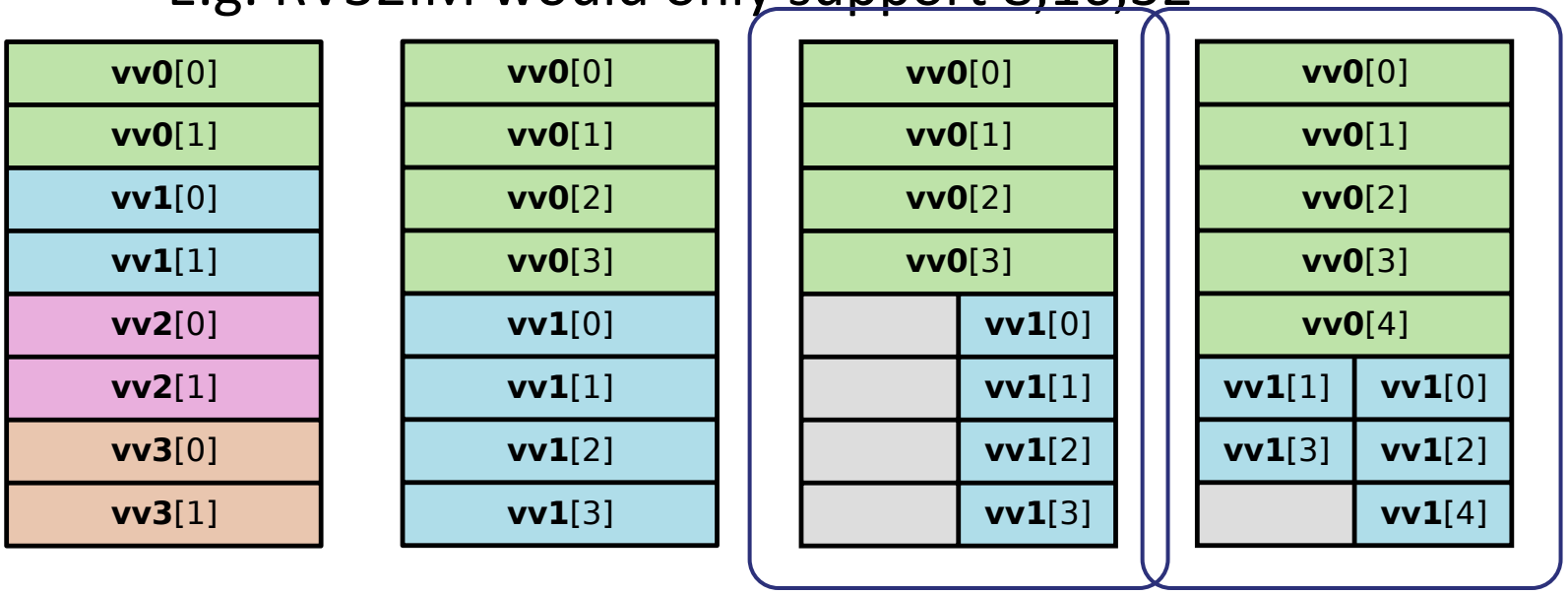
- Programming model allows specifying number of architectural registers (1-32)
- Maximum hardware vector length automatically extends to fill capacity of register file



- Exchange unused architectural registers for longer hardware vectors

Mixed-Precision Support

- Hardware subdivides physical register into multiple narrower architectural registers as requested
  - Subword packing transparent to software
  - Improved utilization of operand communication bandwidth
  - Spatial functional-unit parallelism
- Only support elements up to MAX(XLEN,FLEN)
  - E.g. RV32IM would only support 8,16,32




```
vsetcfg 1, 1
  vlen = 5
```

Programmer's View of Reconfigurability

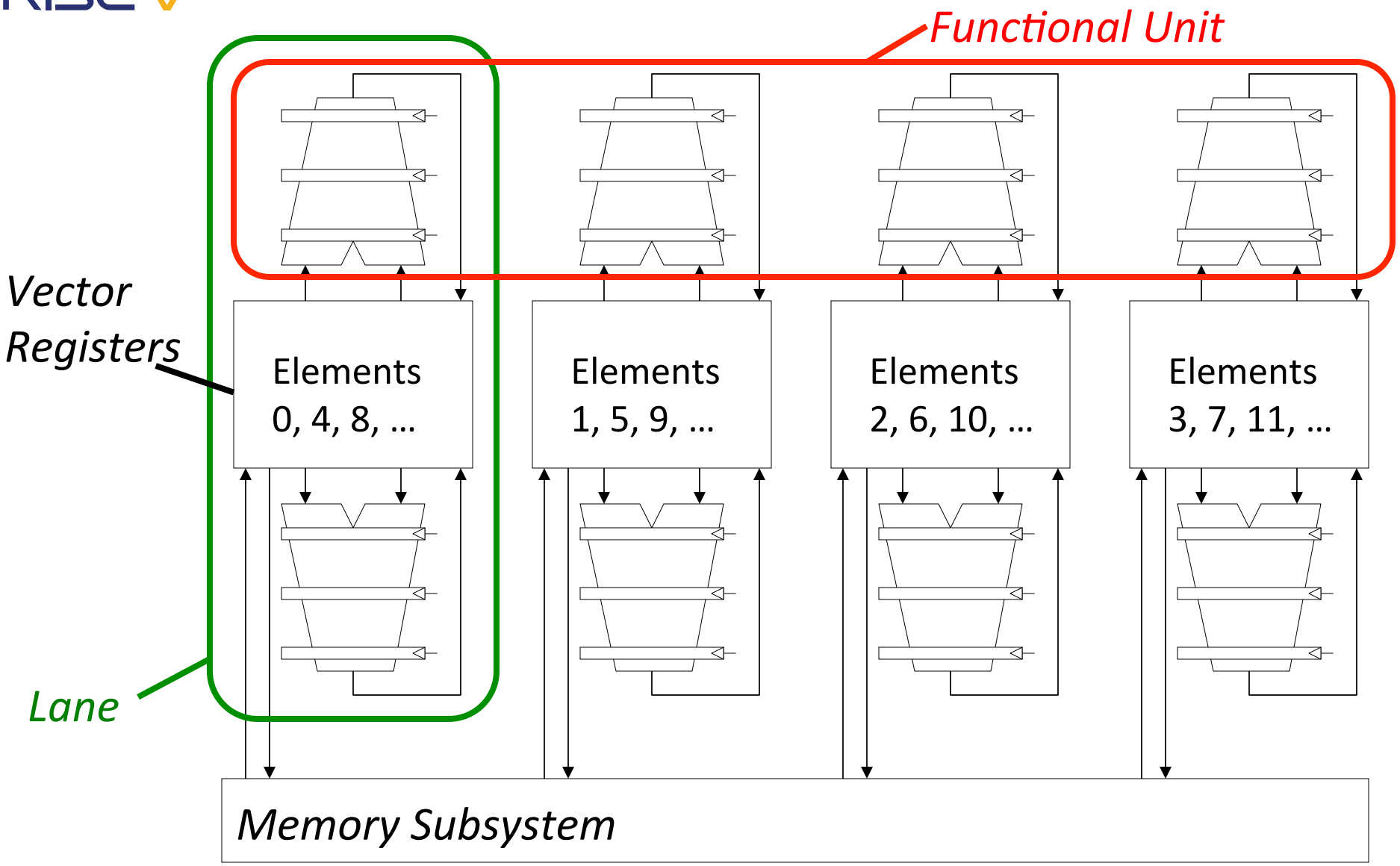
```
    vsetcfg #64, #32, #16, #8

stripmine:
    vsetvl t0, a0
    # …
    # Code for loop body
    # …
    sub a0, a0, t0
    bnez a0, stripmine

    vuncfg # Turn off vector unit
```

- Before loop nest, vector configure instruction sets number required of each width of register
- After loop nest, unconfigure disables vector unit and avoid save/restore at context swap, potentially power down

Higher Performance from Parallel Lanes



Vector Length Portability

- Same binary code works regardless of:
  - Number of physical register bits
  - Number of physical lanes

- Architecture guarantees minimum vector length of four regardless of configuration to avoid stripmine overhead for short vectors
  - E.g., if use 32 * 64-bit vector registers,
  - need 128 * 8-byte physical element registers
  - 1KB SRAM

Polymorphic Instruction Encoding

- Single signed integer ADD opcode works on different size inputs and outputs
  - Size of inputs and outputs inherent in register number
  - Sign-extend smaller input
  - Modulo arithmetic on overflow to destination
  - Restrict supported combinations to simplify hardware
- Integer, Fixed-point, Floating-point arithmetic
- Pros:
  - Denser encoding, sizes inherent in register number
  - Eliminates many difficult cases
- Cons:
  - Can't reuse register for different sizes
  - Can't initialize from memory with smaller type

Vector Loads and Stores

Addressing modes:

- Unit-stride (scalar base)
- Constant stride (scalar base, scalar stride)
- Indexed (scalar base, vector offset)

Types:

- Separate integer and floating-point loads and stores
  - Support FPU internal recoding
- Size inherent in destination register number (for integers, signed/unsigned determined by use)

Support vector AMOs:

- E.g, Vector fetch-and-add

Vector Predication

- Eight vector predicate registers p0-p7, one bit per element
- Logical operations between predicate registers
- All vector instructions are predicated under p0
  - Implicit predicate due to encoding constraints
- Instruction to swap two predicate registers
  - Reduce overhead of scheduling complex control flow
  - Can implement just in rename table if OoO core
- Popcount instruction returns number of active bits in predicate register to scalar integer register
  - Used for divergent control flow optimizations
- Other cross-element flag operations to support complex loop optimizations

Vector Predication and Vector Register Renaming

- Two previous approaches in vector archs:

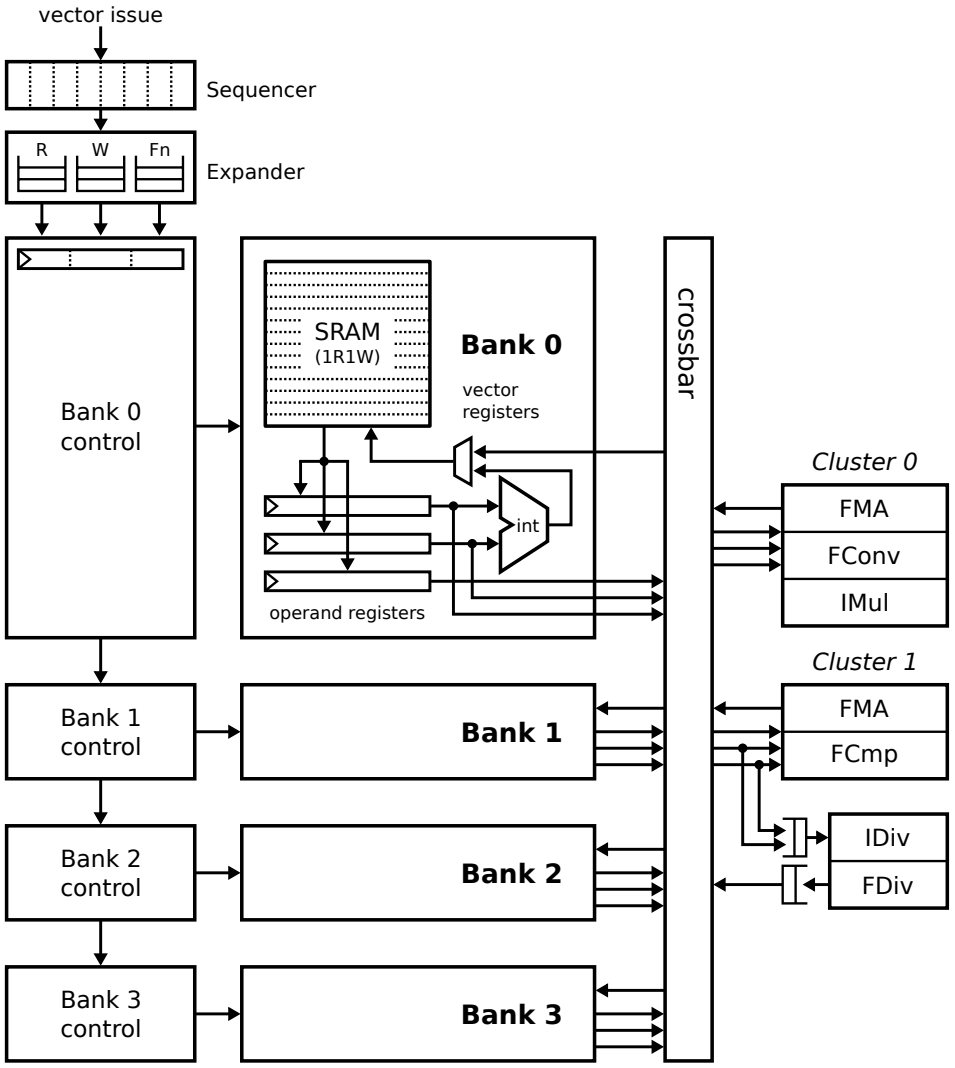
1) Destination has old value if predicate false
   - Simpler spec, better for in-order/no renaming
   - Have to copy old value to new destination with renaming
2) Destination has undefined value if predicate false
   - More complex code, better for out-of-order with renaming
   - Need additional merge(s) to rebuild complete vector

- We're choosing 1), as simpler and safer.
- Use microarchitectural tricks for OoO machines to reduce amount of data transfer.

V versus Xhwacha

- V is proposal for a standard RISC-V vector extension
- Xhwacha is a non-standard Berkeley vector extension designed to push state-of-the-art in-order/decoupled vector machines
  - V and Xhwacha lane microarchitecture very similar
  - Multiple versions of Xhwacha have been fabricated
    - up to 34 GFLOPS/W running DGEMM with IEEE-2008 64-bit fused muladds
- Current Berkeley focus on bringing up OpenCL for Xhwacha
- V to follow

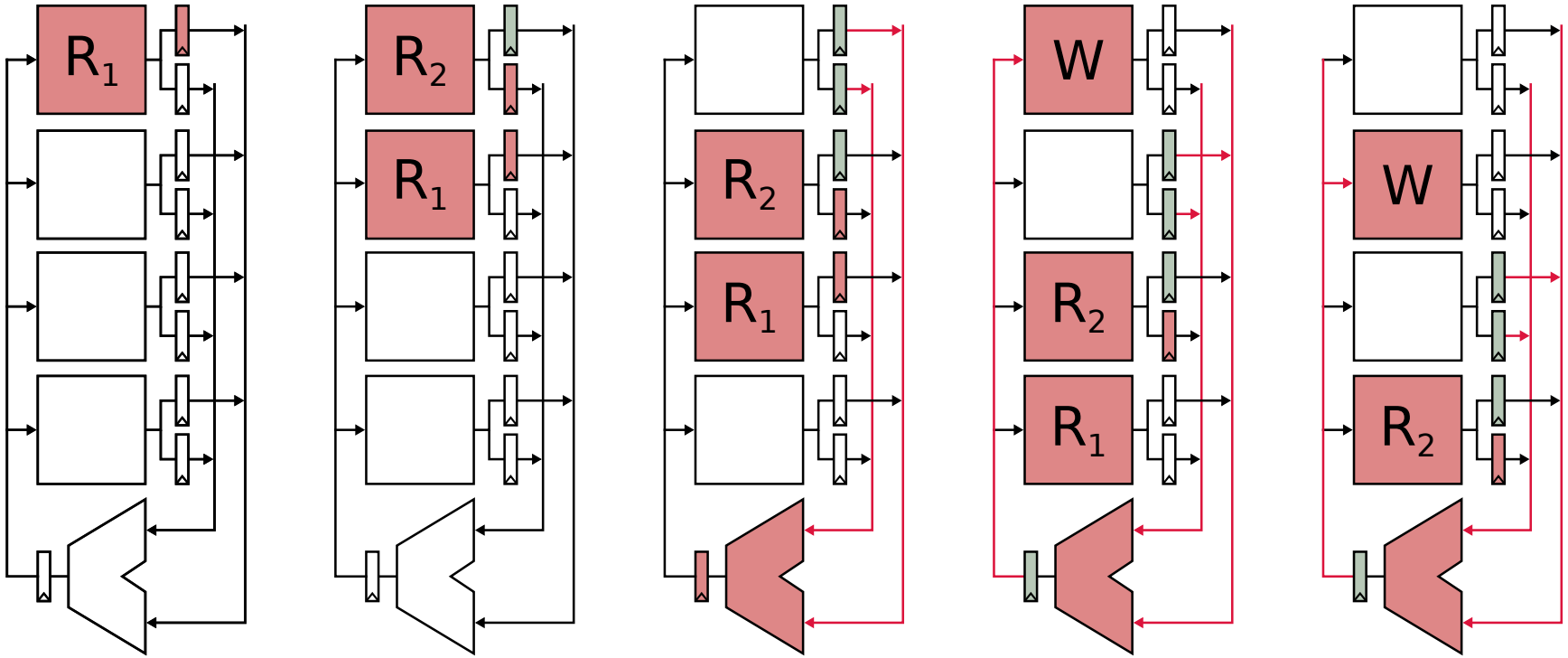
Example Vector Lane Organization (from Hwacha)



- Compact register file of four 1R1W SRAM banks
- Per-bank integer ALU
- Two independently scheduled FMA clusters
  - Total of four double-precision FMAs per cycle
- Pipelined integer multiplier
- Variable-latency decoupled functional units
  - Integer divide
  - Floating-point divide with square root

Systolic Bank Execution

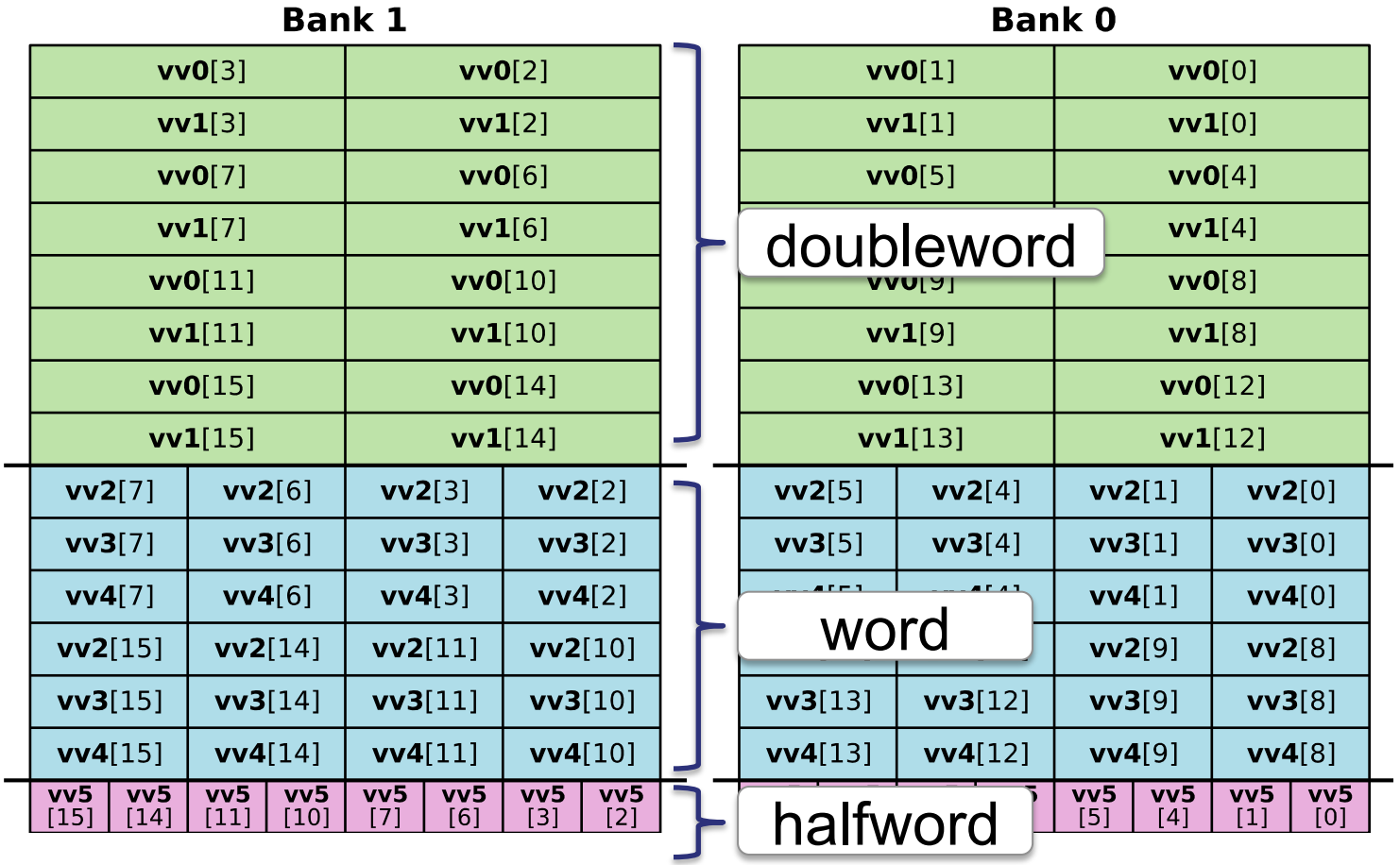
- Sustains *n* operands/cycle after *n*-cycle initial latency



- "Fire and forget" after hazards are cleared upon sequencing
- Chaining follows naturally from interleaving µops belonging to dependent instructions

Physical Vector Register File

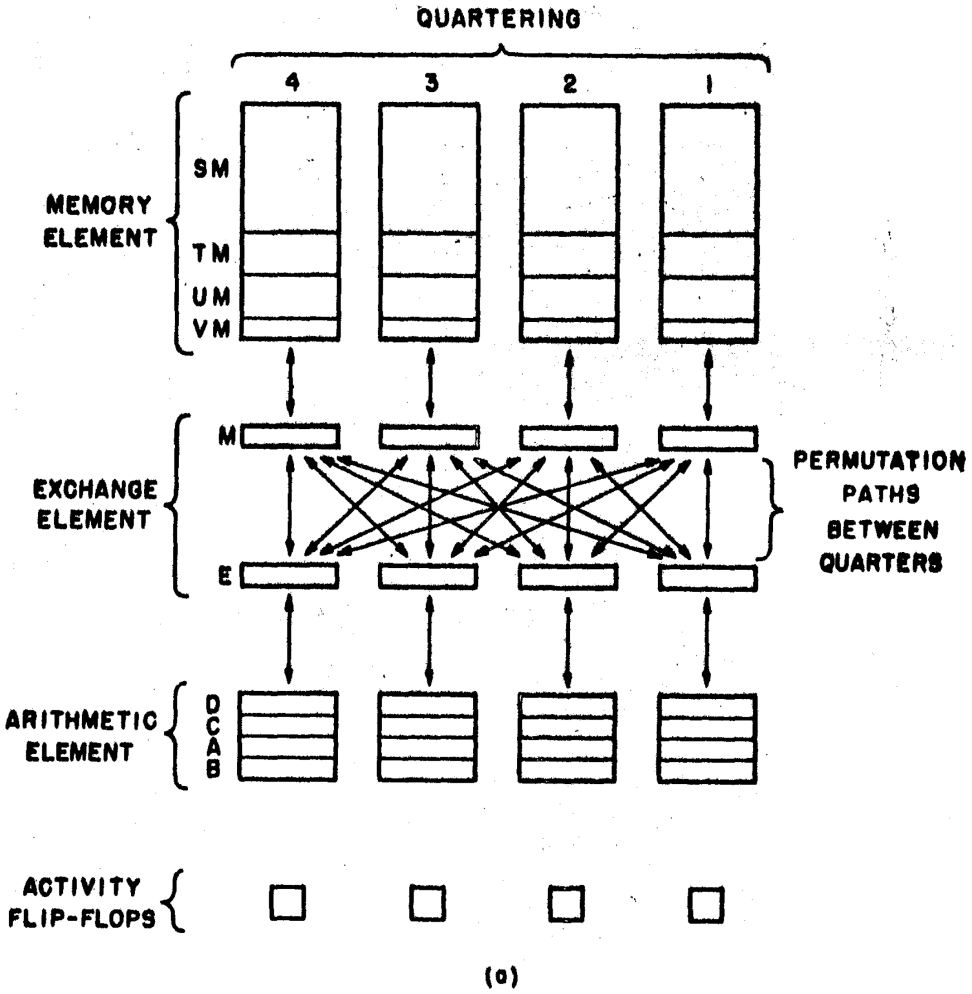
- Bank partitioned into different segments for each supported data type width
  - `vsetcfg 2, 3, 1, 0` (*#64, #32, #16, #8*) ➡ vlen = 16



OS Support

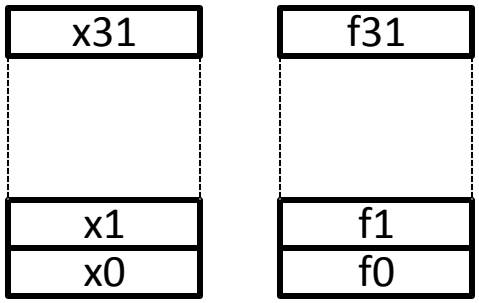
- Restartable page faults via microcode state dump, opaque to OS
  - Similar to DEC Vector Vax implementation
- Privileged specification describes XS sstatus field used to encode coprocessor status (Off, Initial, Clean, Dirty) to reduce context save/restore overhead.

Minimal V Implementation

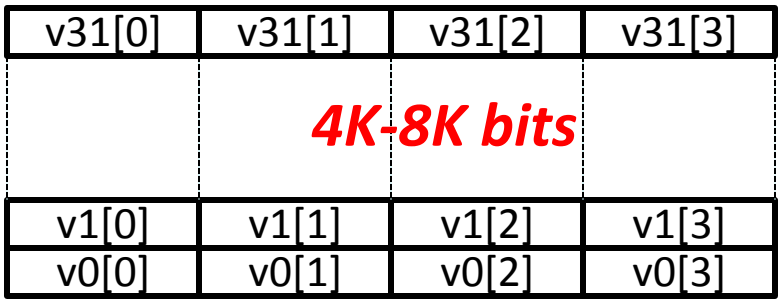


Minimal V Coprocessor Implementation

Standard RISC-V scalar x and f registers



32 vector data registers, v0-v31, four elements each, MAX(XLEN,FLEN)-bits wide

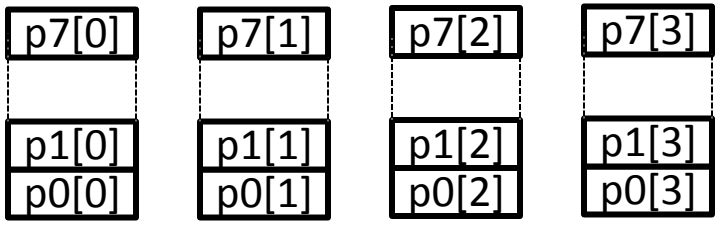


Vector configuration CSR

10-15 bits

Vector length CSR

3 bits



32 bits

8 vector predicate registers, each four elements, with 1 bit per element

Questions?