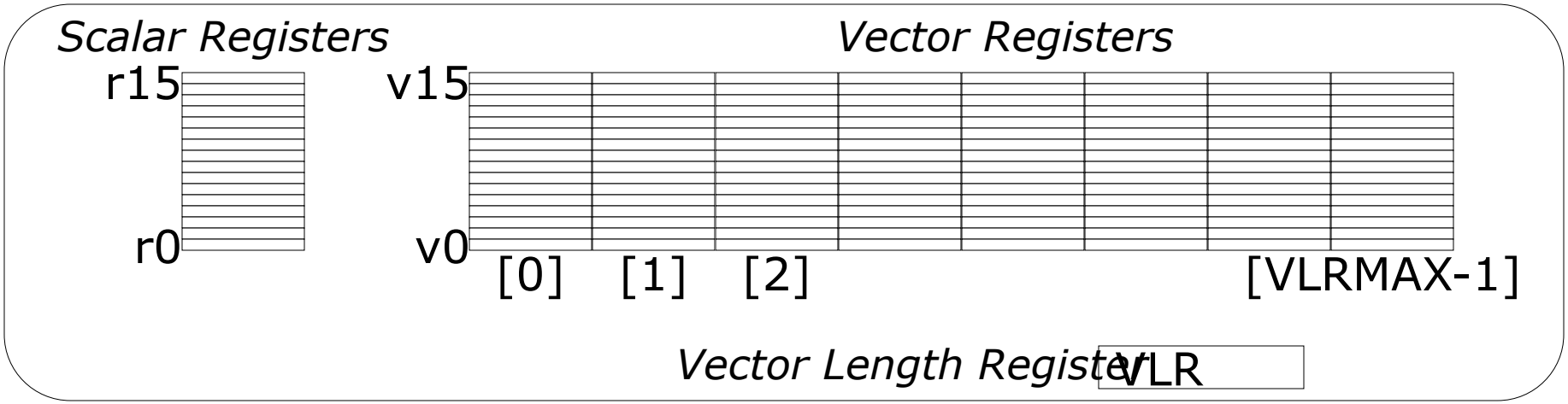
Section 10: Vectors and Lab 4

3/31/2016

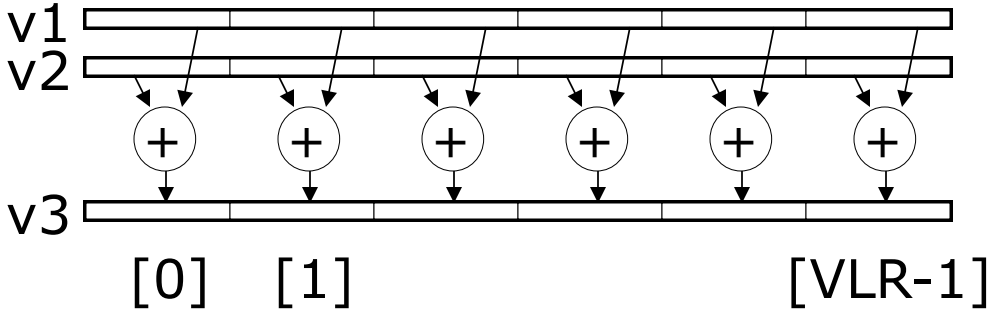
Section 10

Colin Schmidt

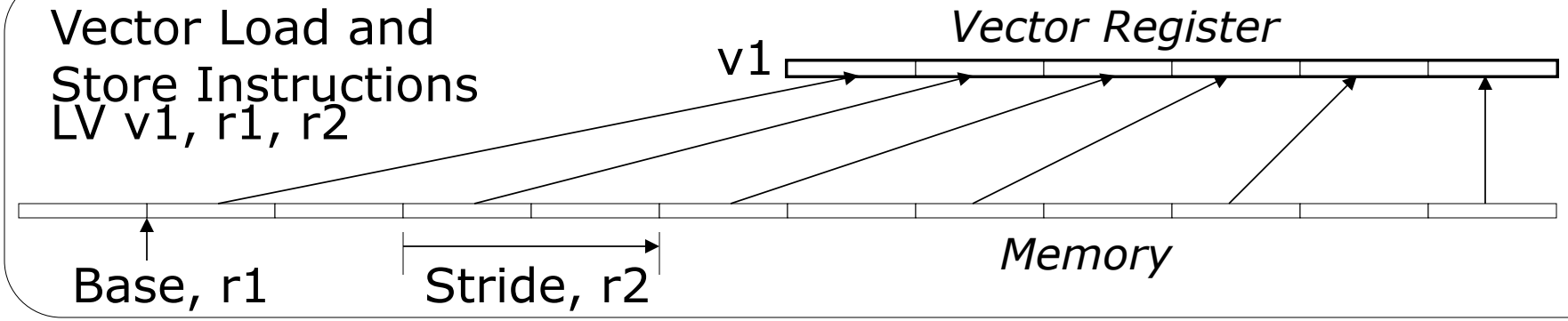
Vector Programming Model



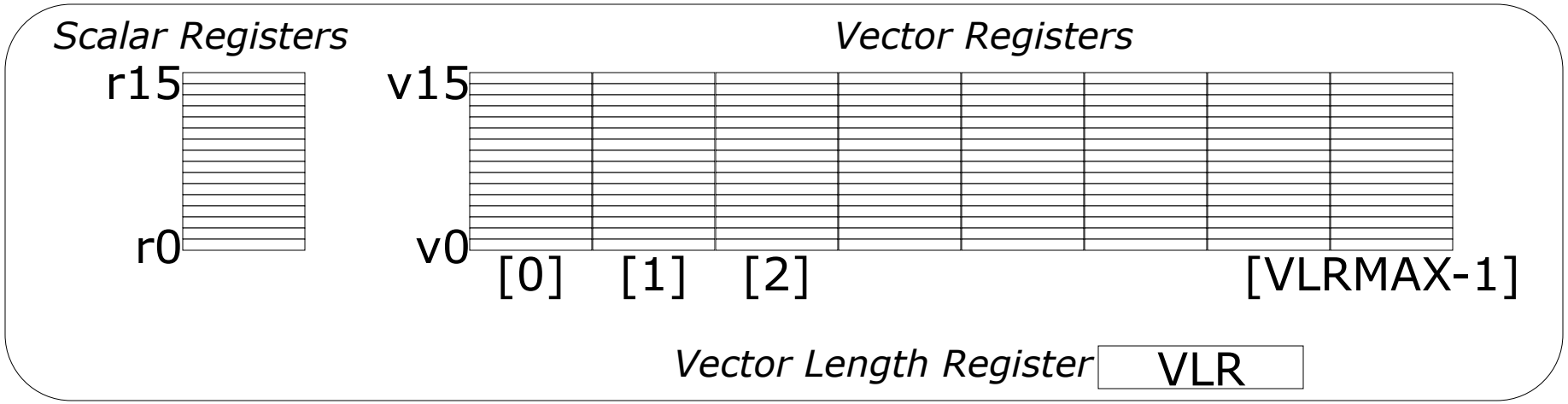
Vector Arithmetic Instructions
ADDV v3, v1, v2



Vector Load and Store Instructions
LV v1, r1, r2

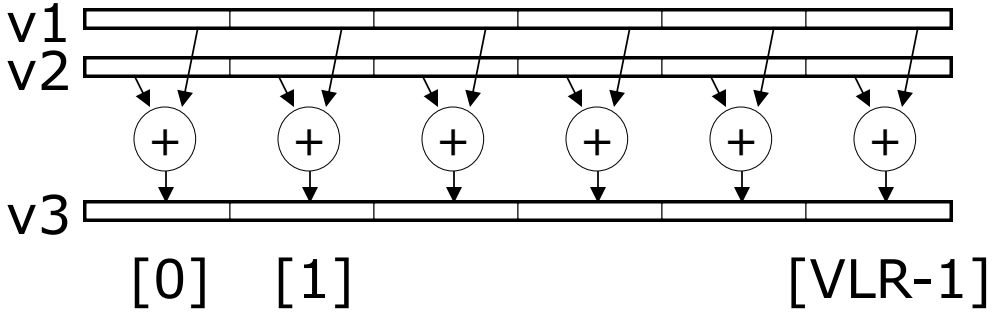


Vector Programming Model



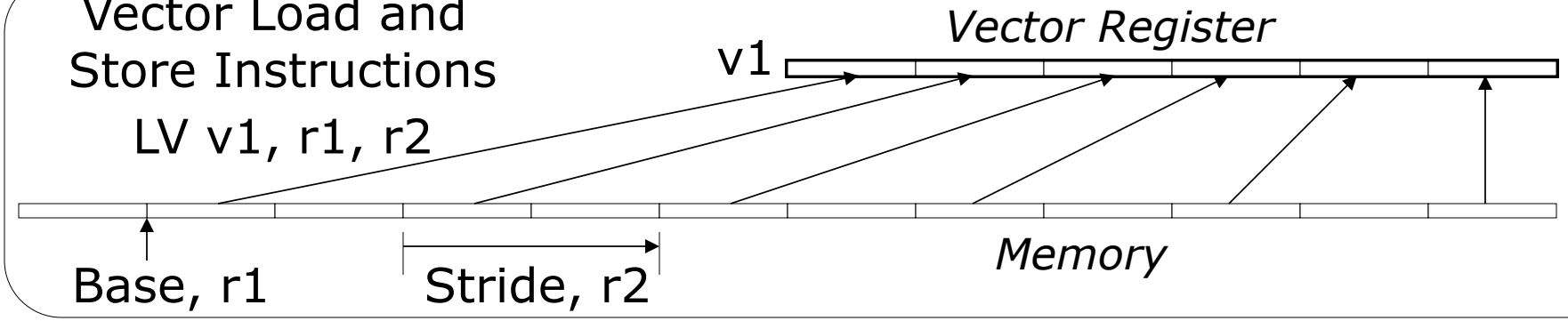
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

LV v1, r1, r2



Vector Code Example

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

Flynn's Taxonomy

- Single instruction, single data (SISD)
  - E.g., our in-order processor
- Single instruction, multiple data (SIMD)
  - Multiple processing elements, same operation, different data
  - Vector
  - Multiple processing units execute the same instruction on different data in a lockstep. Either all complete or none do. Therefore, all units have to execute the same instruction at a given time
- Multiple instruction, multiple data (MIMD)
  - Multiple autonomous processors executing different instructions on different data
  - Most common and general parallel machine
- Multiple instruction, single data (MISD)
  - Why would anyone do this?

More Categories

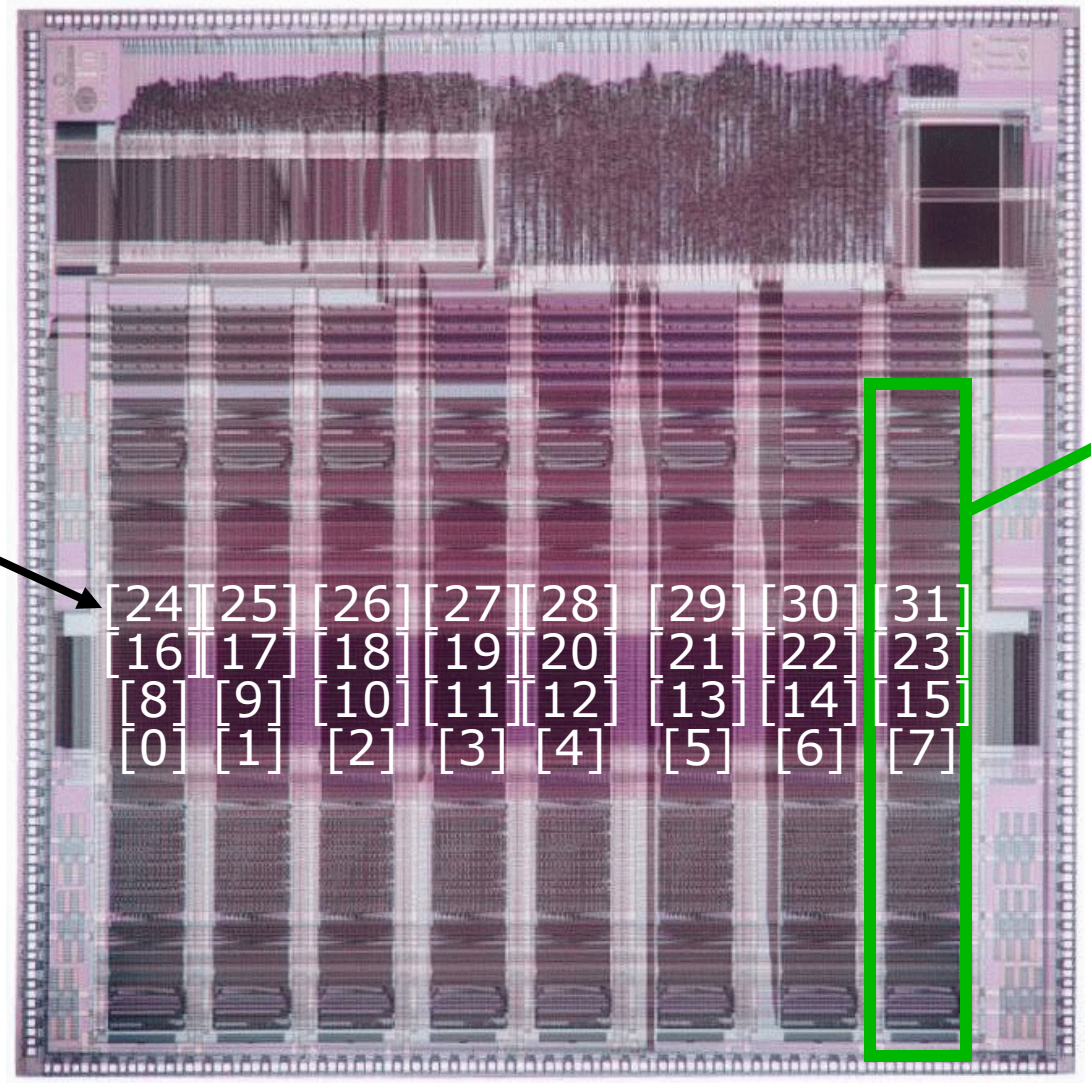
- Single program, multiple data (SPMD)
  - Multiple autonomous processors execute the program at independent points
  - Difference with SIMD: SIMD imposes a lockstep
  - Programs at SPMD can be at independent points
  - SPMD can run on general purpose processors
  - Most common method for parallel computing

- Multiple program, multiple data (MPMD)
  - Multiple autonomous processors simultaneously operating at least 2 independent programs

Vector Instruction Set Advantages

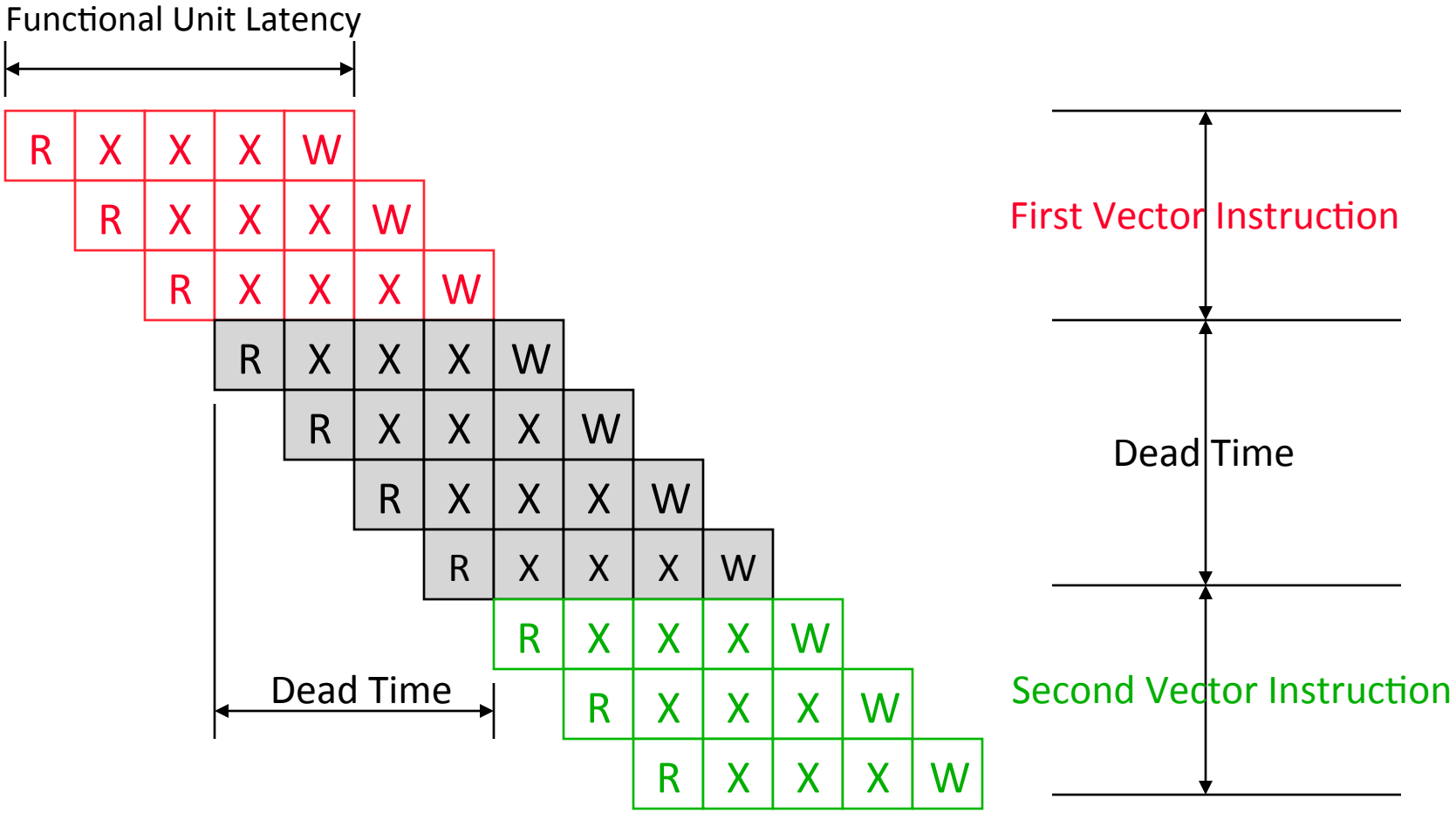
- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same code on more parallel pipelines (lanes)

T0 Vector Microprocessor (UCB/ICSI, 1995)

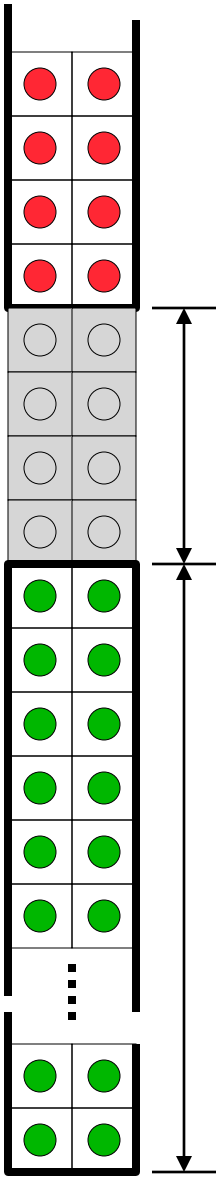


Vector Startup

- Two components of vector startup penalty
  - functional unit latency (time through pipeline)
  - dead time or recovery time (time before another vector instruction can start down pipeline). Some pipelines reduce control logic by requiring dead time between instructions to the same vector unit



Dead Time and Short Vectors

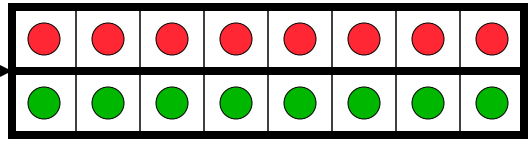


4 cycles dead time

64 cycles active

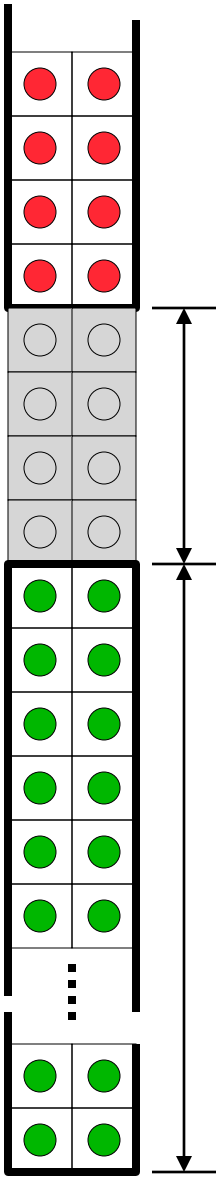
Cray C90, Two lanes
4 cycle dead time
Maximum efficiency 94%
with 128 element vectors

No dead time



T0, Eight lanes
No dead time
100% efficiency with 8 element vectors

Dead Time and Short Vectors

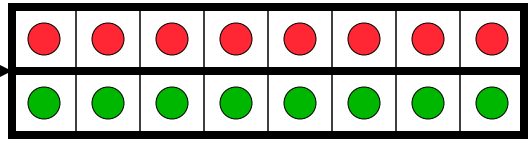


4 cycles dead time

64 cycles active

Cray C90, Two lanes
4 cycle dead time
Maximum efficiency 94%
with 128 element vectors

No dead time



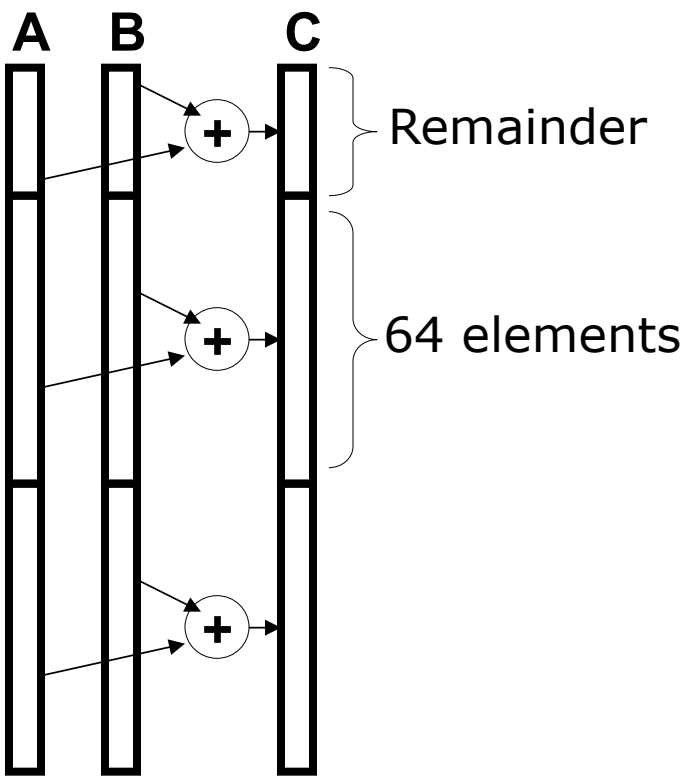
T0, Eight lanes
No dead time
100% efficiency with 8 element vectors

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, *"Stripmining"*

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```




```
ANDI R1, N, 63    # N mod 64
 MTC1 VLR, R1     # Do remainder
loop:
 LV V1, RA
 DSLL R2, R1, 3   # Multiply by 8
 DADDU RA, RA, R2 # Bump pointer
 LV V2, RB
 DADDU RB, RB, R2
 ADDV.D V3, V1, V2
 SV V3, RC
 DADDU RC, RC, R2
 DSUBU N, N, R1 # Subtract elements
 LI R1, 64
 MTC1 VLR, R1   # Reset full length
 BGTZ N, loop   # Any more to do?
```

Lab 4: Hwacha Vector-Fetch Machine

- Vector-Fetch Architecture
- More advanced/complicated vector programming model
  - Research into highly decoupled, efficient vector machines
  - Targeting mobile processors
    - Closest comparison is a phone GPU
- Hwacha.org
  - links to tech reports on ISA, microarch, etc.

Autovectorization Programming Model

SAXPY

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

Autovectorization Programming Model

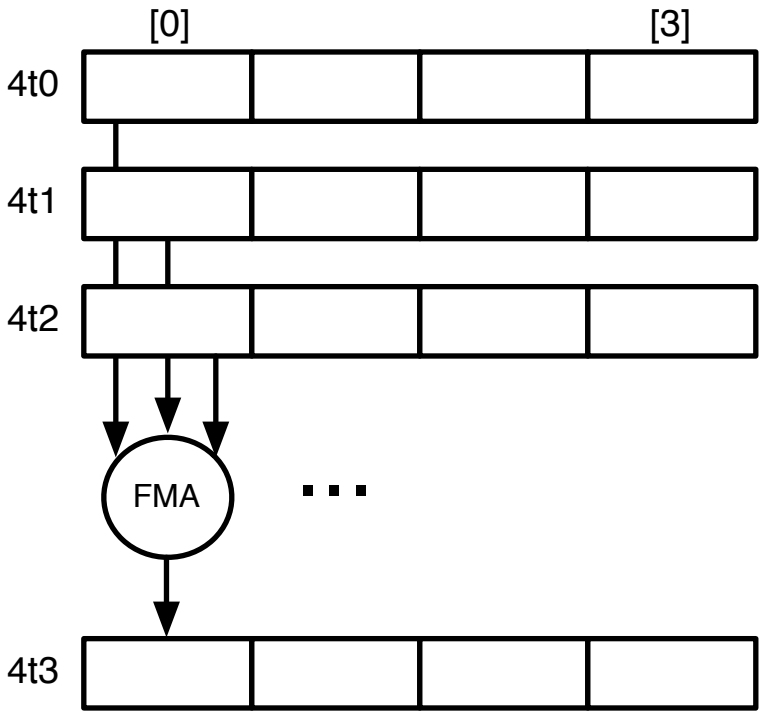
- Mostly automatic, possibly some restructuring from the application writer

SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

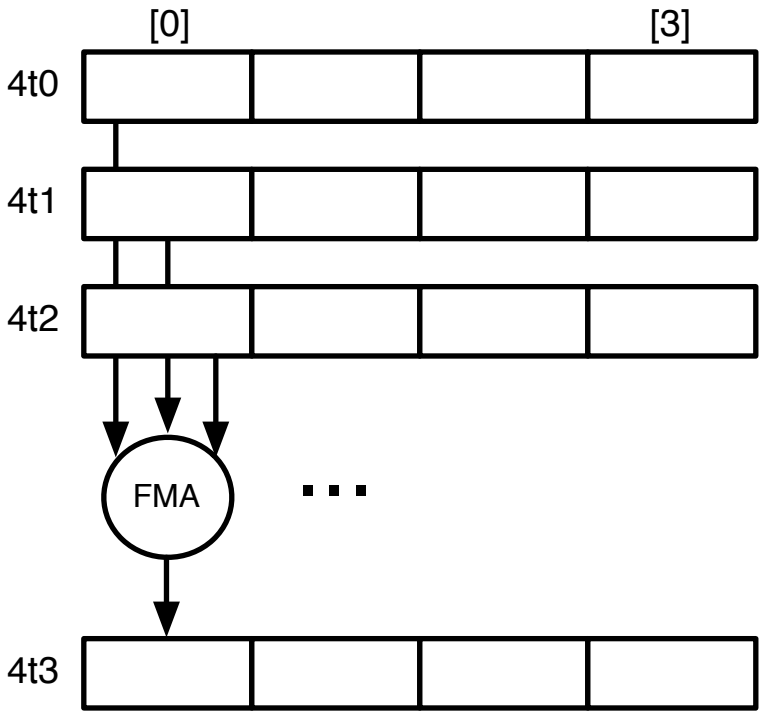


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

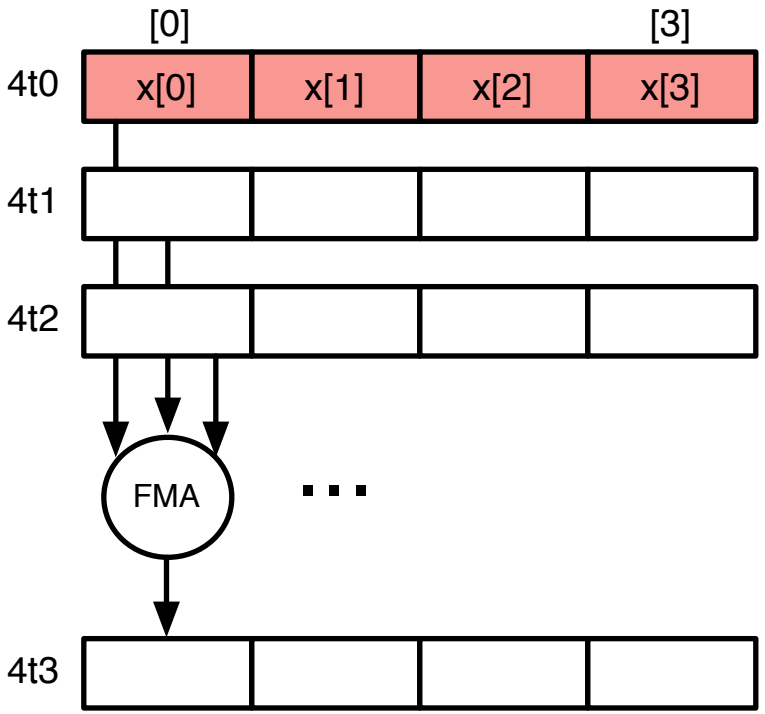


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

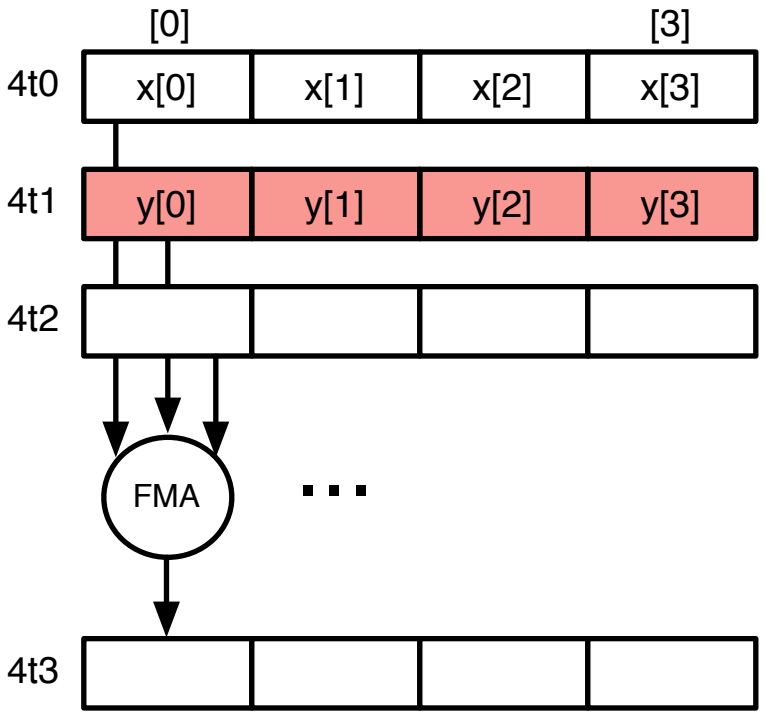


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

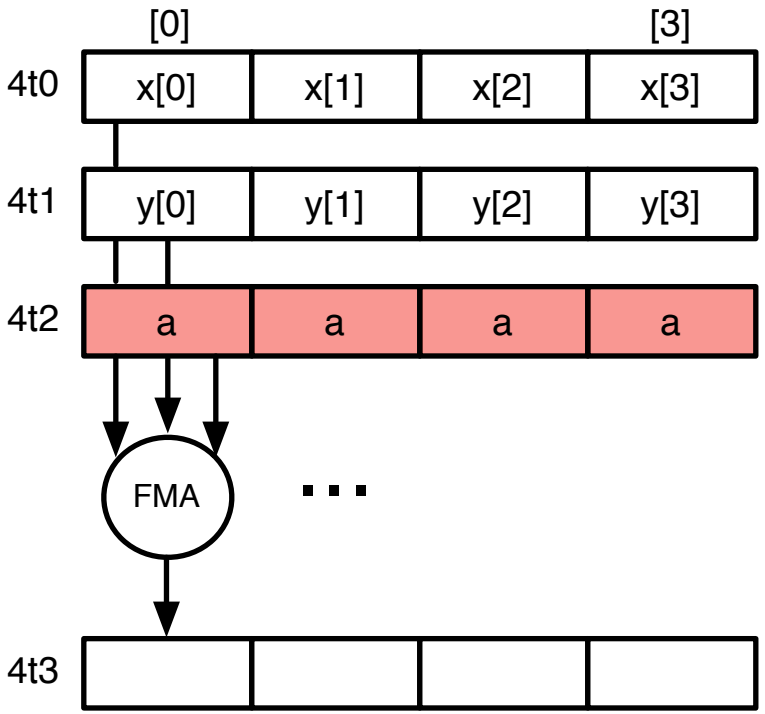


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

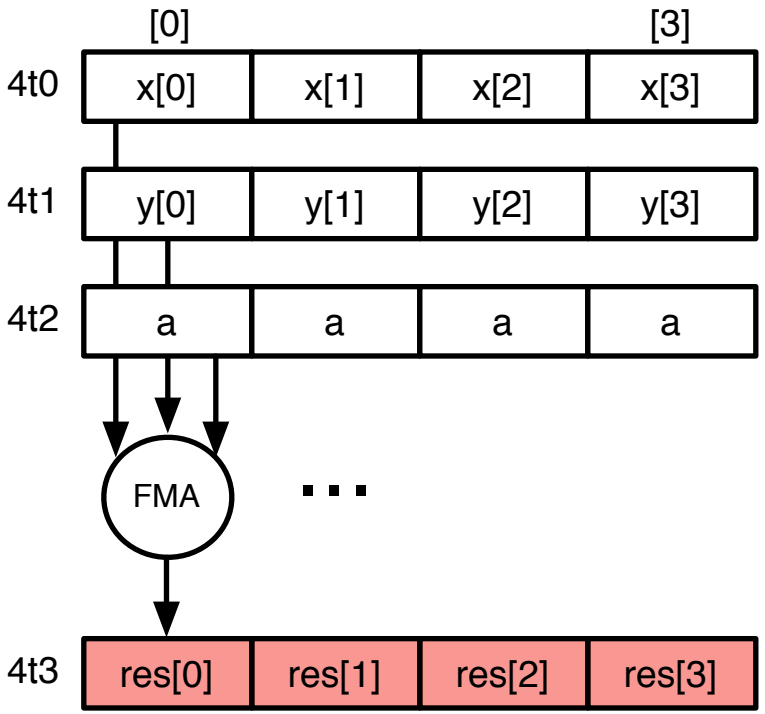


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

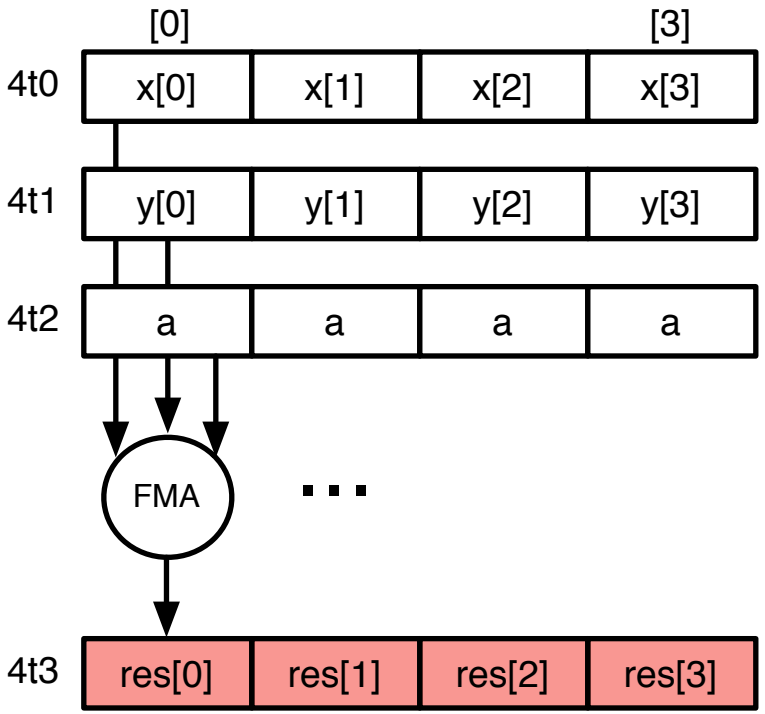


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

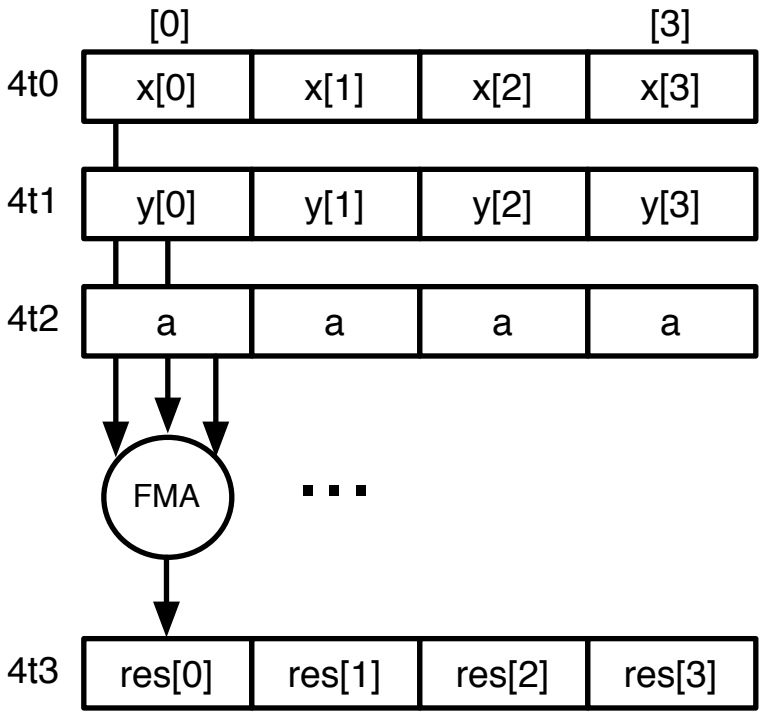


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

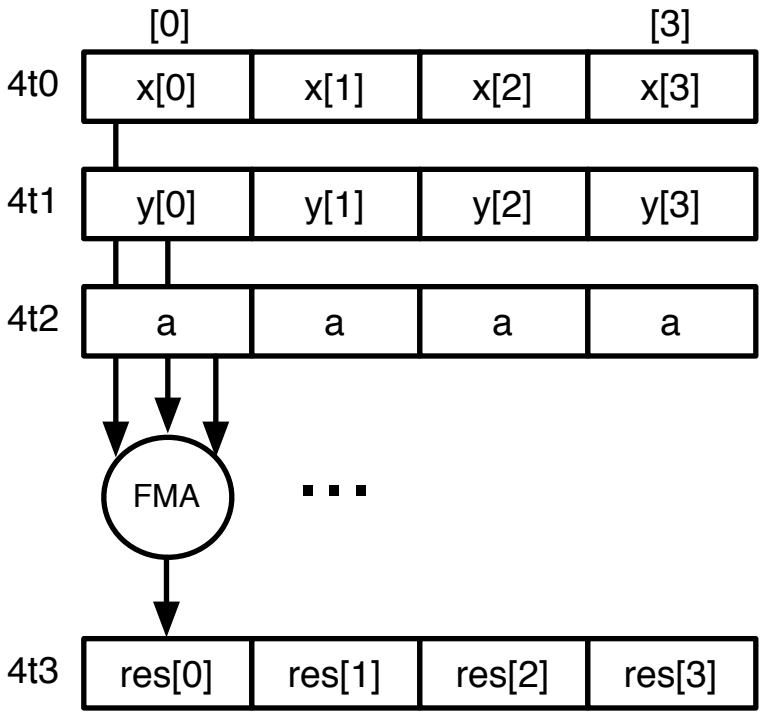


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

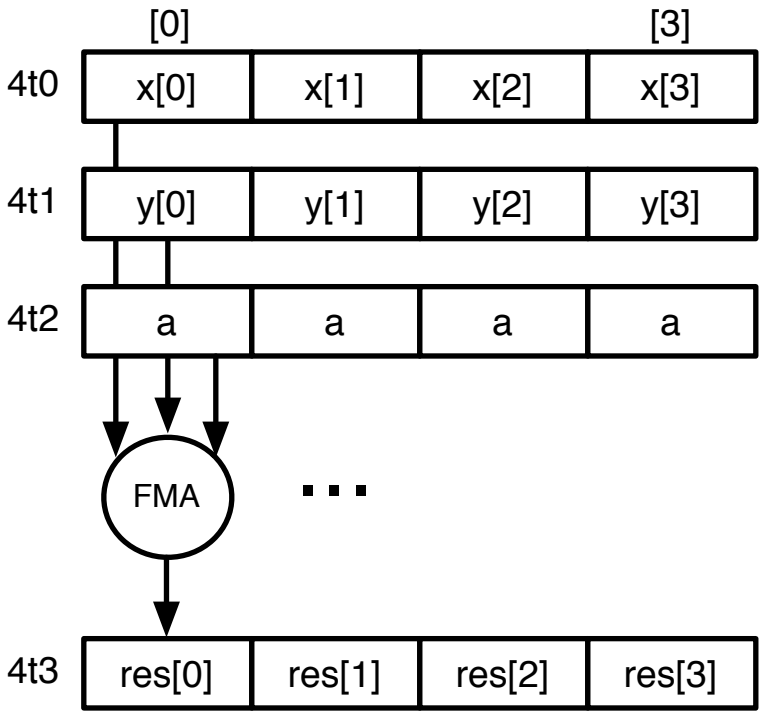


SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD



SAXPY mapped to SIMD Architecture

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vlw8 8t0, a2
  vlw8 8t1, a3
  vsplat8 8t2, a1
  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 SIMD

SIMD Arch. vs. Traditional Vector Arch.

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vector

SIMD Arch. vs. Traditional Vector Arch.

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vector

SIMD Arch. vs. Traditional Vector Arch.

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vector

SIMD Arch. vs. Traditional Vector Arch.

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vector

SIMD Arch. vs. Traditional Vector Arch.

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vector

SIMD Arch. vs. Traditional Vector Arch.

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

stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vsplat4 4t2, a1
  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
```

SIMD

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

stripmine:
  vsetvl t0, a0
  vlw vv0, a2
  vlw vv1, a3
  vfma vv1, a1, vv0, vv1
  vsw vv1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Traditional Vector

SPMD Programming Model

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

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

SPMD
Programming Model

- Classic model brought to forefront again by CUDA/OpenCL
- Same restructuring as autovectorization, more on top to get performance

SPMD Programming Model

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

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

SPMD
Programming Model

- Classic model brought to forefront again by CUDA/OpenCL
- Same restructuring as autovectorization, more on top to get performance

SPMD Programming Model

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

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

SPMD
Programming Model

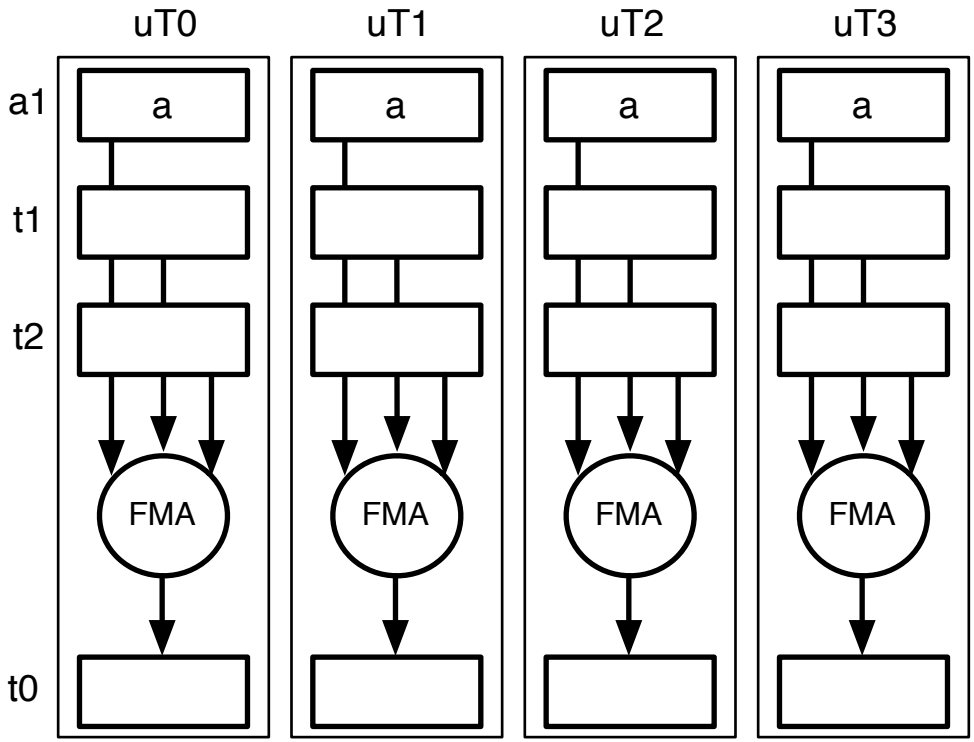
- Classic model brought to forefront again by CUDA/OpenCL
- Same restructuring as autovectorization, more on top to get performance

SIMT Architecture

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

  mv t0, tid
  bge t0, n, 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

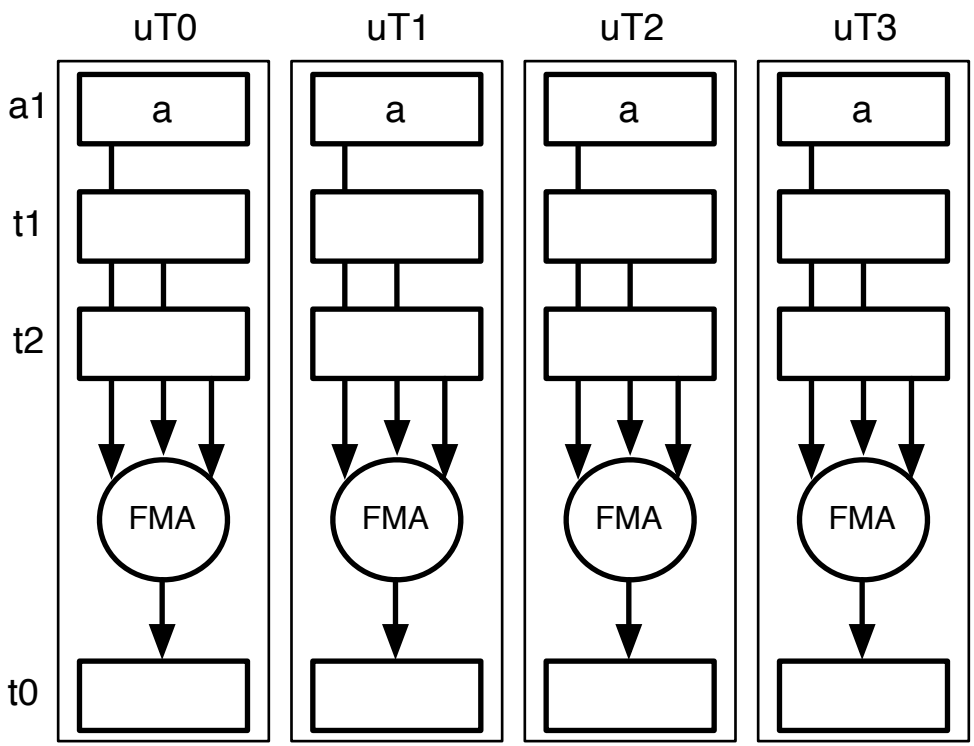


SIMT Architecture

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

  mv t0, tid
  bge t0, n, 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

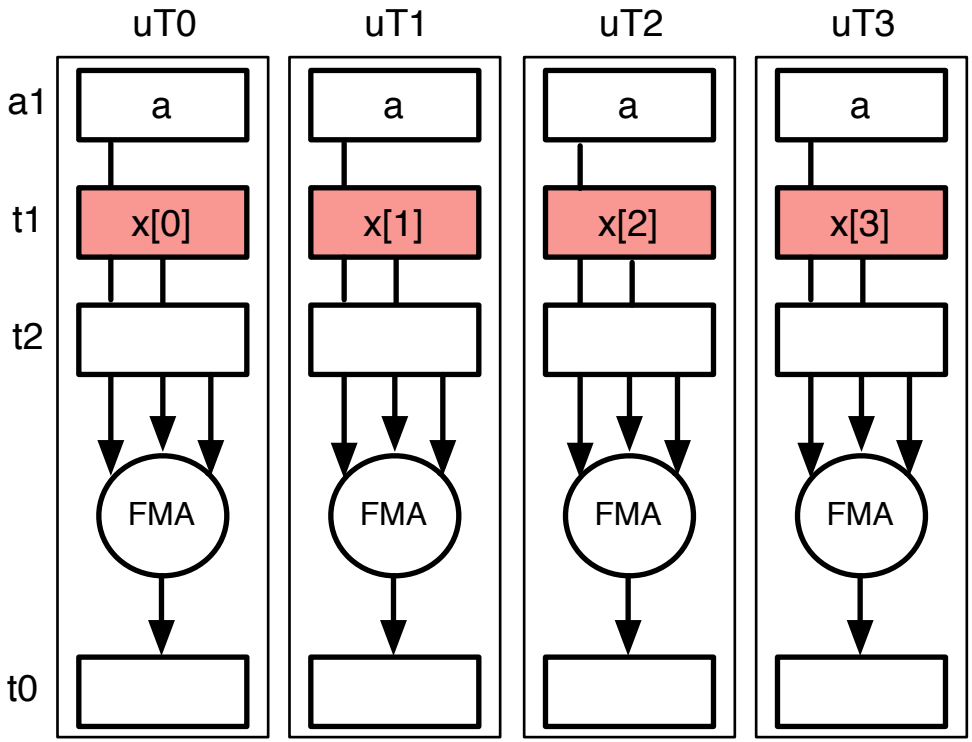


SIMT Architecture

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

  mv t0, tid
  bge t0, n, 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

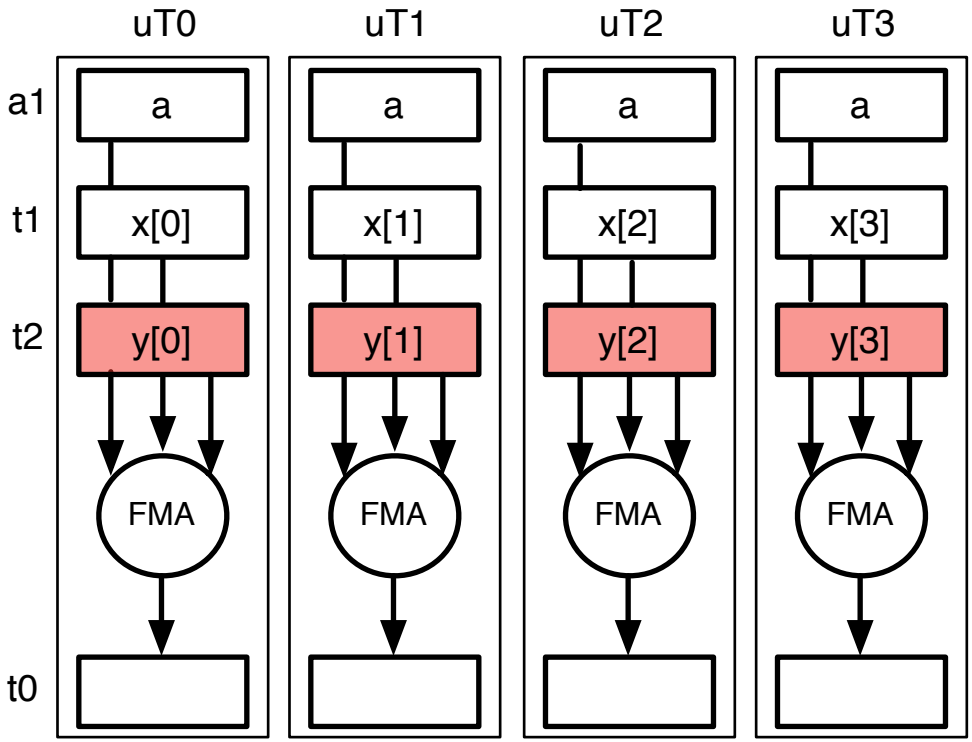


SIMT Architecture

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

  mv t0, tid
  bge t0, n, 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

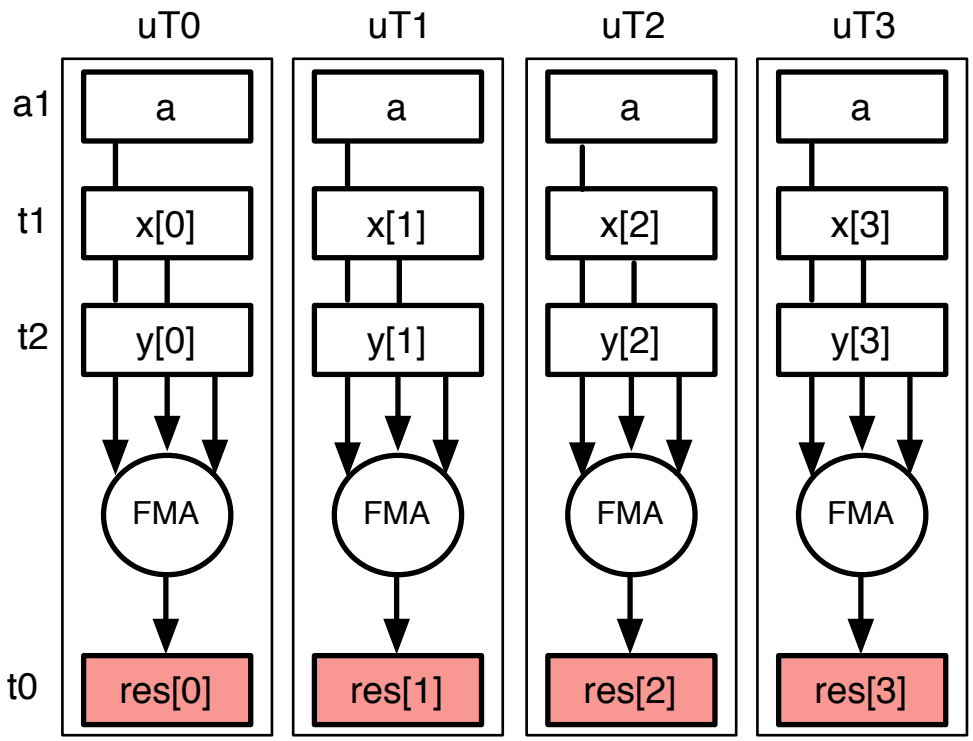


SIMT Architecture

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

  mv t0, tid
  bge t0, n, 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

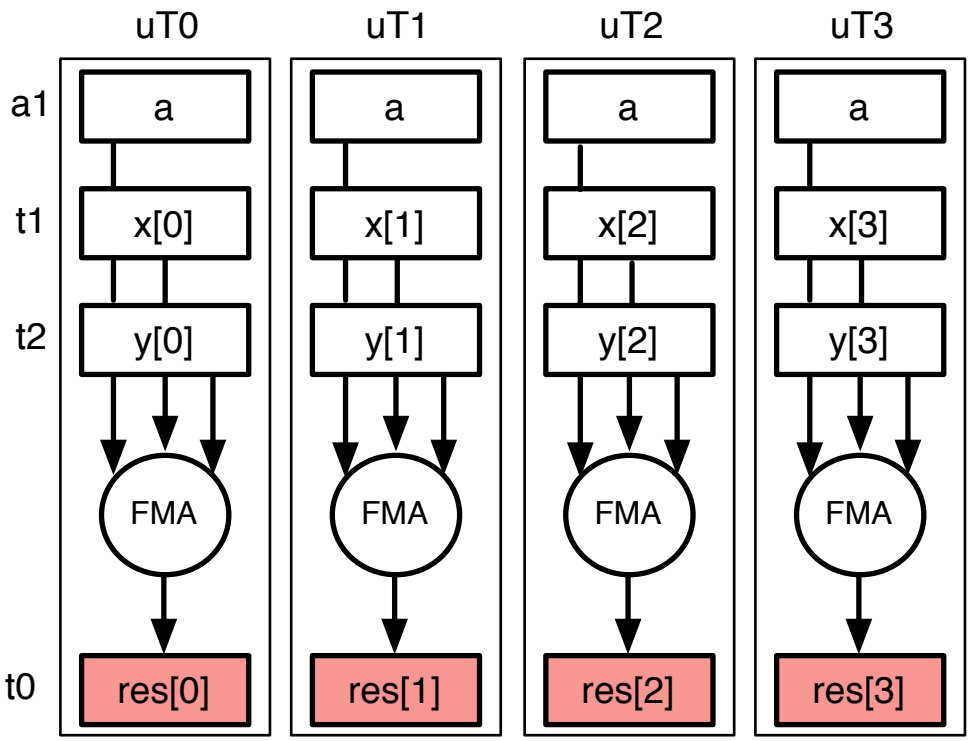


SIMT Architecture

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

  mv t0, tid
  bge t0, n, 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



SIMT Arch. vs. Traditional Vector Arch.

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

  mv t0, tid
  bge t0, n, 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

SIMT Arch. vs. Traditional Vector Arch.

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

  mv t0, tid
  bge t0, n, 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

SIMT Arch. vs. Traditional Vector Arch.

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

  mv t0, tid
  bge t0, n, 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

SIMT Arch. vs. Traditional Vector Arch.

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

  mv t0, tid
  bge t0, n, 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

SIMT Arch. vs. Traditional Vector Arch.

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

  mv t0, tid
  bge t0, n, 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

But how could we do vectors better?

Hwacha Summary

- Push in-order vector design to efficiency limit
- Designed as a non-standard RISC-V extension to hang off Rocket
- Shared memory space with Rocket
- Designed to work with the OS with restartable exceptions
- And much more…

Hwacha Vector-Fetch Architectural Paradigm

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

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread

Hwacha Vector-Fetch Architectural Paradigm

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

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread

Hwacha Vector-Fetch Architectural Paradigm

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

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread

Hwacha Vector-Fetch Architectural Paradigm

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

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread

Hwacha Vector-Fetch Architectural Paradigm

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

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
      vlw vv0, va0
      vlw vv1, va1
      vfcmpeq vp0, vv0, vv1
@vp0  vfma.svv vv1, vs0, vv0, vv1
      vsw vv1, va1
      vstop
```

Worker Thread

Hwacha Vector-Fetch Architectural Paradigm

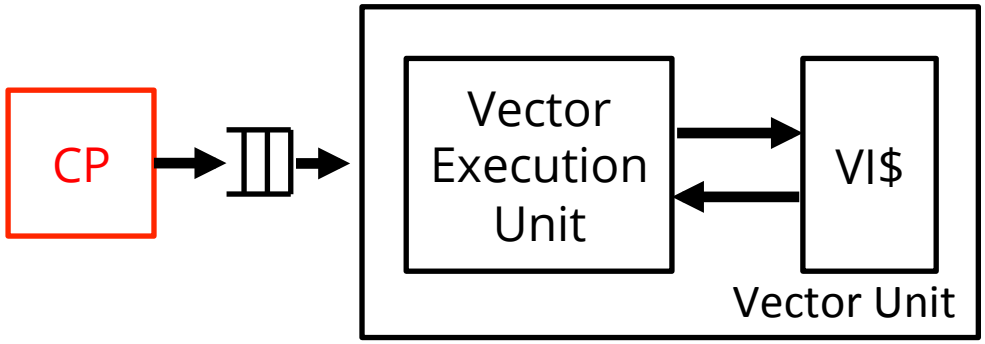
```
a0: n, a1: a,
a2: *x, a3: *y

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread



Hwacha Vector-Fetch Architectural Paradigm

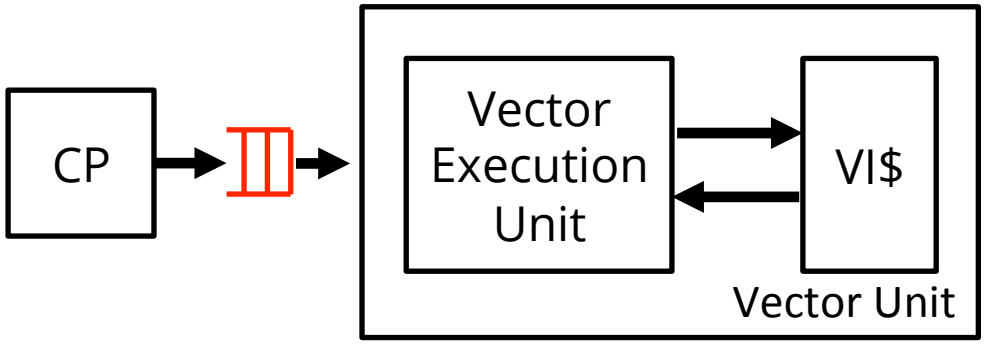
```
a0: n, a1: a,
a2: *x, a3: *y

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread



Hwacha Vector-Fetch Architectural Paradigm

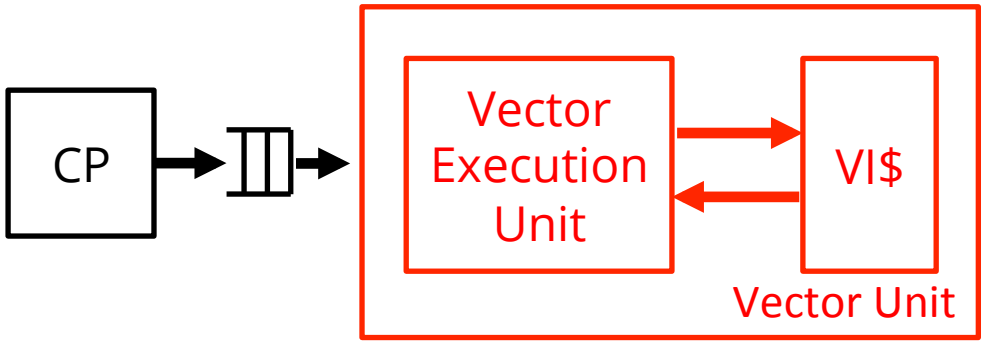
```
a0: n, a1: a,
a2: *x, a3: *y

  vmss vs0, a1
stripmine:
  vsetvl t0, a0
  vmsa va0, a2
  vmsa va1, a3
  vf saxpy
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```

Control Thread

```
saxpy:
  vlw vv0, va0
  vlw vv1, va1
  vfma.svv vv1, vs0, vv0, vv1
  vsw vv1, va1
  vstop
```

Worker Thread



Hwacha Compilers

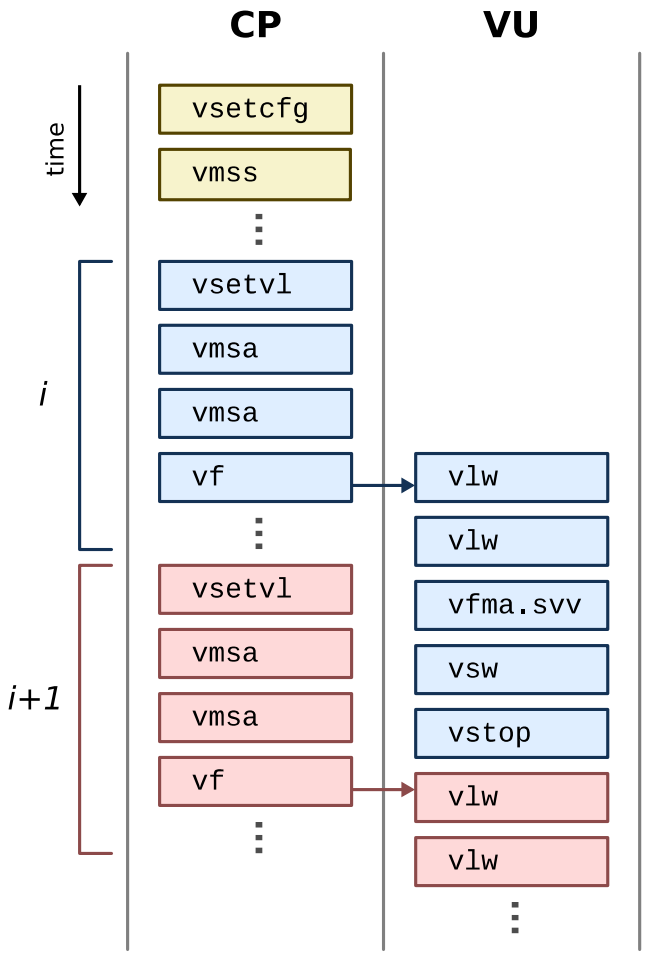
```
for (i=0; i<n; i++) {
  y[i] = a*x[i] + y[i];
}
```

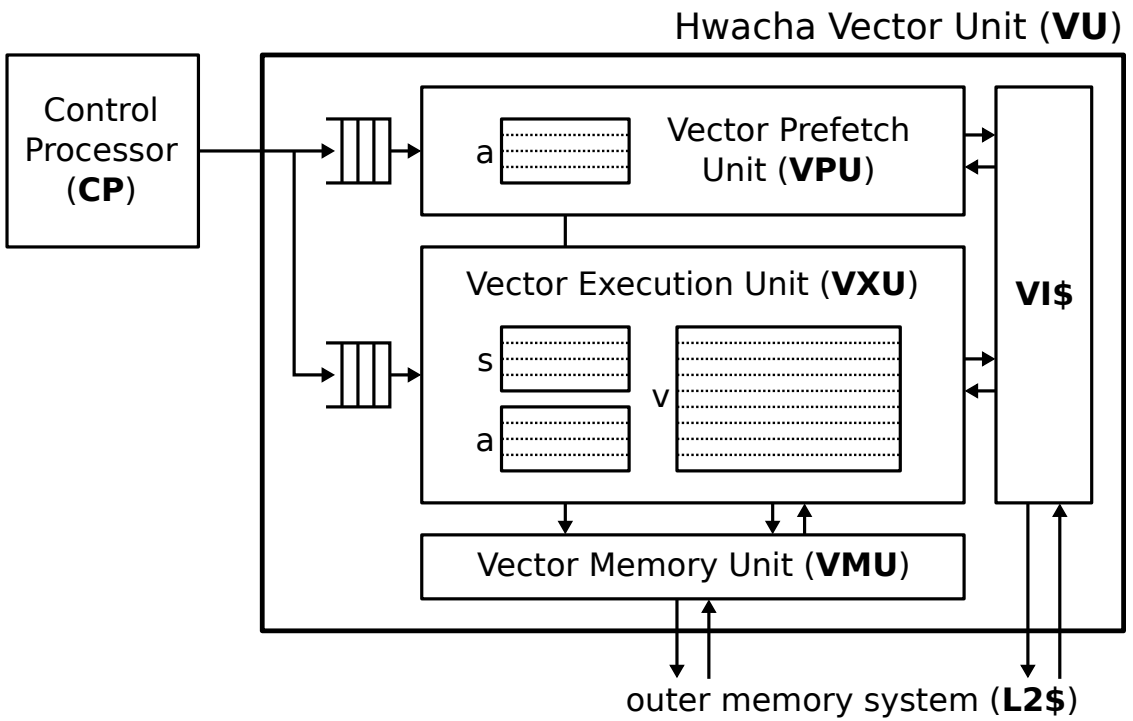
```
Kernel(int n, float a,
       float* x, float* y) {
  if (tid < n) {
    y[tid] = a*x[tid]+y[tid];
  }
}

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

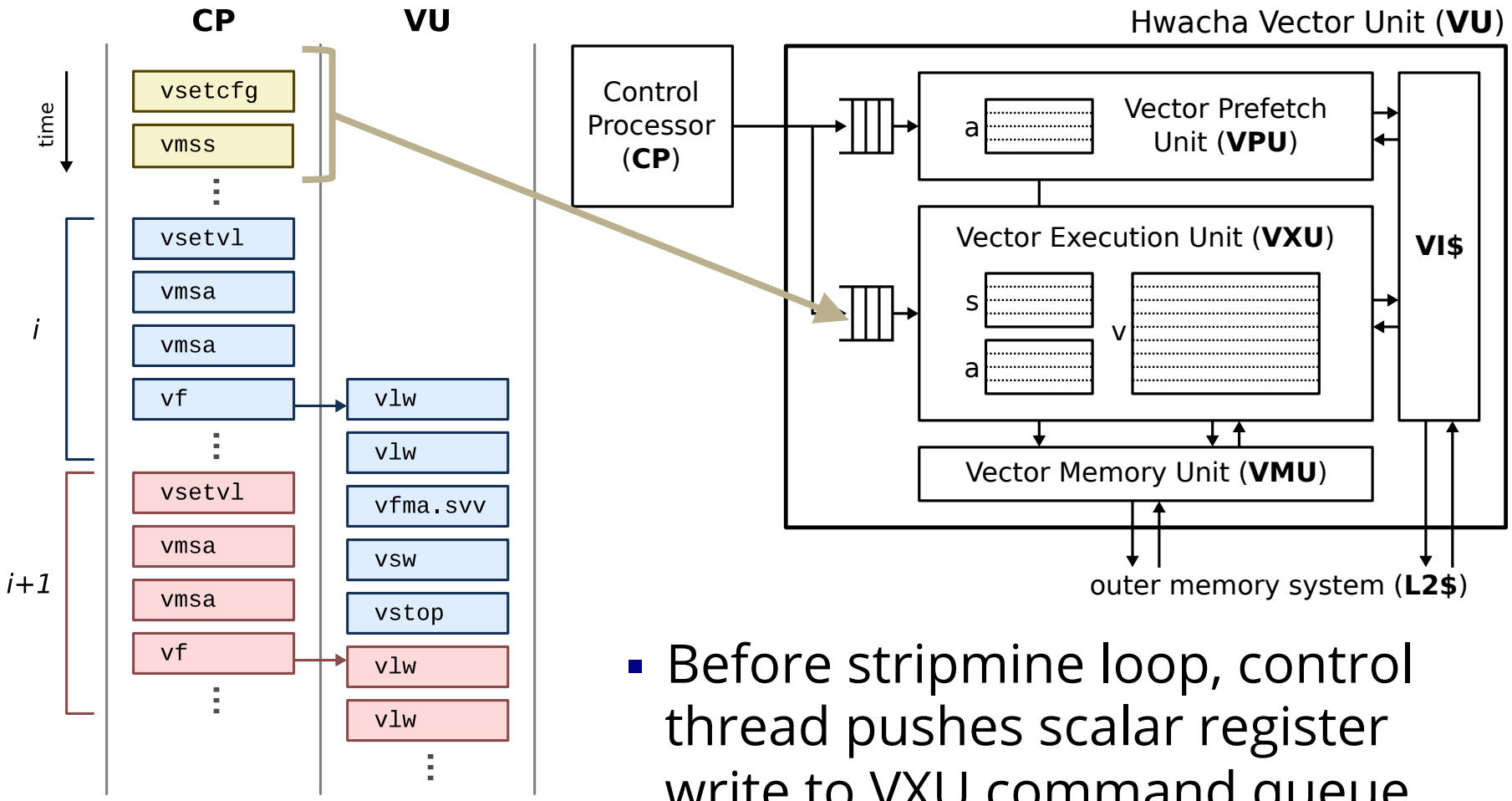
- SPMD-style programming model
  - Update our OpenCL compiler in LLVM to new ISA
  - With predication pass and scalarization pass
- Autovectorization programming model
  - Support exists in LLVM infrastructure
  - Previous autovectorization work from ParLab

Decoupling – SAXPY Example



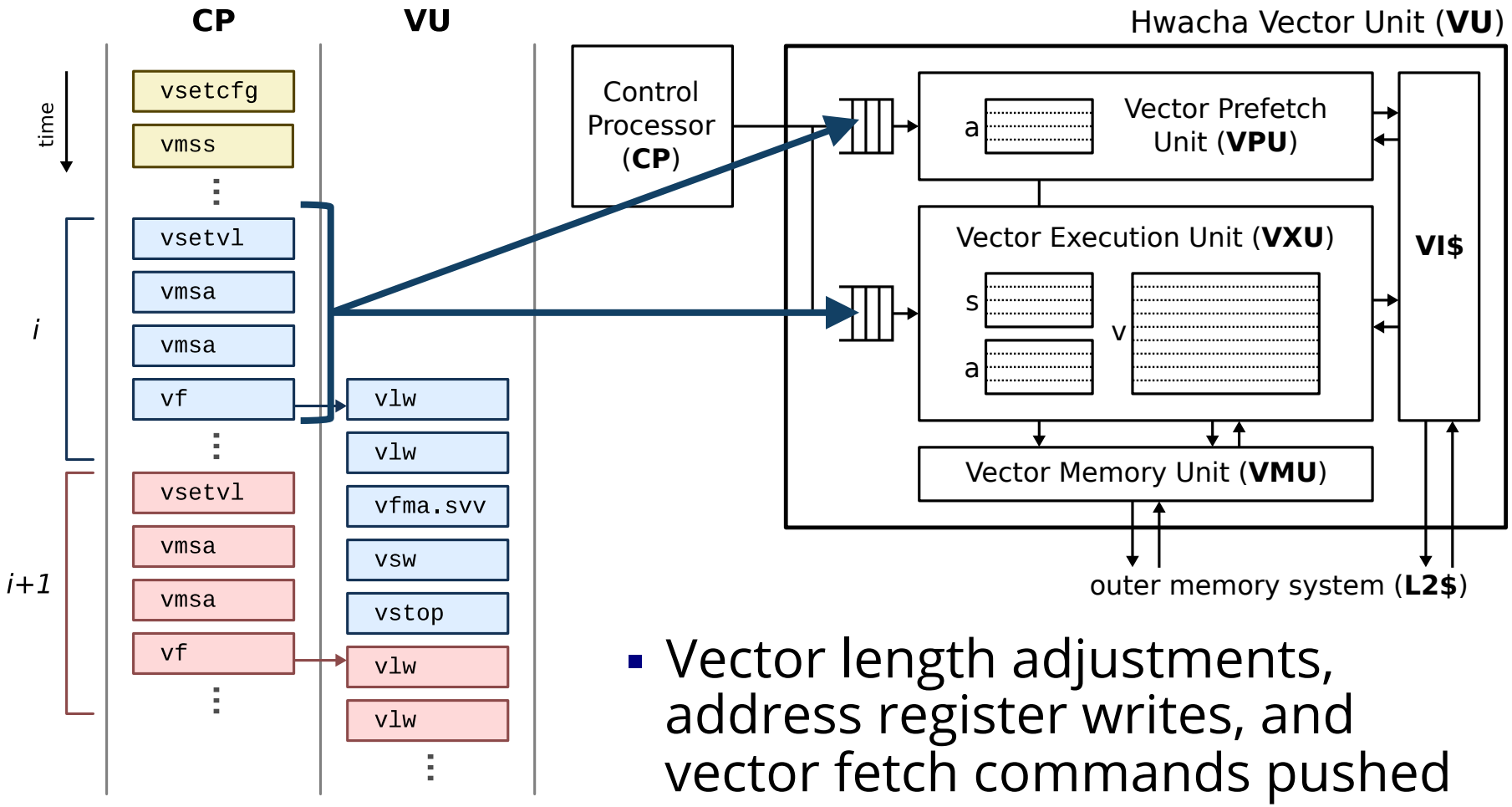


Decoupling – SAXPY Example



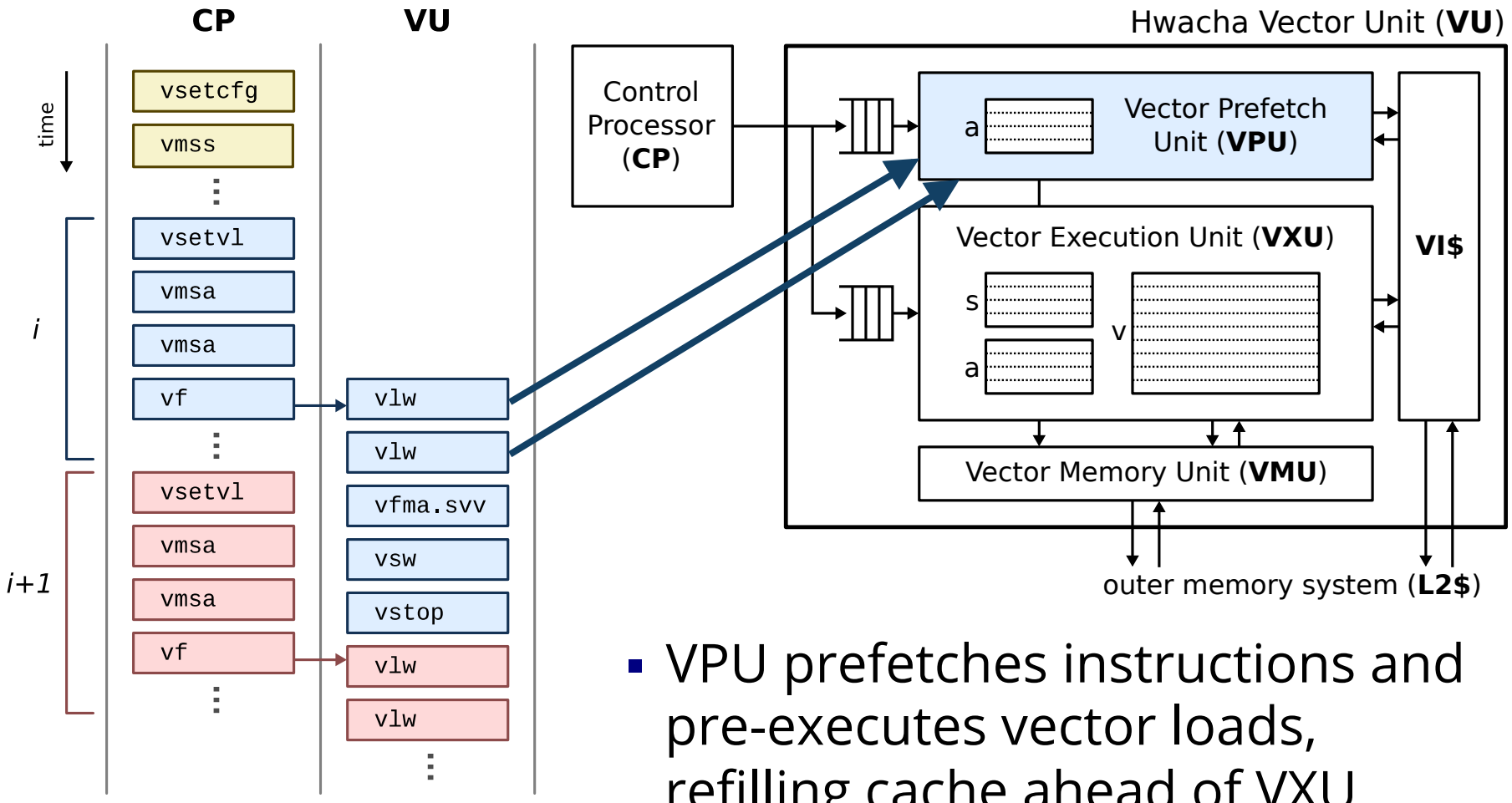
- Before stripmine loop, control thread pushes scalar register write to VXU command queue

Decoupling – SAXPY Example



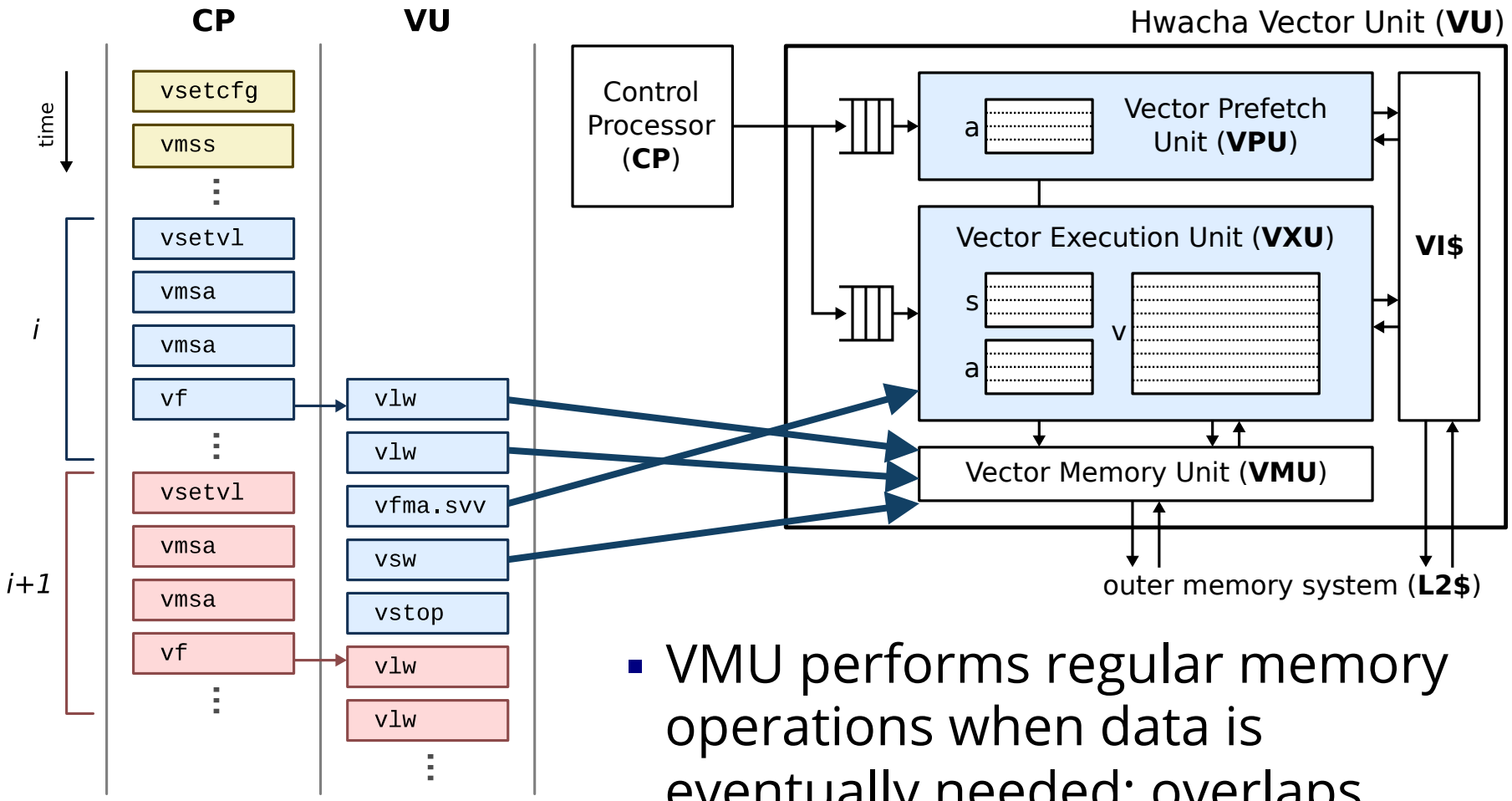
- Vector length adjustments, address register writes, and vector fetch commands pushed to both queues for each stripmine iteration

Decoupling – SAXPY Example



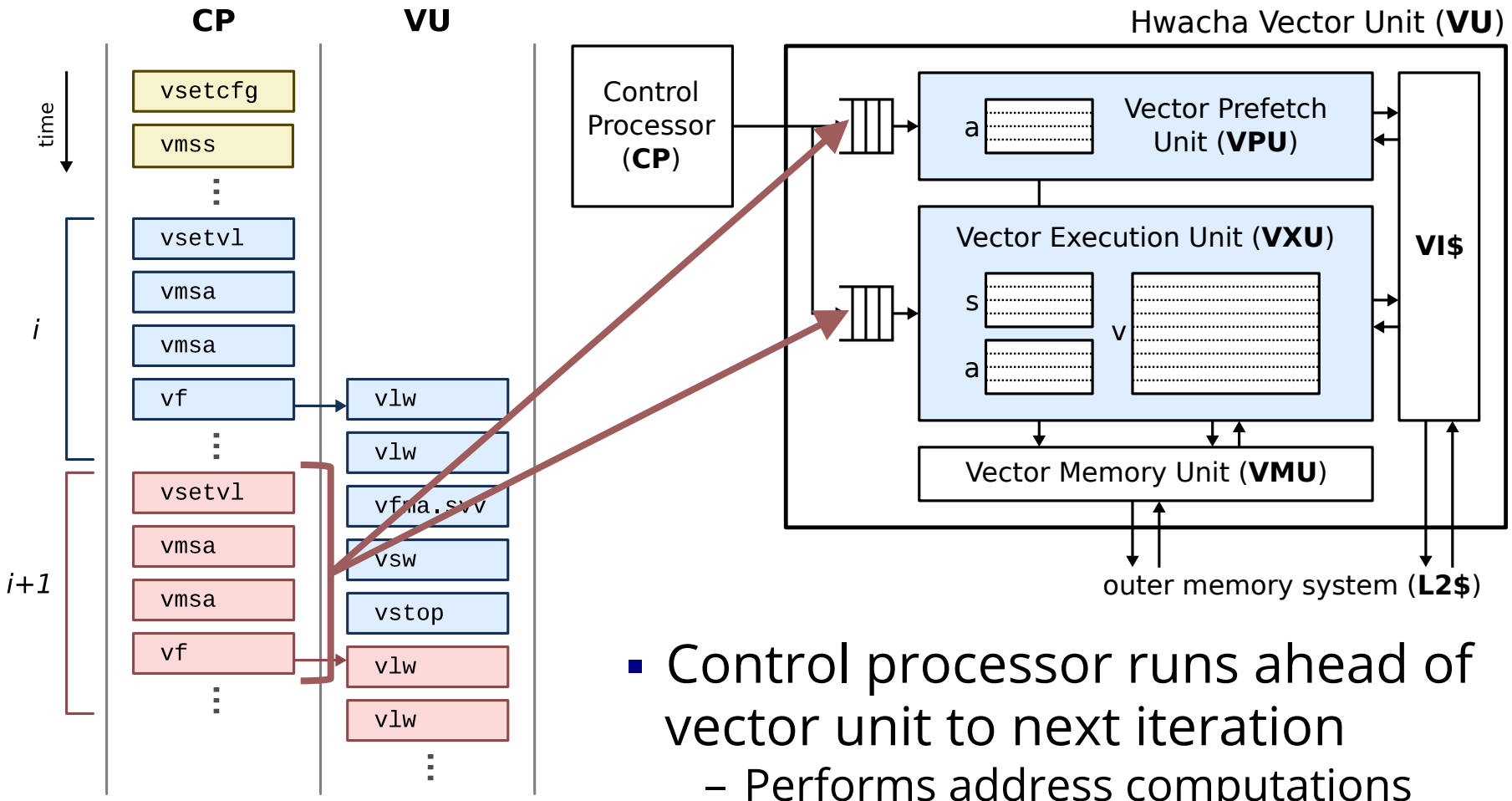
- VPU prefetches instructions and pre-executes vector loads, refilling cache ahead of VXU access

Decoupling – SAXPY Example



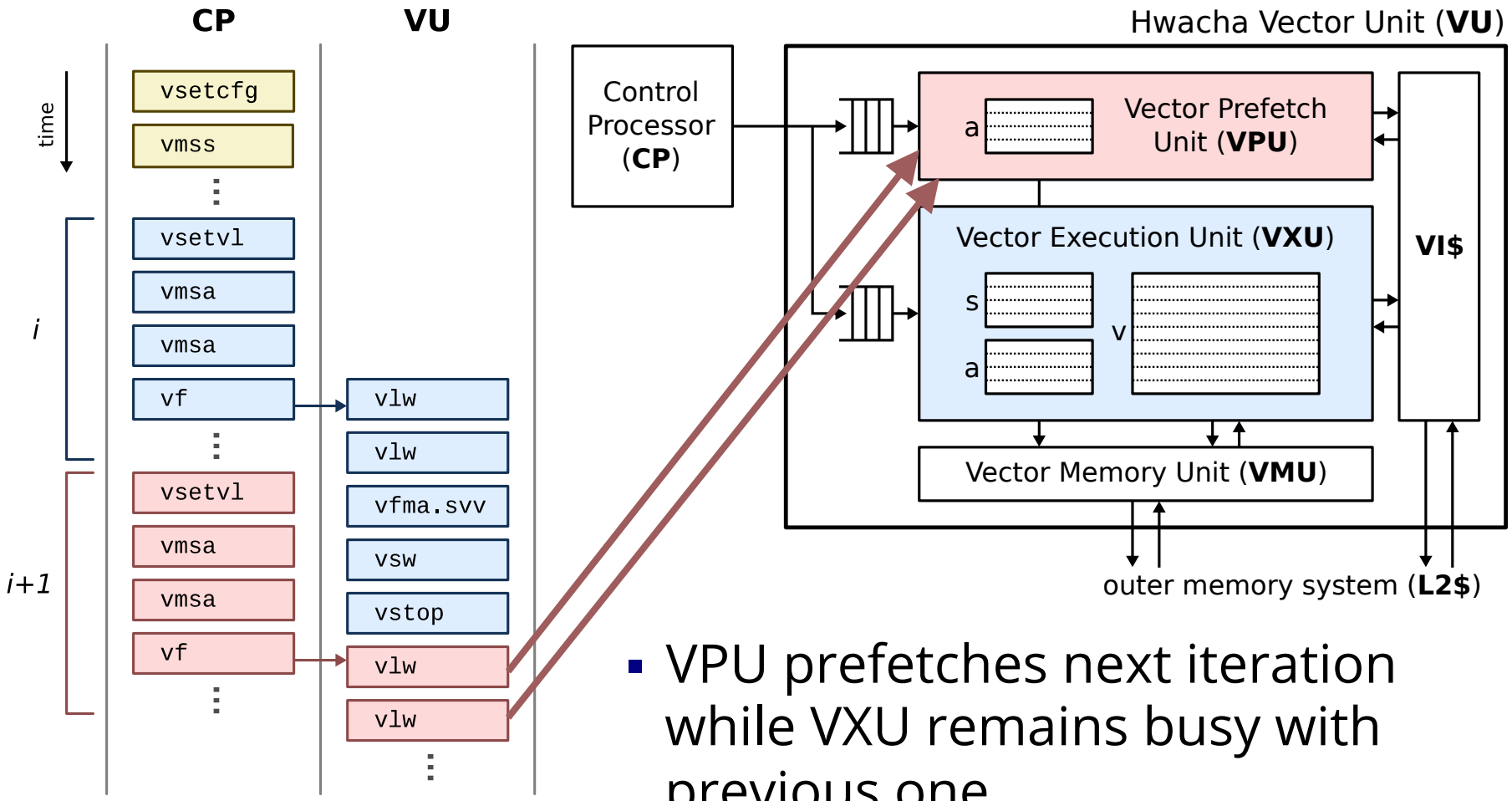
- VMU performs regular memory operations when data is eventually needed; overlaps with VXU computation

Decoupling – SAXPY Example



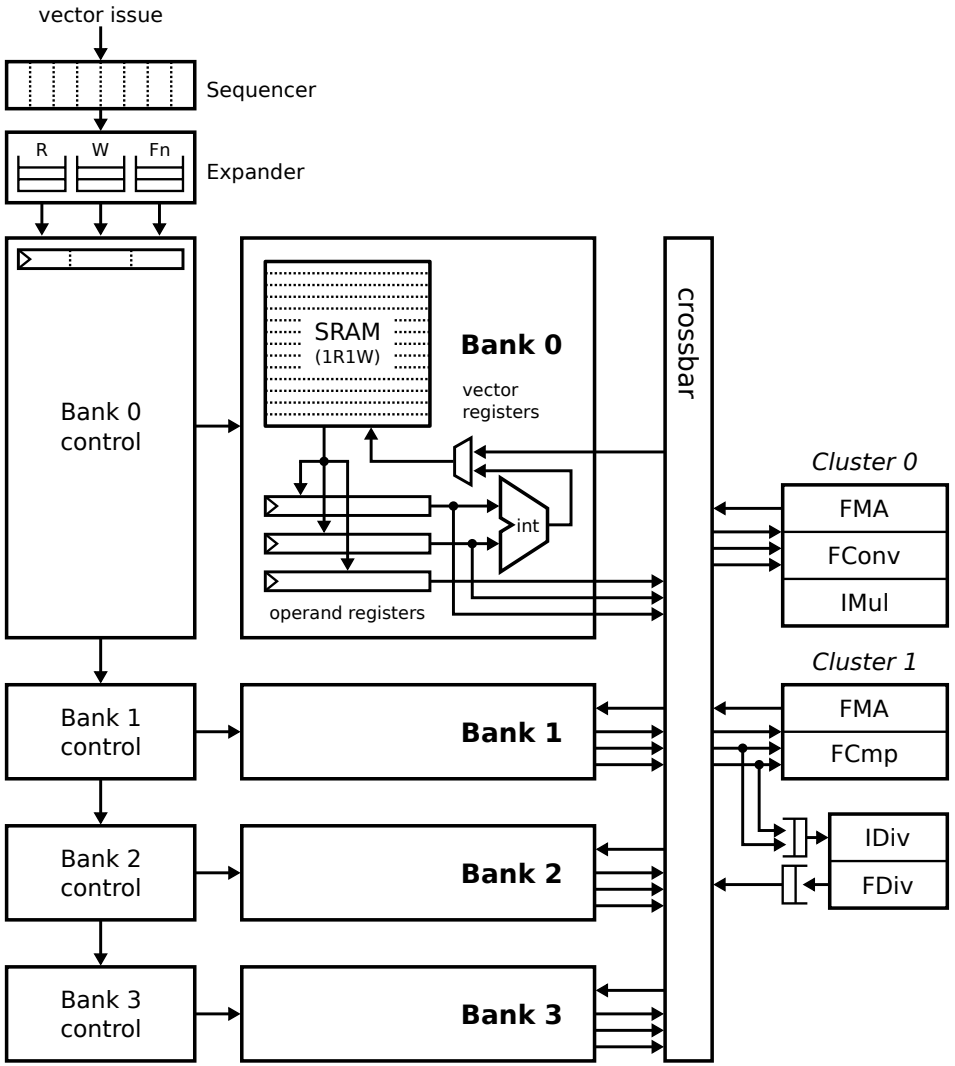
- Control processor runs ahead of vector unit to next iteration
  - Performs address computations
  - Pushes next vector fetch command

Decoupling – SAXPY Example



- VPU prefetches next iteration while VXU remains busy with previous one

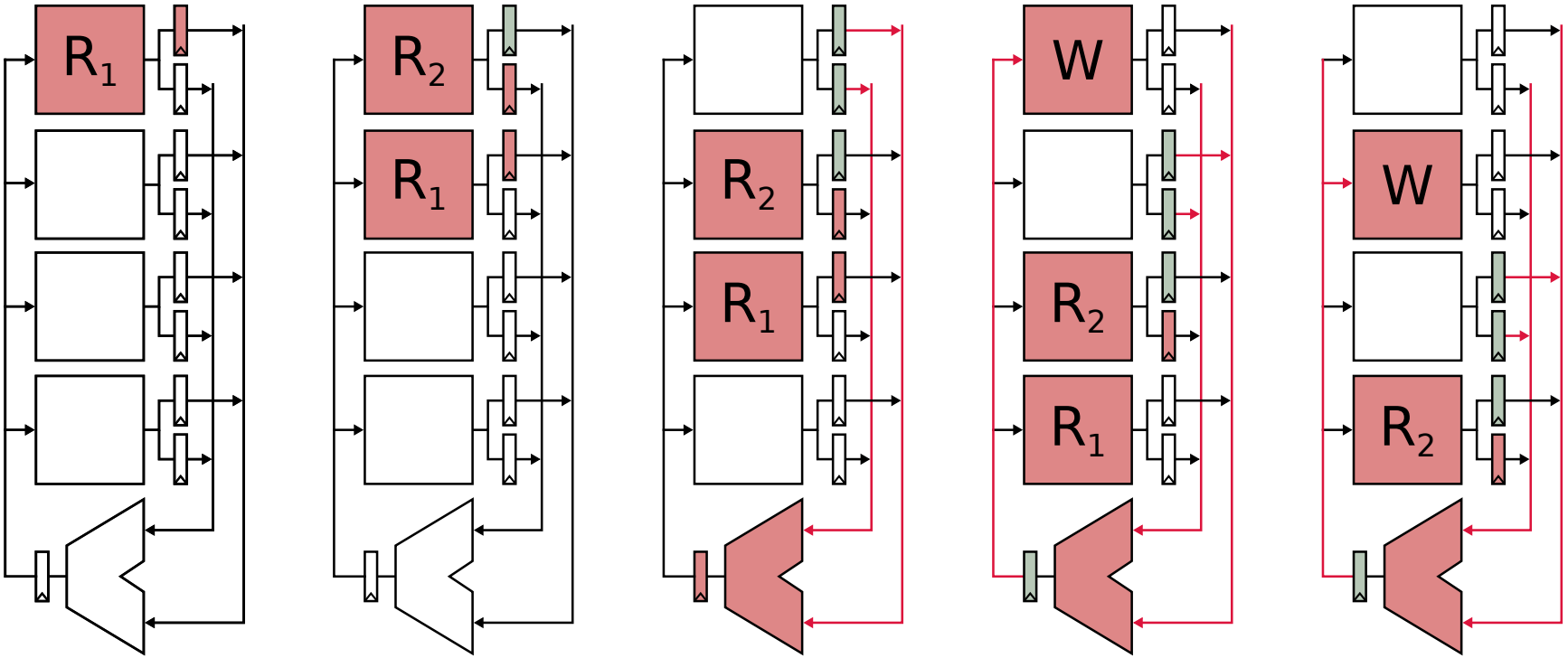
Vector Lane Organization



- 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

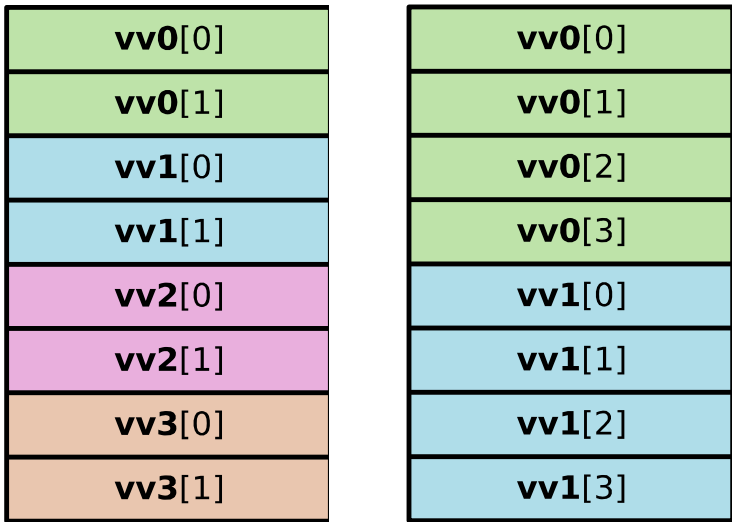
- Sustains *n* operands/cycle after *n*-cycle initial latency



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

Reconfigurable Vector Register File

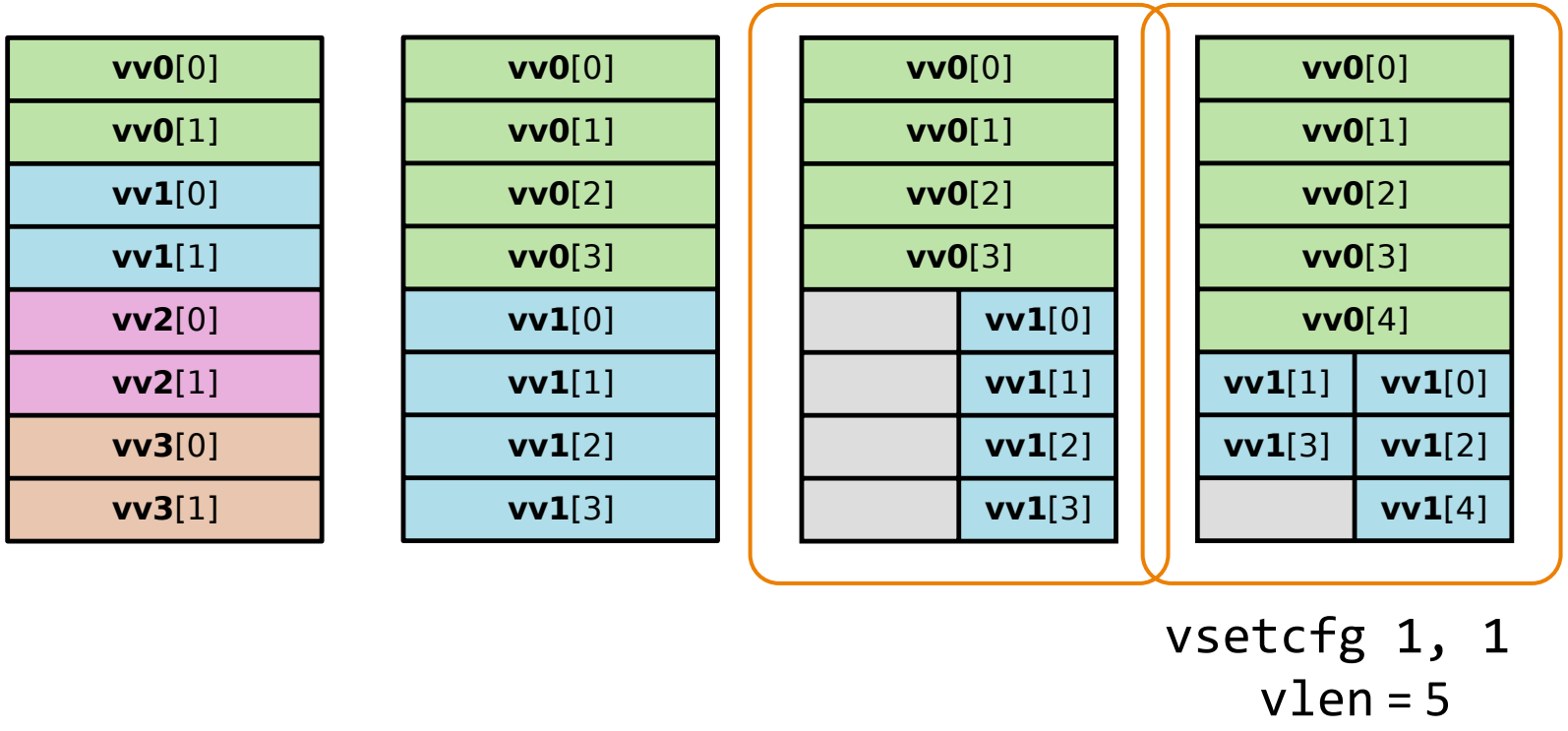
- Programming model allows specifying number of architectural registers
- Maximum hardware vector length automatically extends to fill the capacity of the register file



- Exchange unused architectural registers for longer hardware vectors

Mixed-Precision Support

- Hardware can subdivide a physical register into multiple narrower architectural registers as needed
  - Subword packing transparent to software
  - Improved utilization of operand communication bandwidth
  - Spatial functional unit parallelism




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