Cornell University
Computer Systems Laboratory

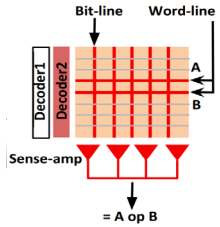
Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

Courtney Golden[1,3], Dan Ilan[2], Caroline Huang[1], Niansong Zhang[1], Zhiru Zhang[1], and Christopher Batten[1]

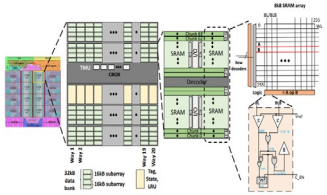
[1]Cornell University, [2]GSI Technology, [3]Massachusetts Institute of Technology

COMPUTE-IN-SRAM

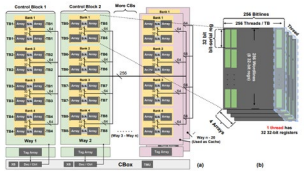
Simulation



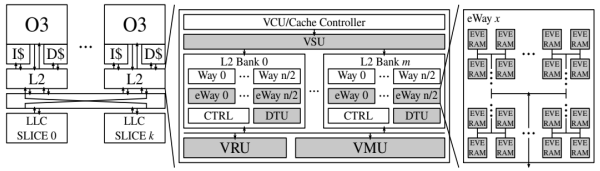
Compute Caches



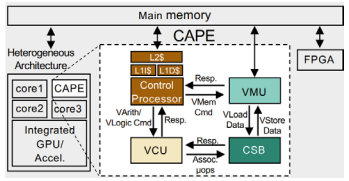
Neural Cache



Duality Cache

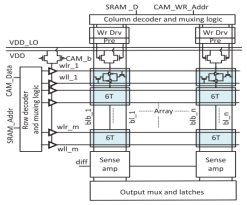


EVE

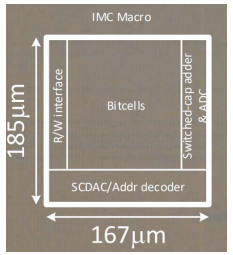


CAPE

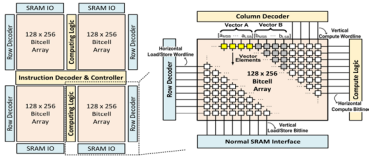
Test Chips



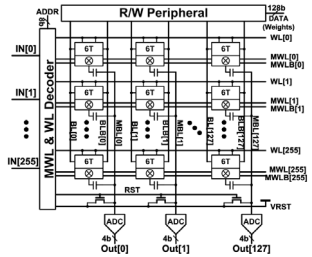
28nm 6T SRAM



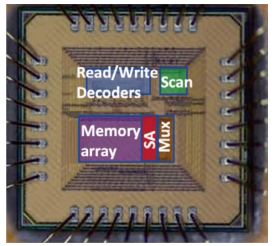
28nm DCNN



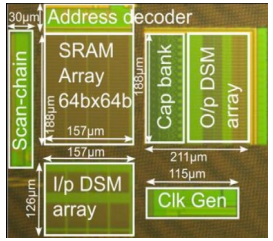
28nm Vector CRAM



28nm PIMCA



65nm PSRAM



65nm DSM

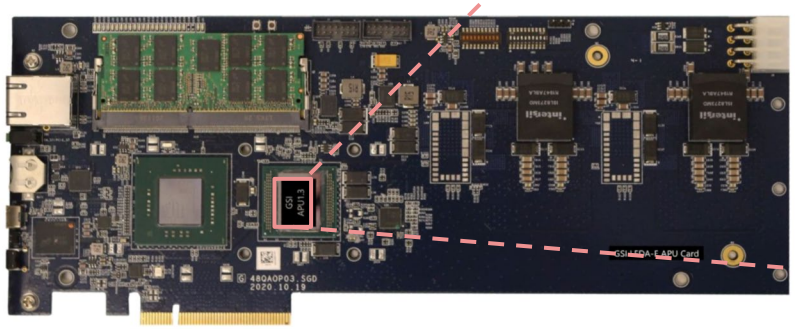
GSI Gemini Platform: Associative Processing Unit (APU)

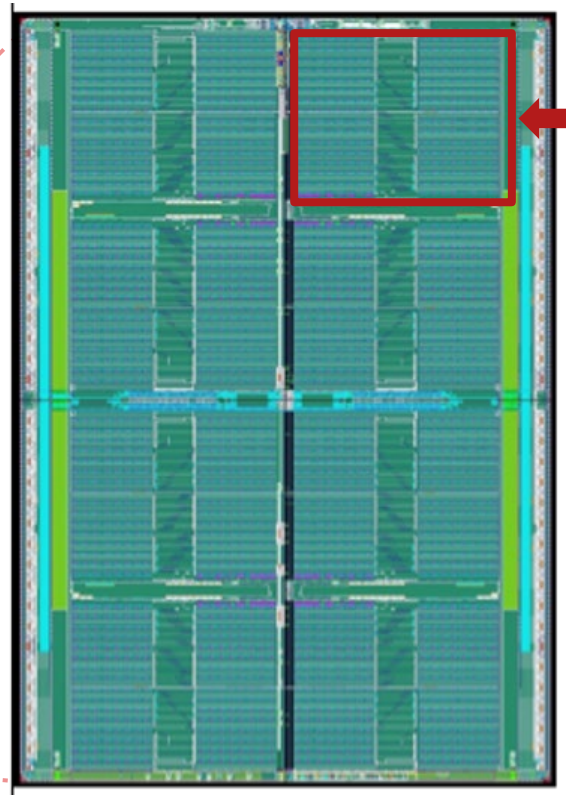
microcode

```
0xFFFF: RL = VRF[vsrc0];
0xFFFF: RL |= VRF[vsrc1];
0xFFFF: VRF[vdst] = RL;
```

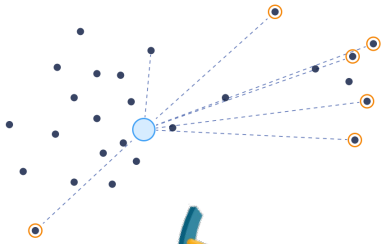
higher-level abstraction

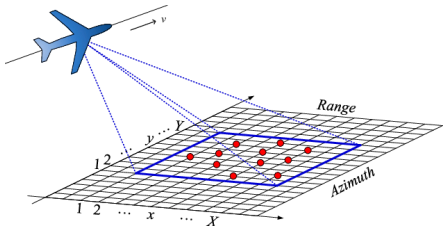
```
gvml_add_u16(vr1, vr2, vr3);
gvml_conv_u32_gf16_8k(vr0, vm0, 4, 3, …, vr7);
gvml_blk_sha1_folded(vm0, vm1, vm2, vm3, vr0);
```





compute-enabled SRAM array

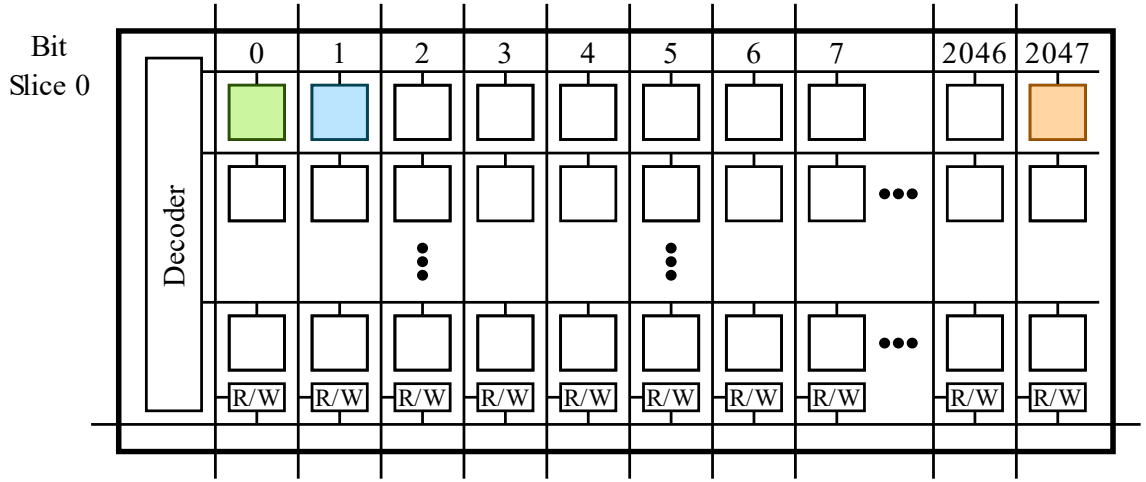




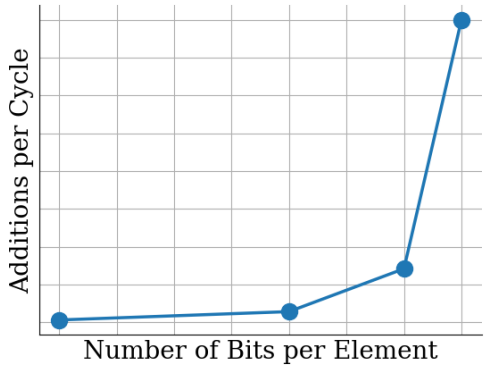
Our Goal: Explore the potential for supporting a virtual vector instruction set on a commercial compute-in-SRAM architecture to generalize beyond specific application domains

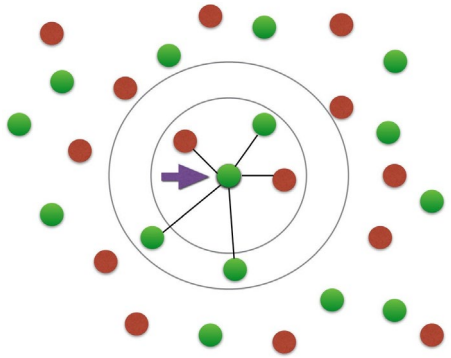
Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

- Motivation
- **APU Microarchitecture**
- APU Microcoding
- Virtual Vector Instruction Set
- Microbenchmarking Results

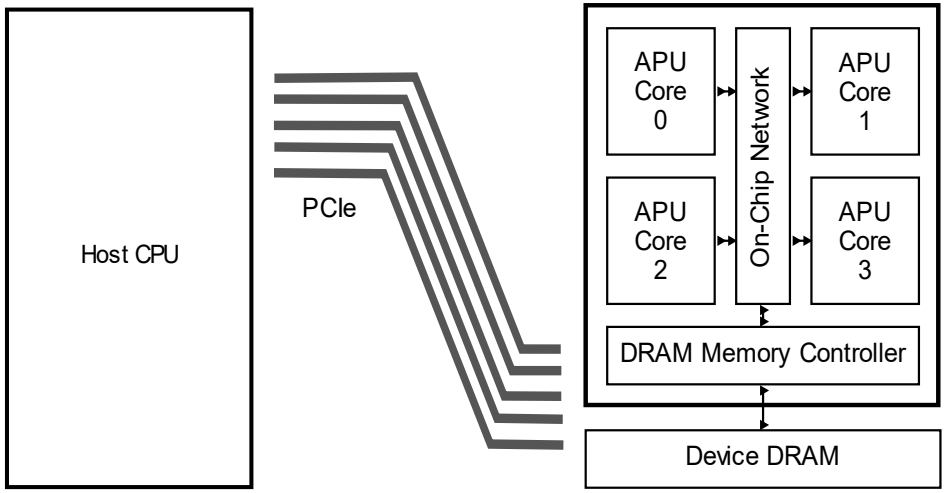



```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```



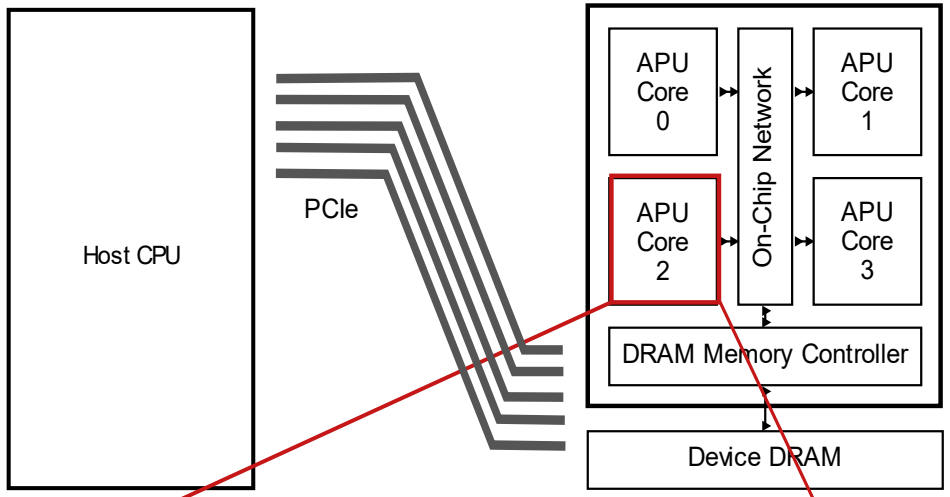


APU MICROARCHITECTURE

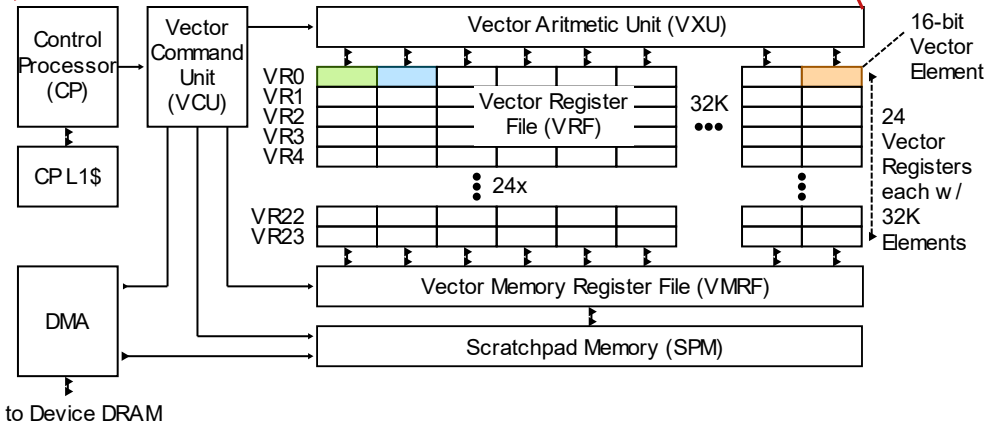


System Overview

APU MICROARCHITECTURE

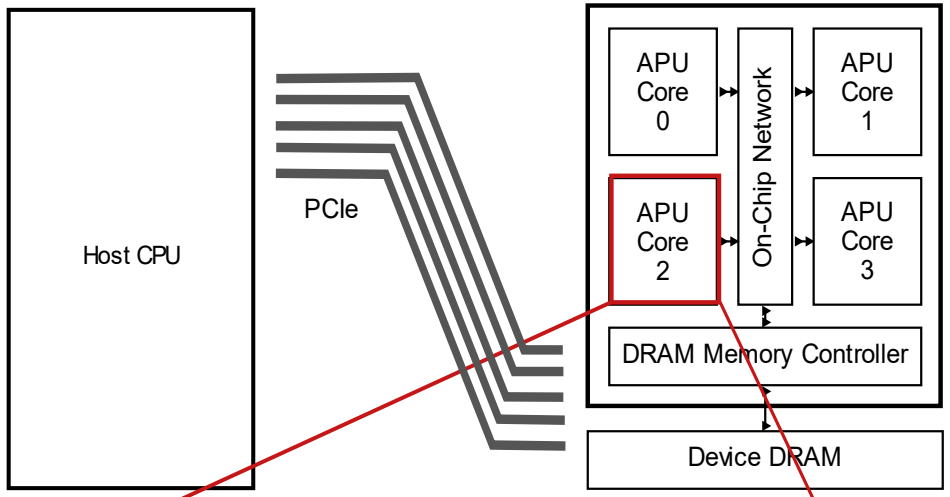


System Overview

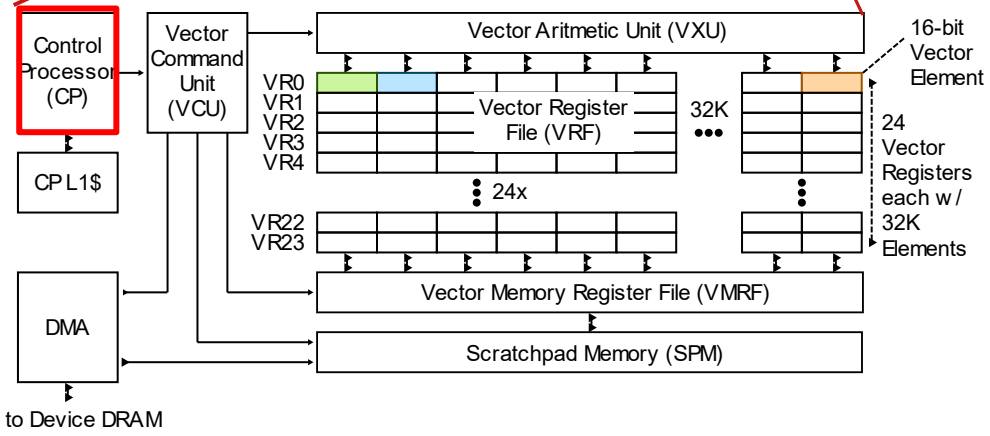


APU Core Logical View

APU MICROARCHITECTURE

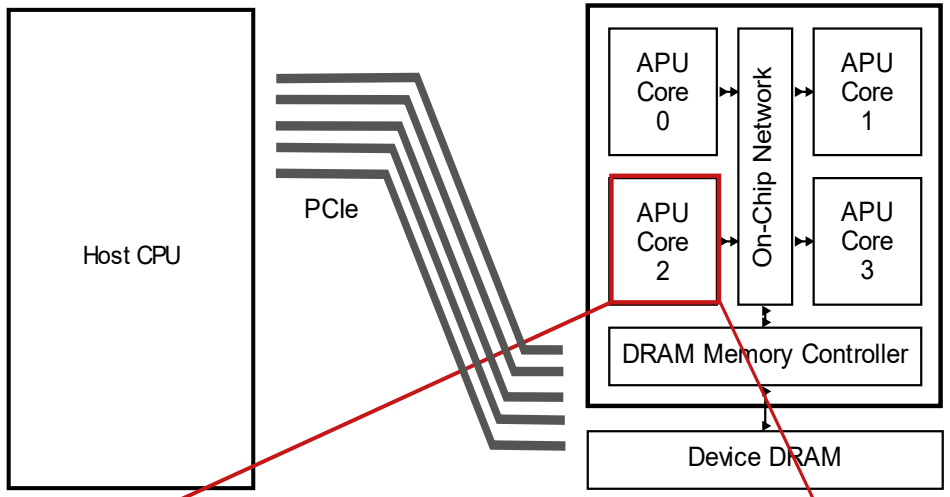


System Overview

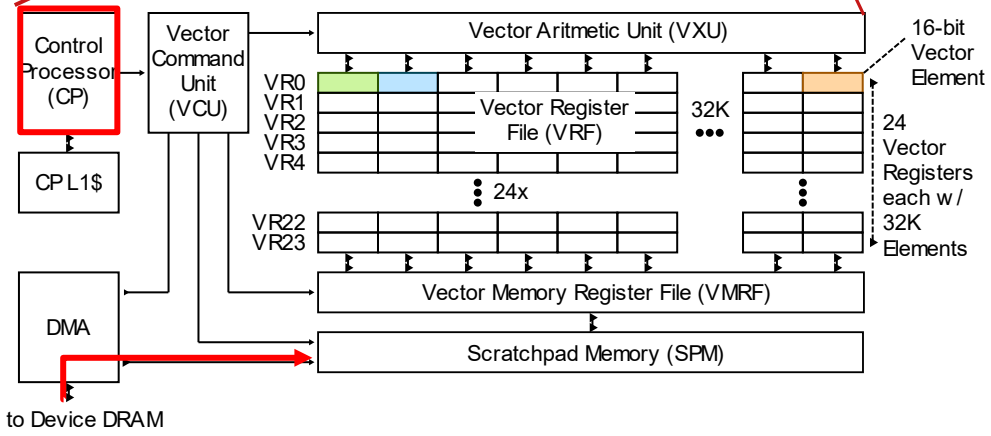


APU Core Logical View

APU MICROARCHITECTURE

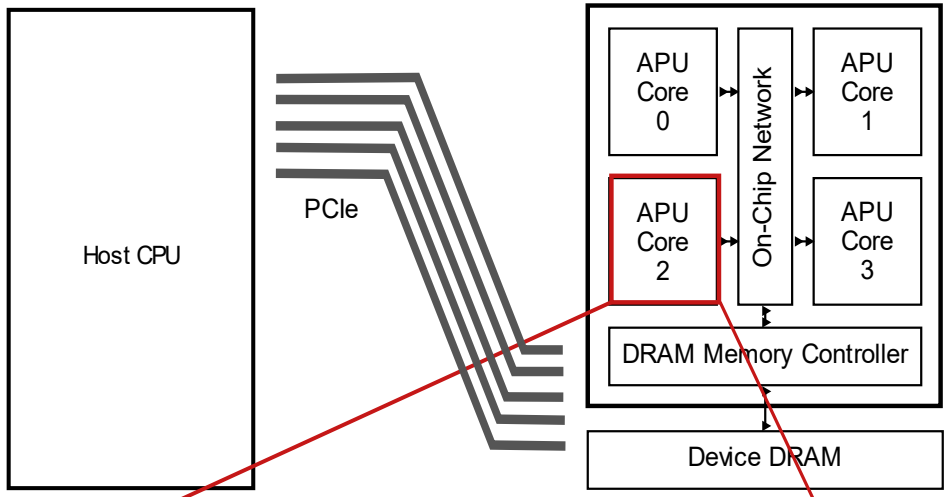


System Overview

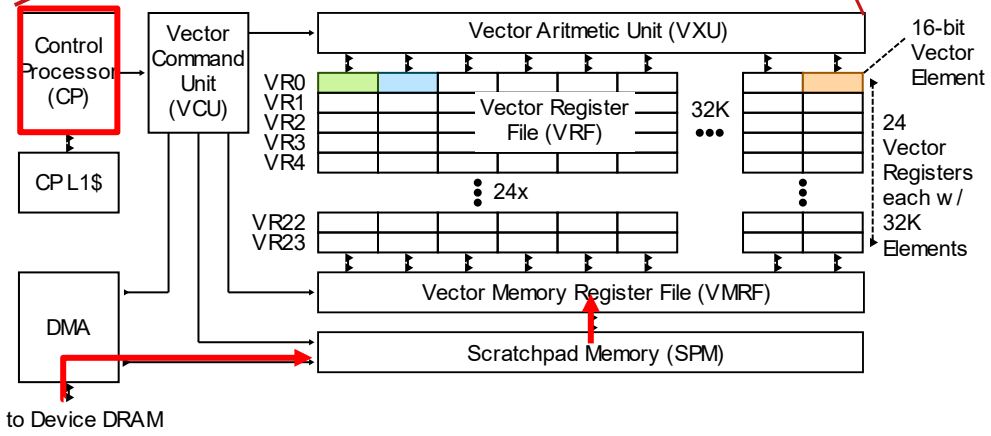


APU Core Logical View

APU MICROARCHITECTURE

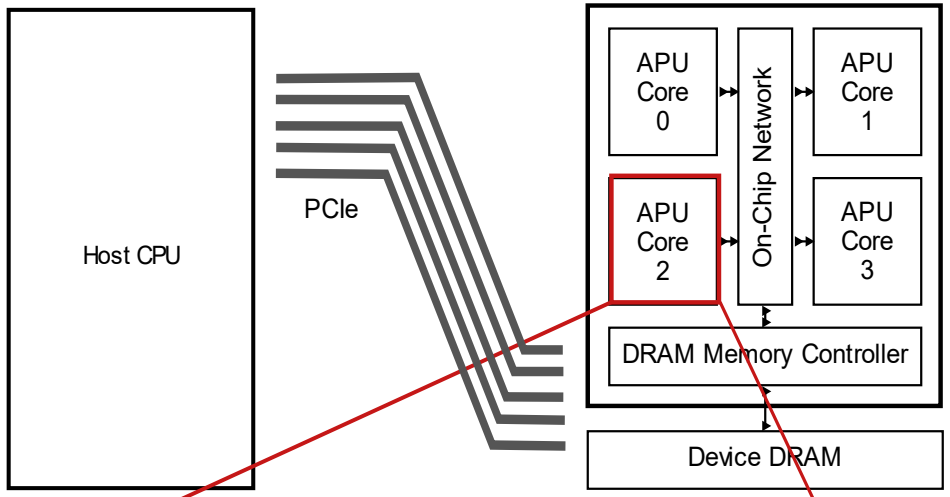


System Overview

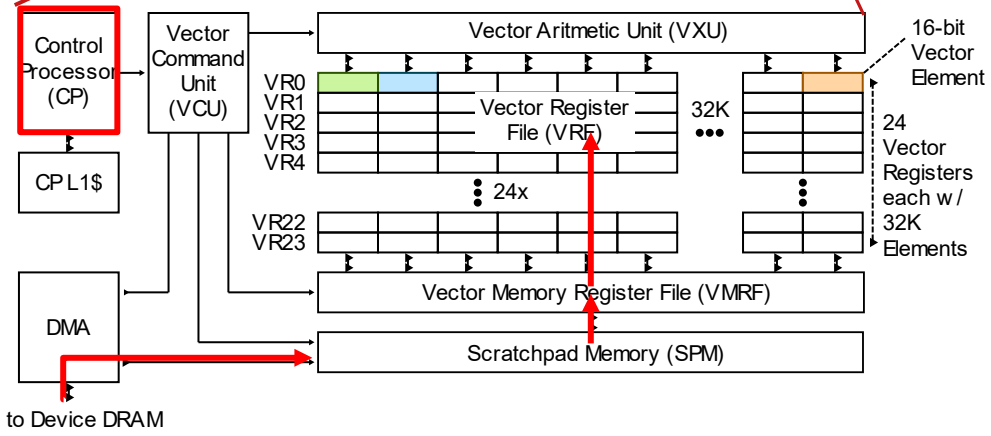


APU Core Logical View

APU MICROARCHITECTURE

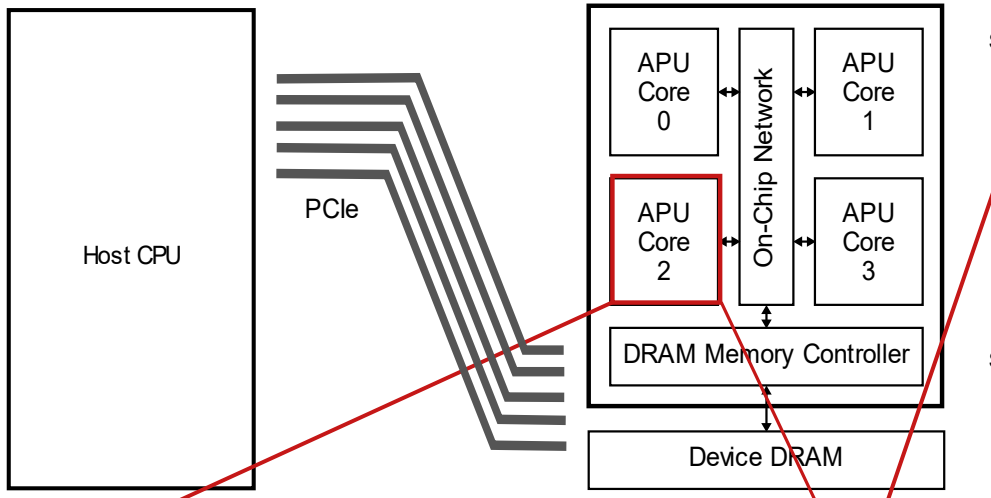


System Overview

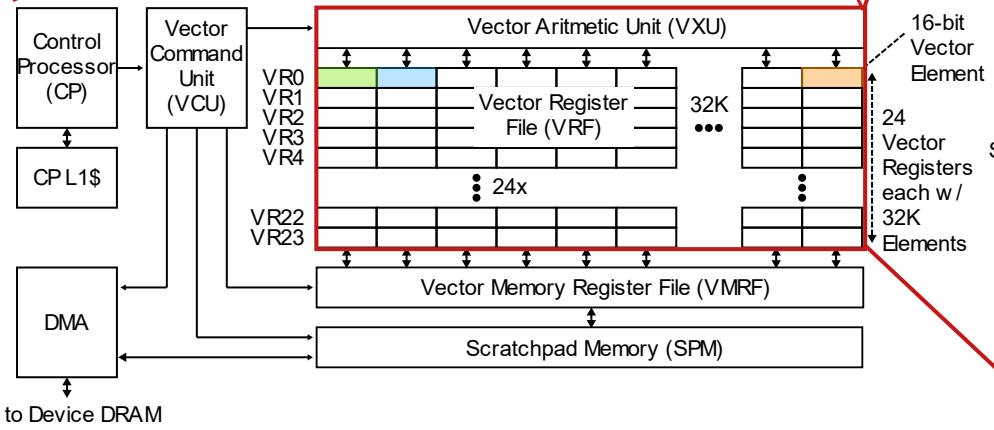


APU Core Logical View

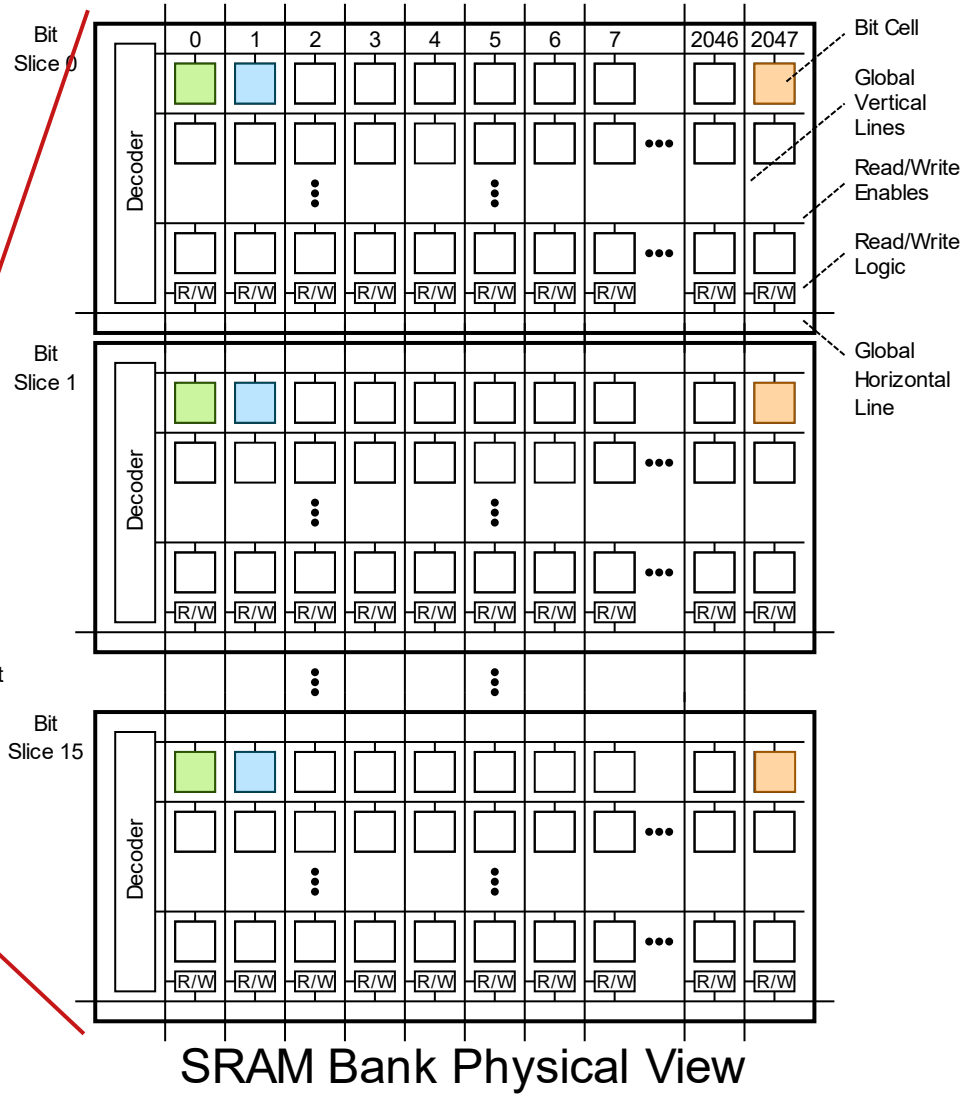
APU MICROARCHITECTURE



System Overview

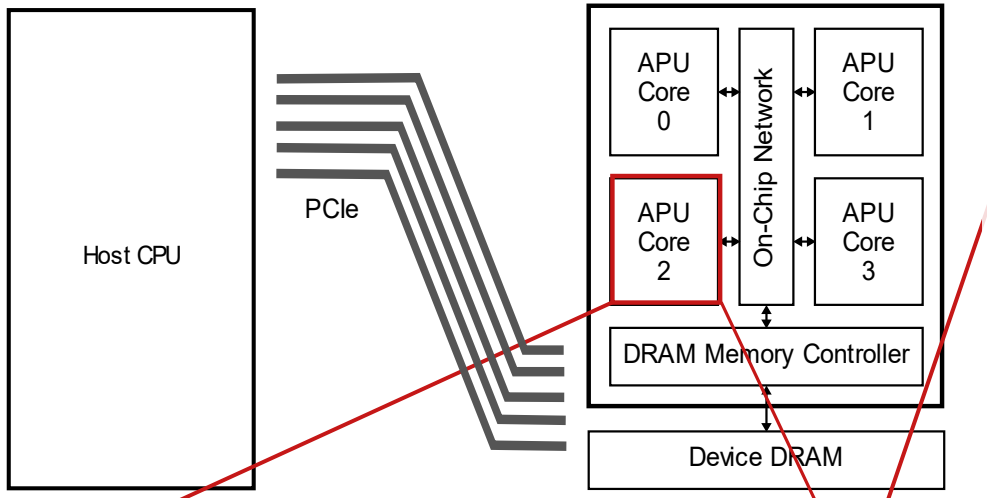


APU Core Logical View

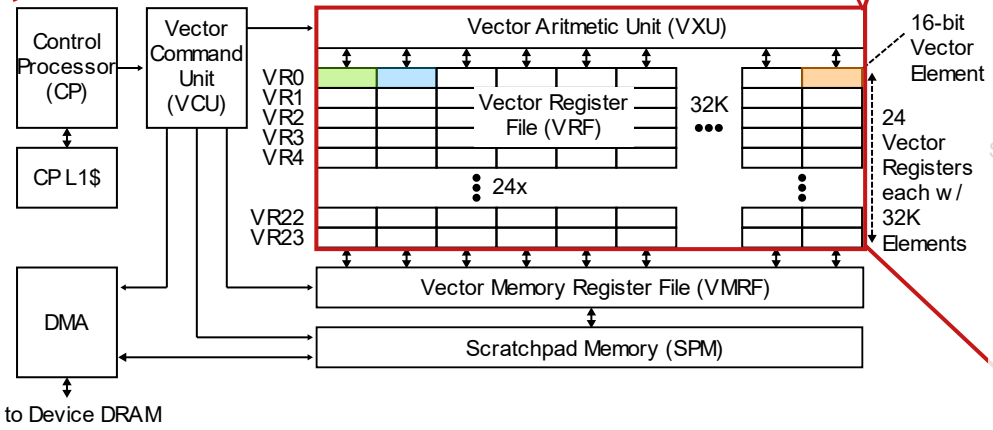


SRAM Bank Physical View

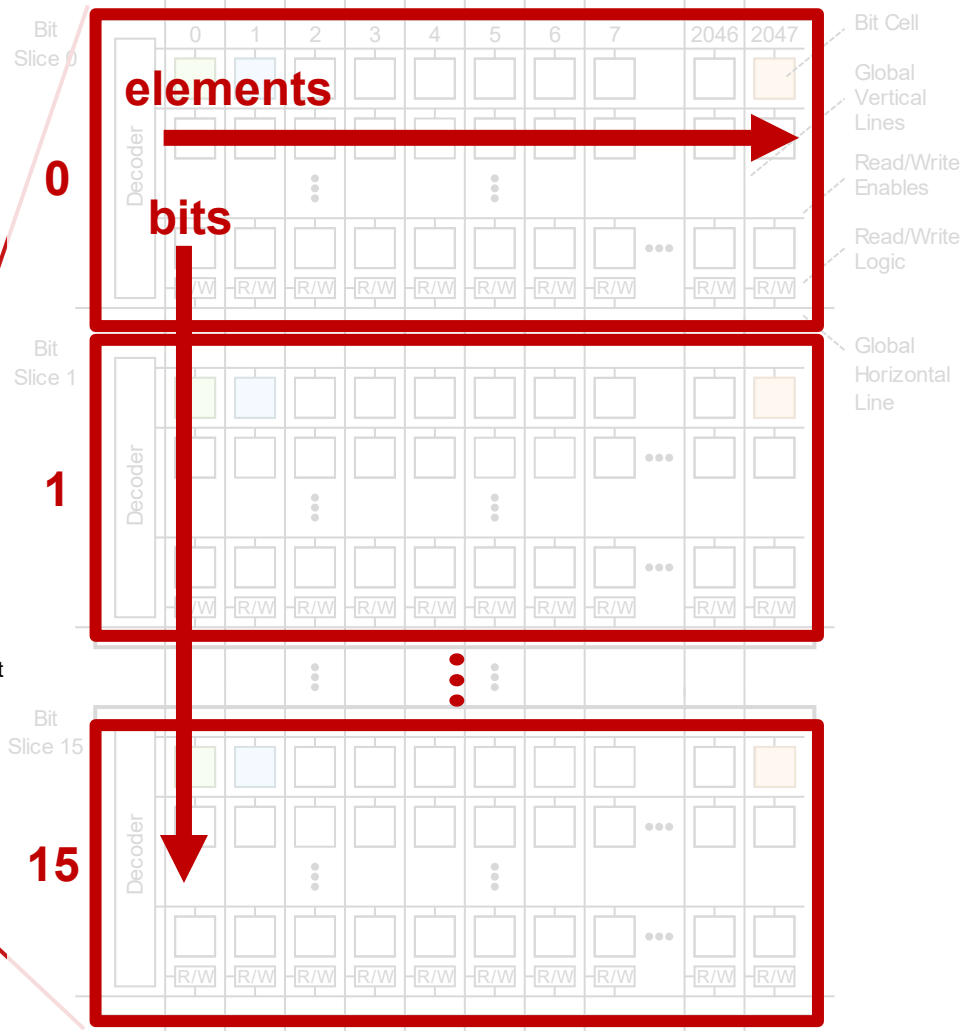
APU MICROARCHITECTURE



System Overview

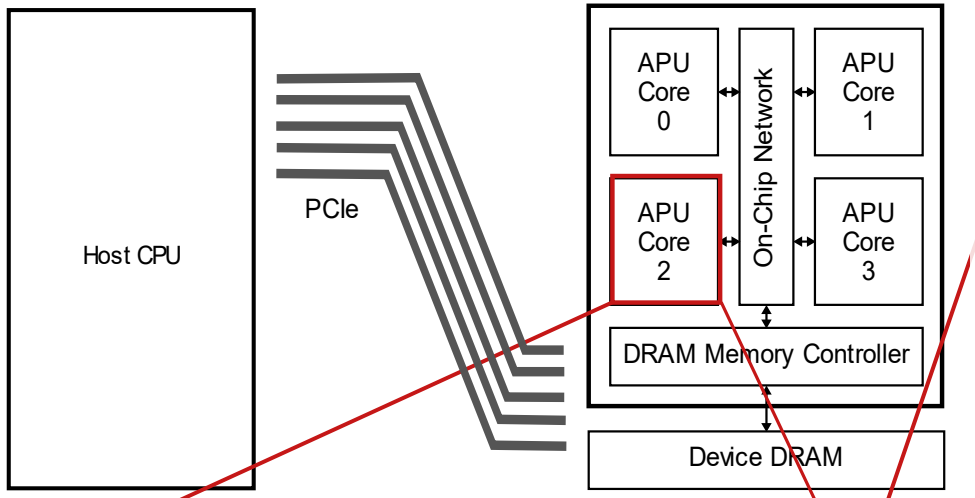


APU Core Logical View

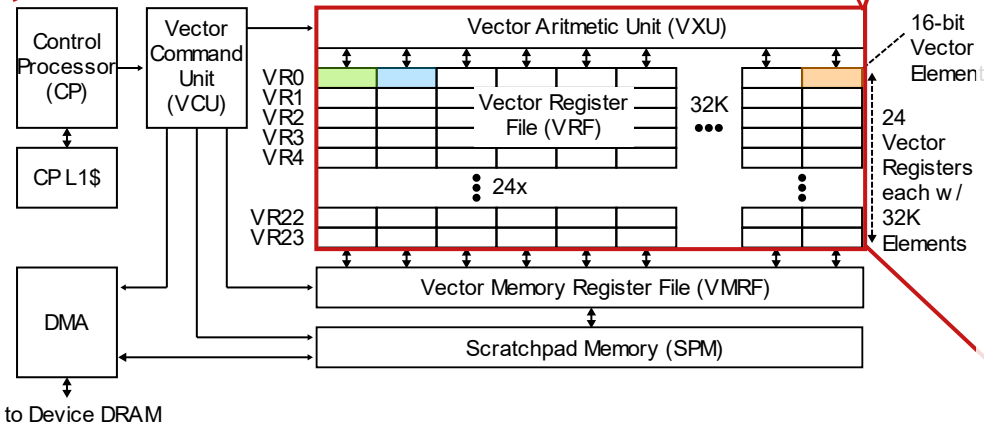


SRAM Bank Physical View

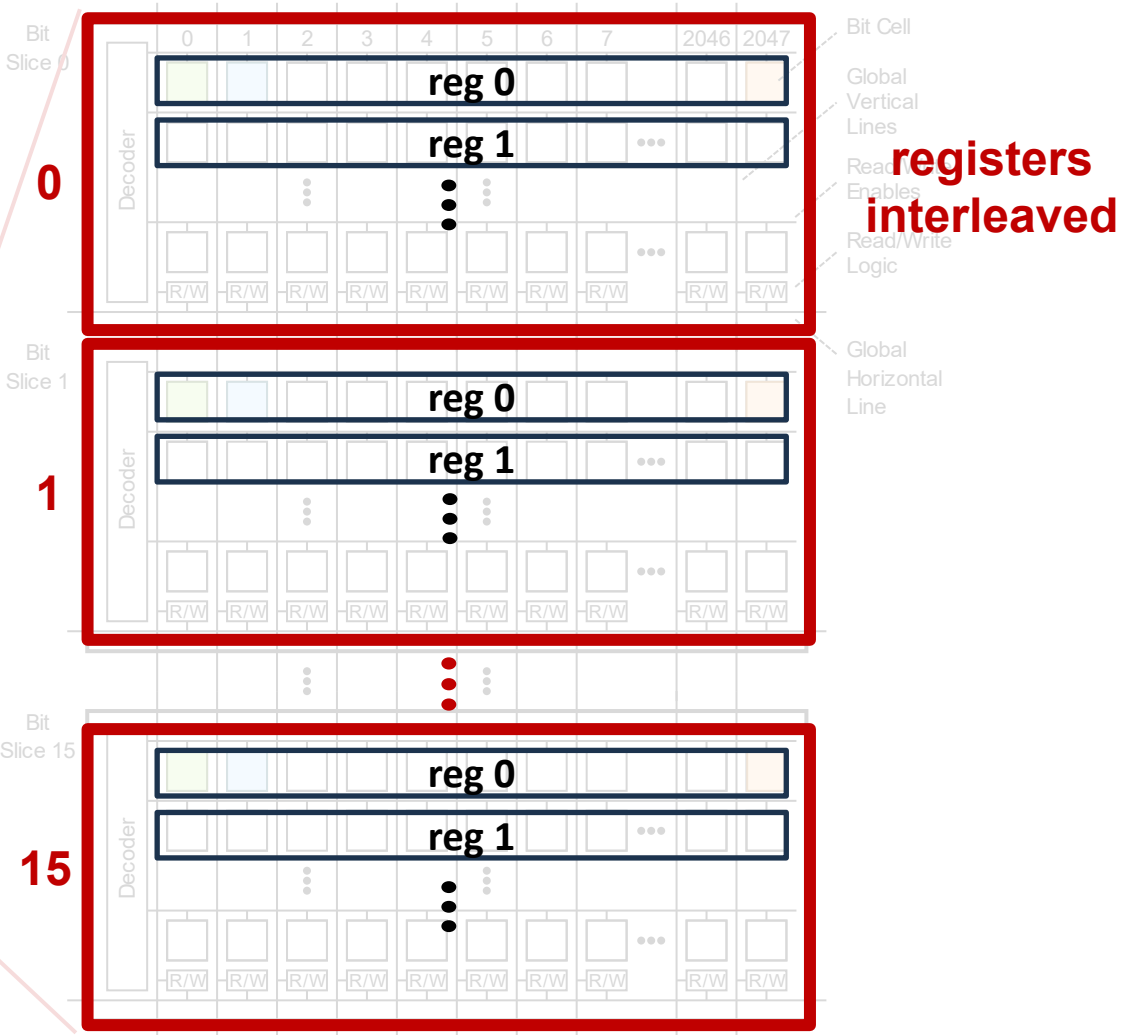
APU MICROARCHITECTURE



System Overview

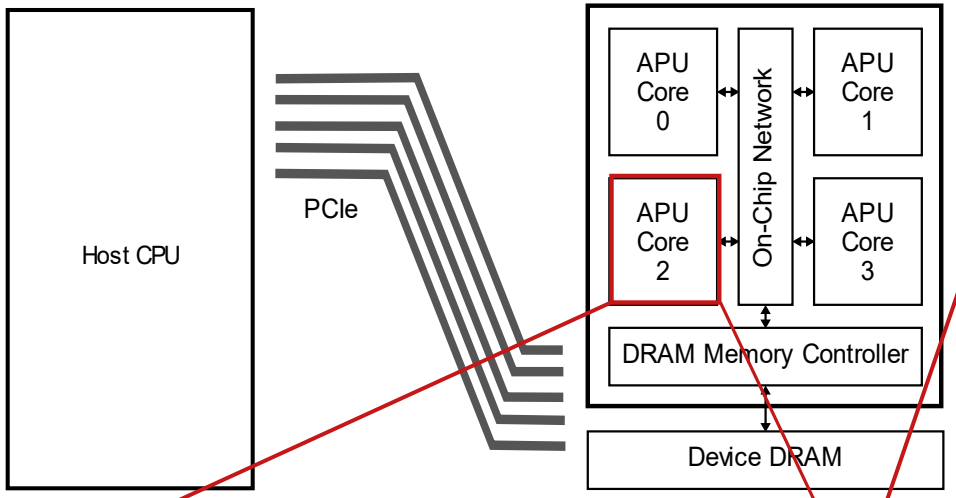


APU Core Logical View

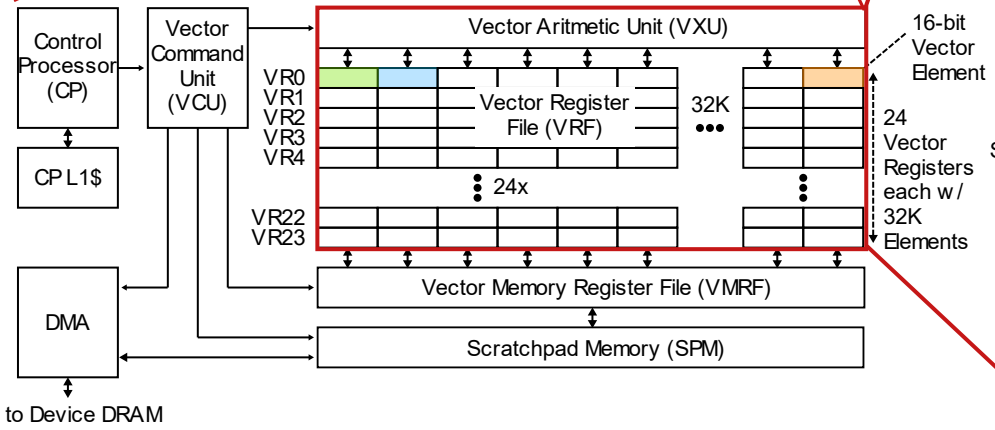


SRAM Bank Physical View

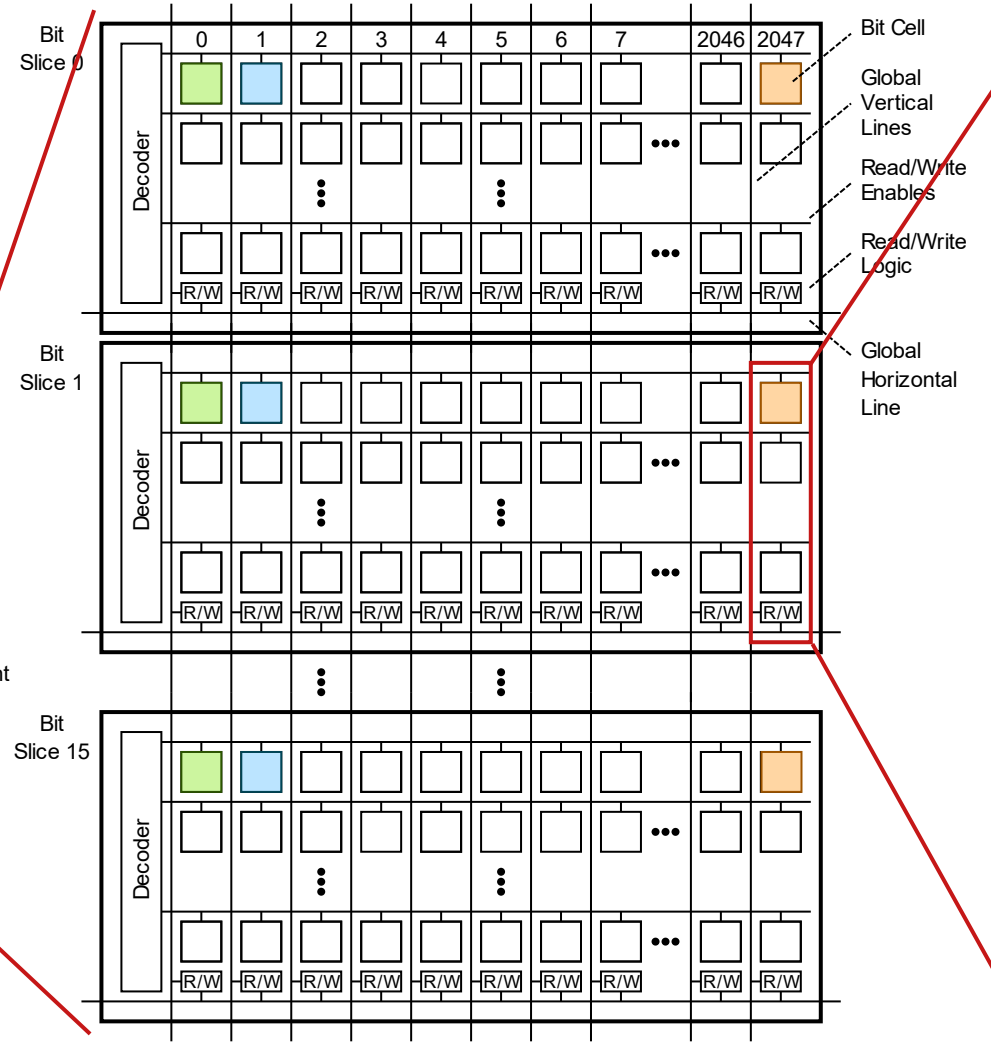
APU MICROARCHITECTURE



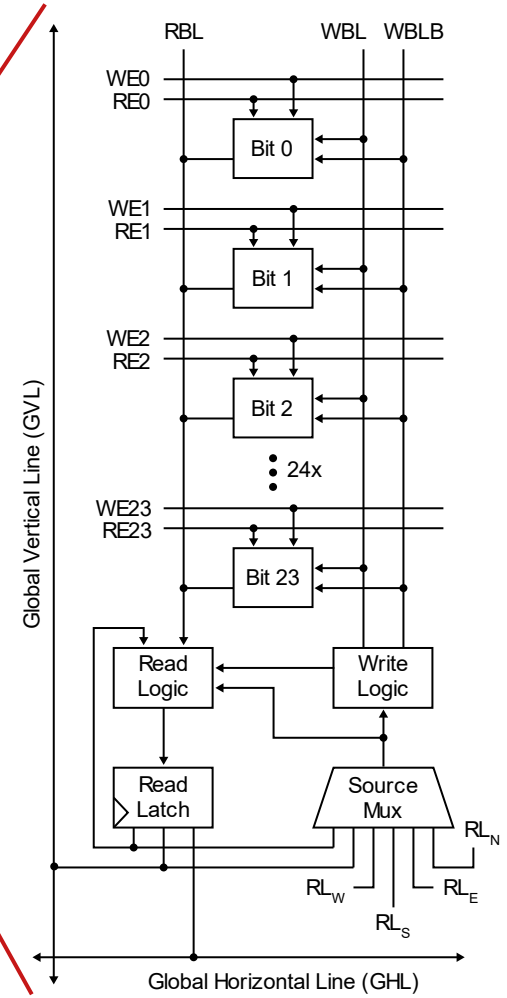
System Overview



APU Core Logical View

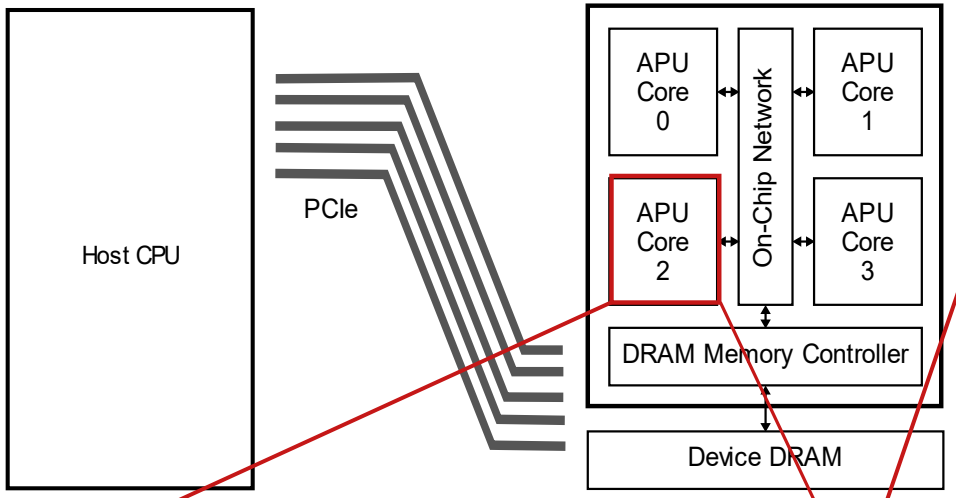


SRAM Bank Physical View

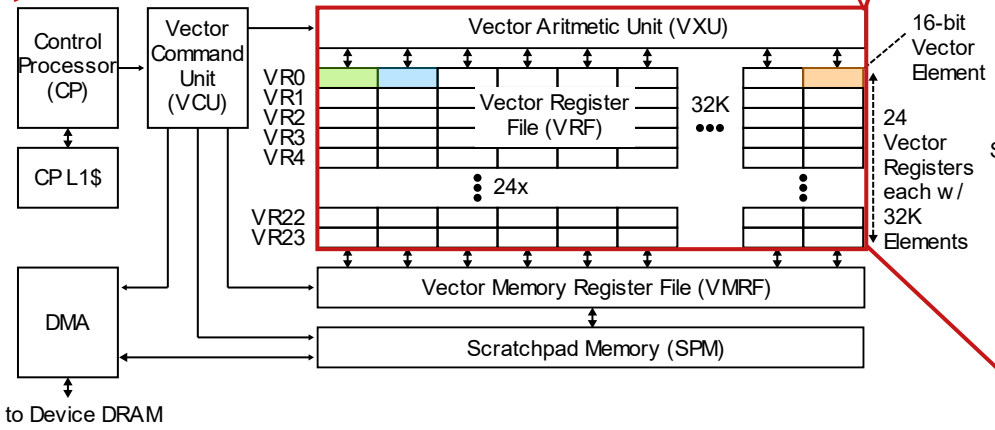


Bit Processor Circuitry

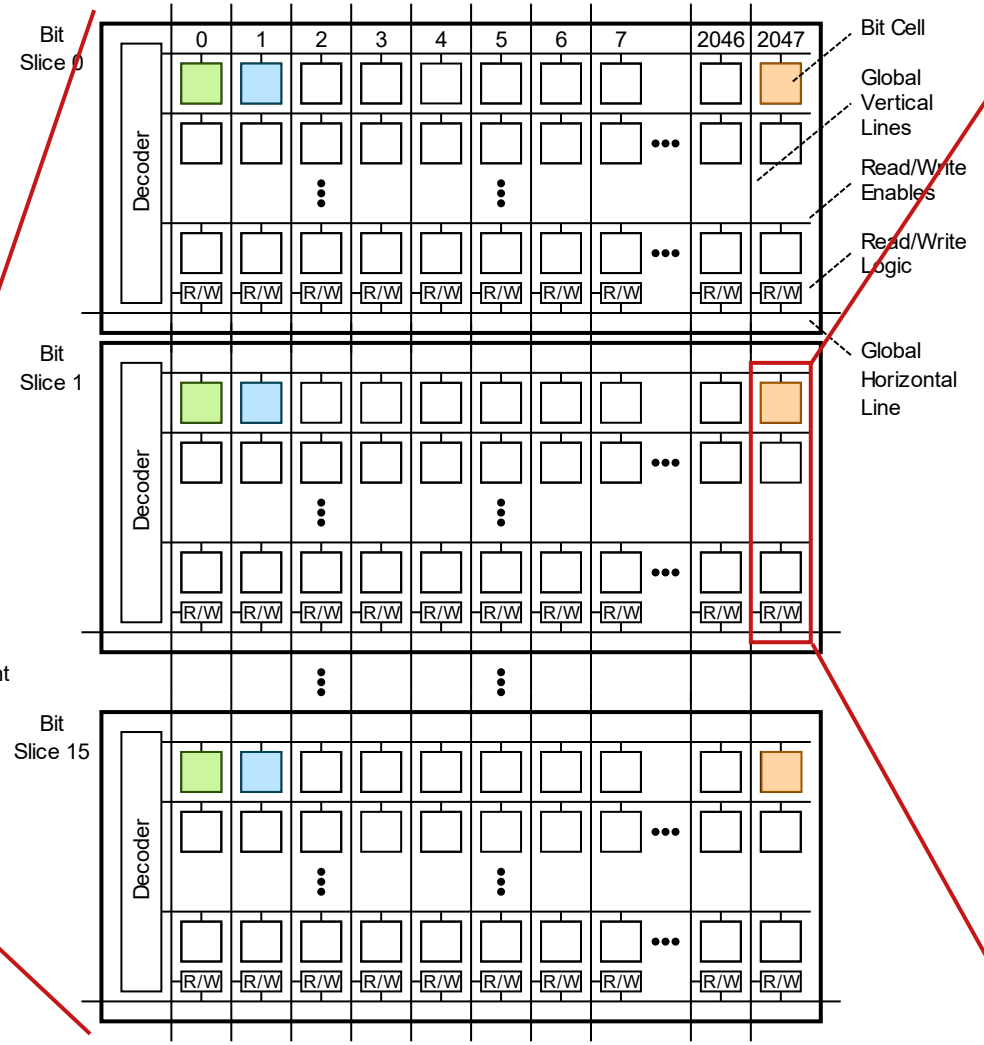
APU MICROARCHITECTURE



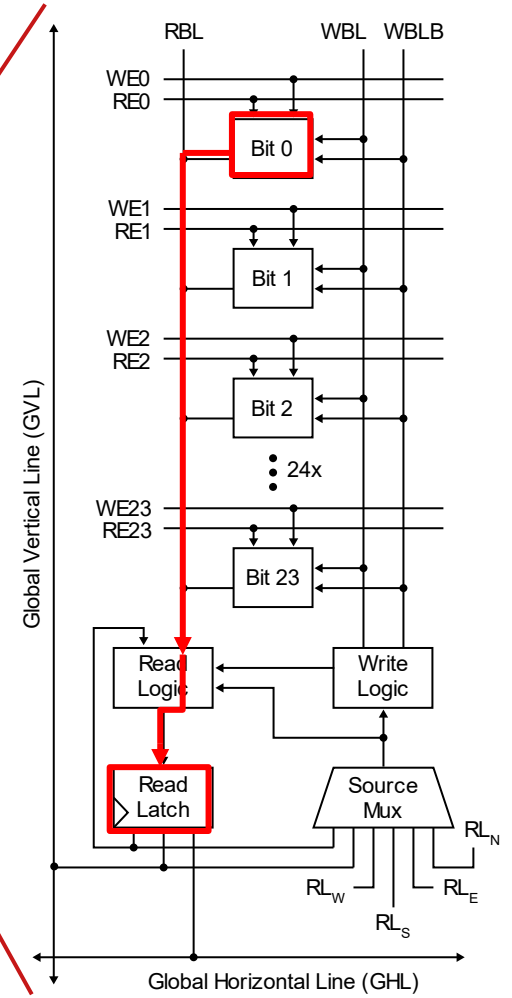
System Overview



APU Core Logical View

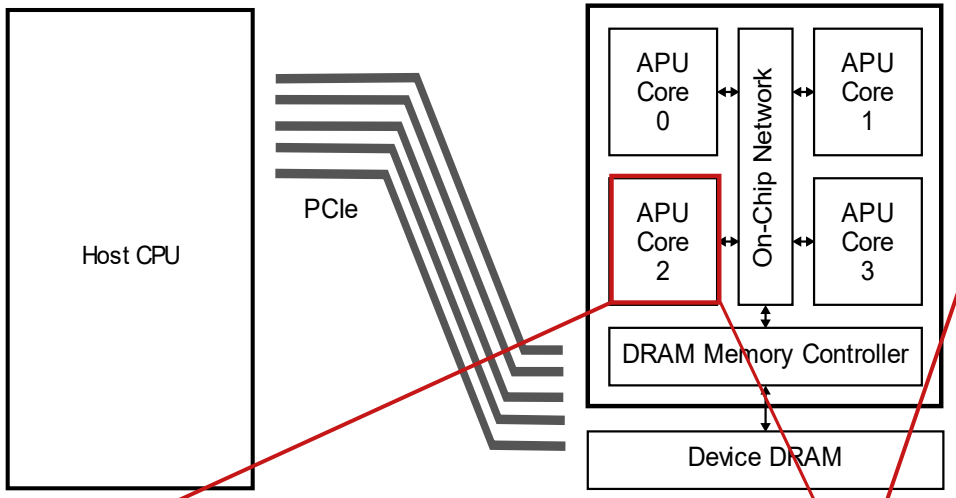


SRAM Bank Physical View

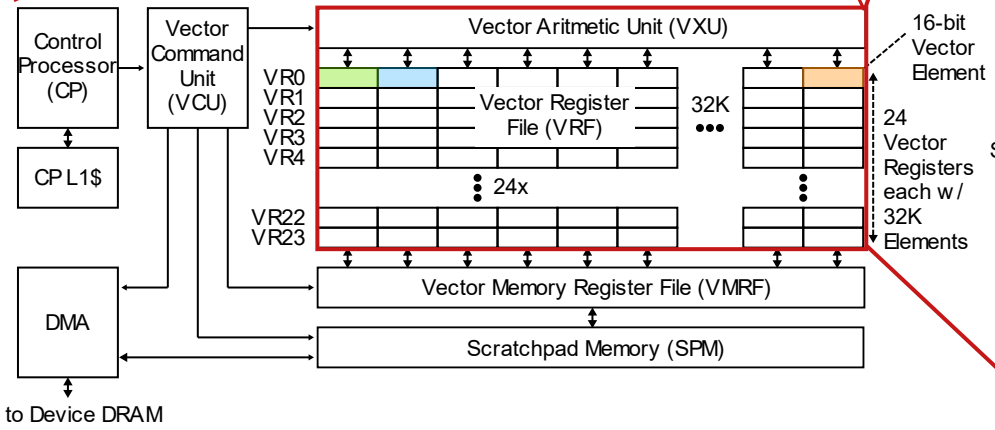


Bit Processor Circuitry

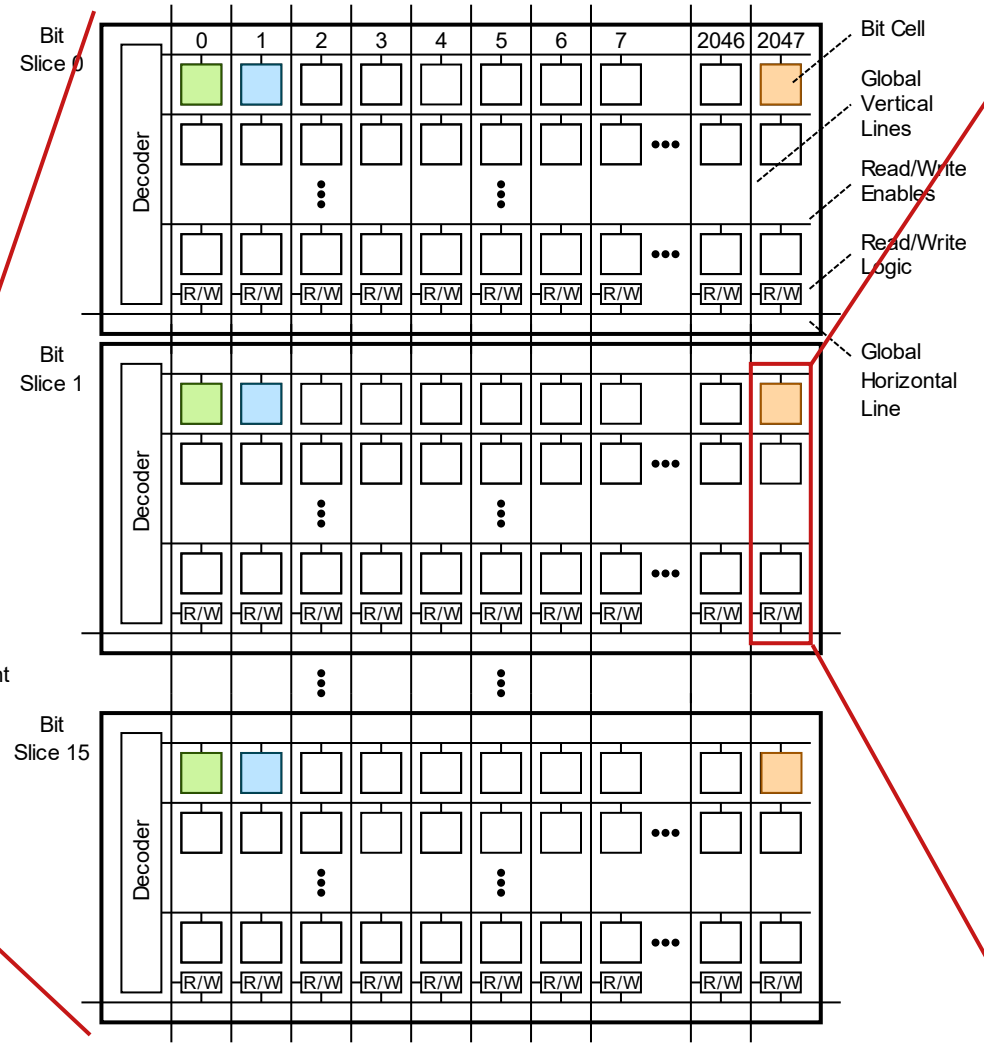
APU MICROARCHITECTURE



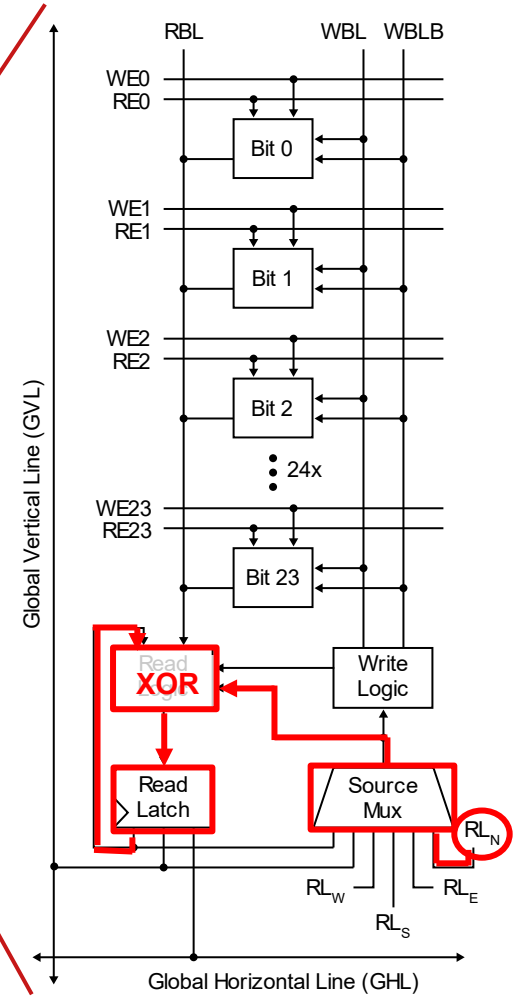
System Overview



APU Core Logical View



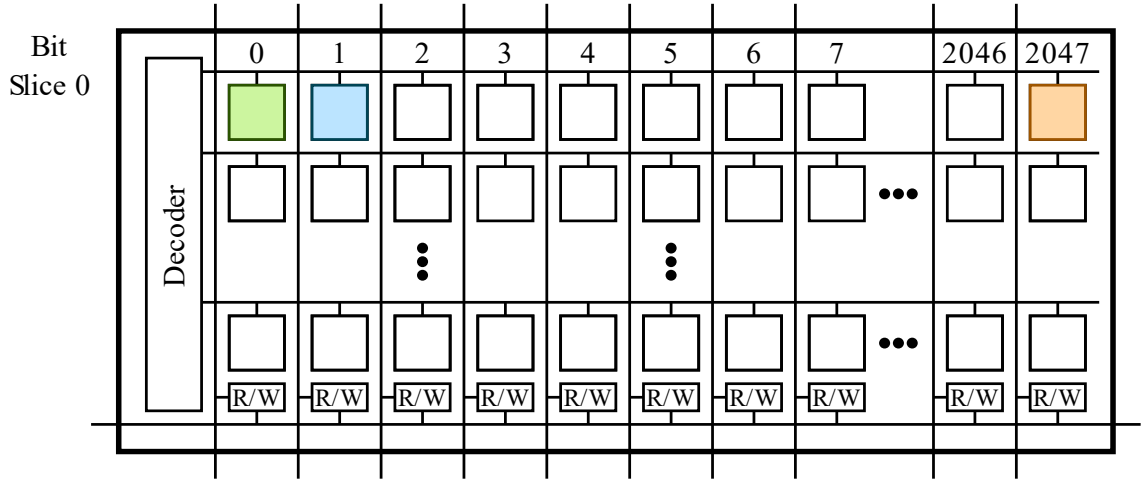
SRAM Bank Physical View



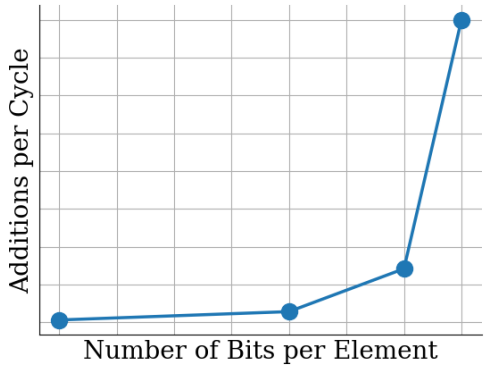
Bit Processor Circuitry

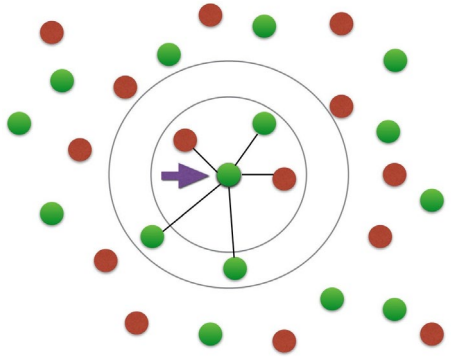
Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

- Motivation
- APU Microarchitecture
- **APU Microcoding**
- Virtual Vector Instruction Set
- Microbenchmarking Results




```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```

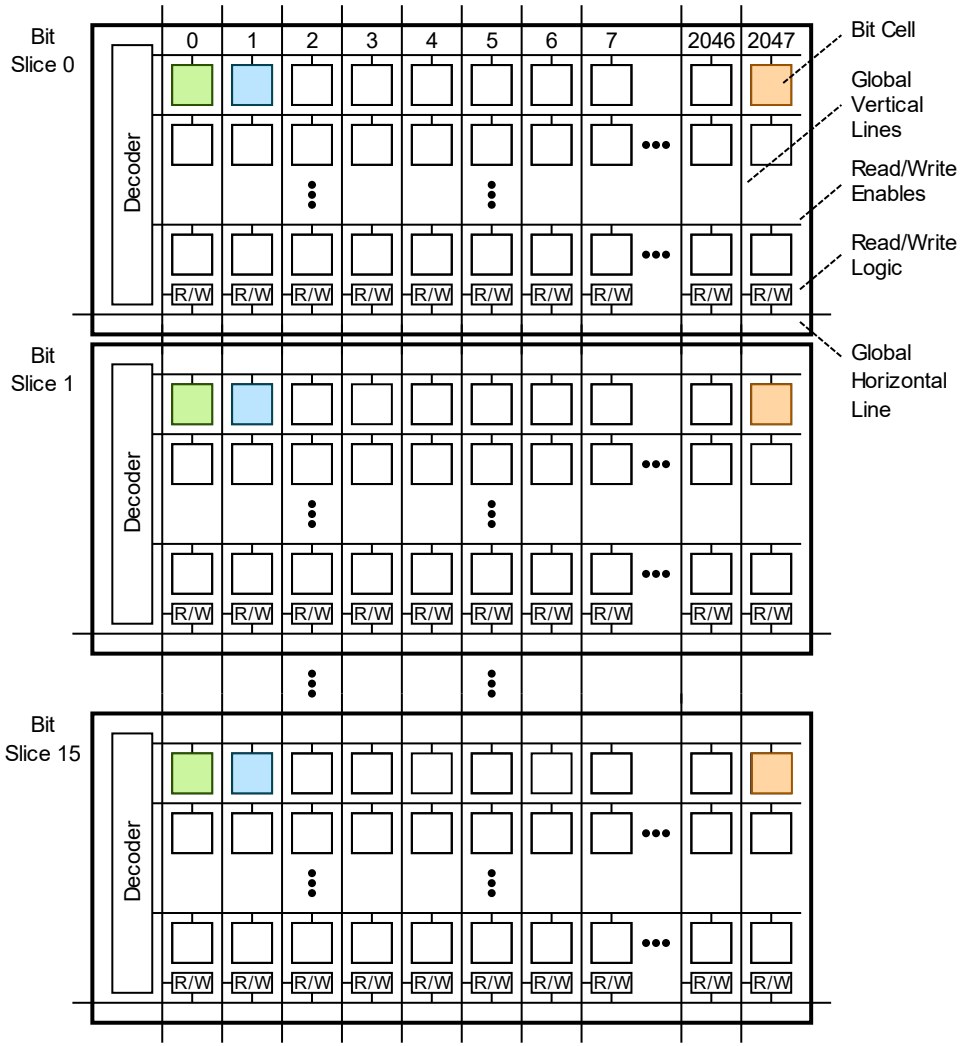




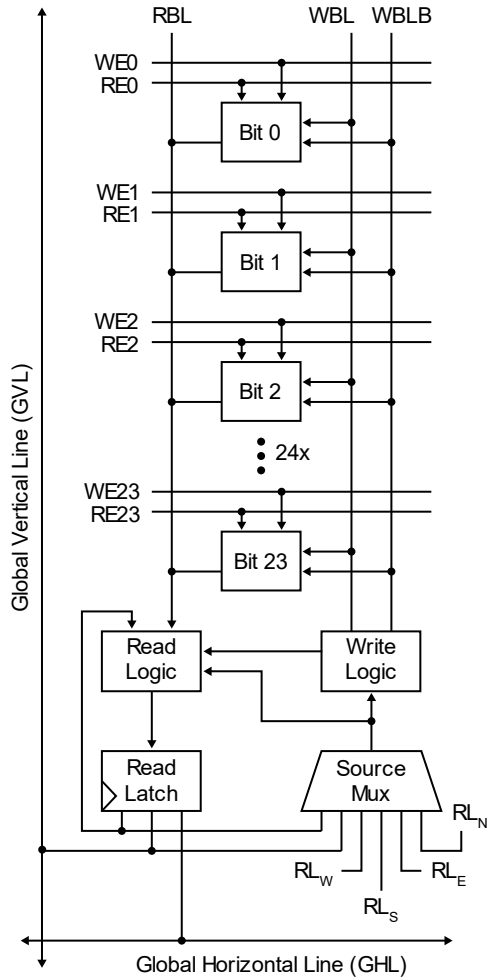
EXAMPLE #1: BITWISE OR

```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```

```
GAL_TASK_ENTRY_POINT(apu_task_1, in, out)
{
  gvml_init_once();
  struct commonif_struct *cmn_handle =
                (struct commonif_struct *)in;
  enum gvml_vr16 a      = GVML_VR16_0;
  enum gvml_vr16 b      = GVML_VR16_1;
  enum gvml_vr16 output = GVML_VR16_2;
  size_t vlen = 32768;
  vload(a, cmn_handle->a_hndl, vlen);
  vload(b, cmn_handle->b_hndl, vlen);
  bitwise_or(output, a, b);
  vstore(cmn_handle->output_hndl, output, vlen);
  return SUCCESS;
}
```



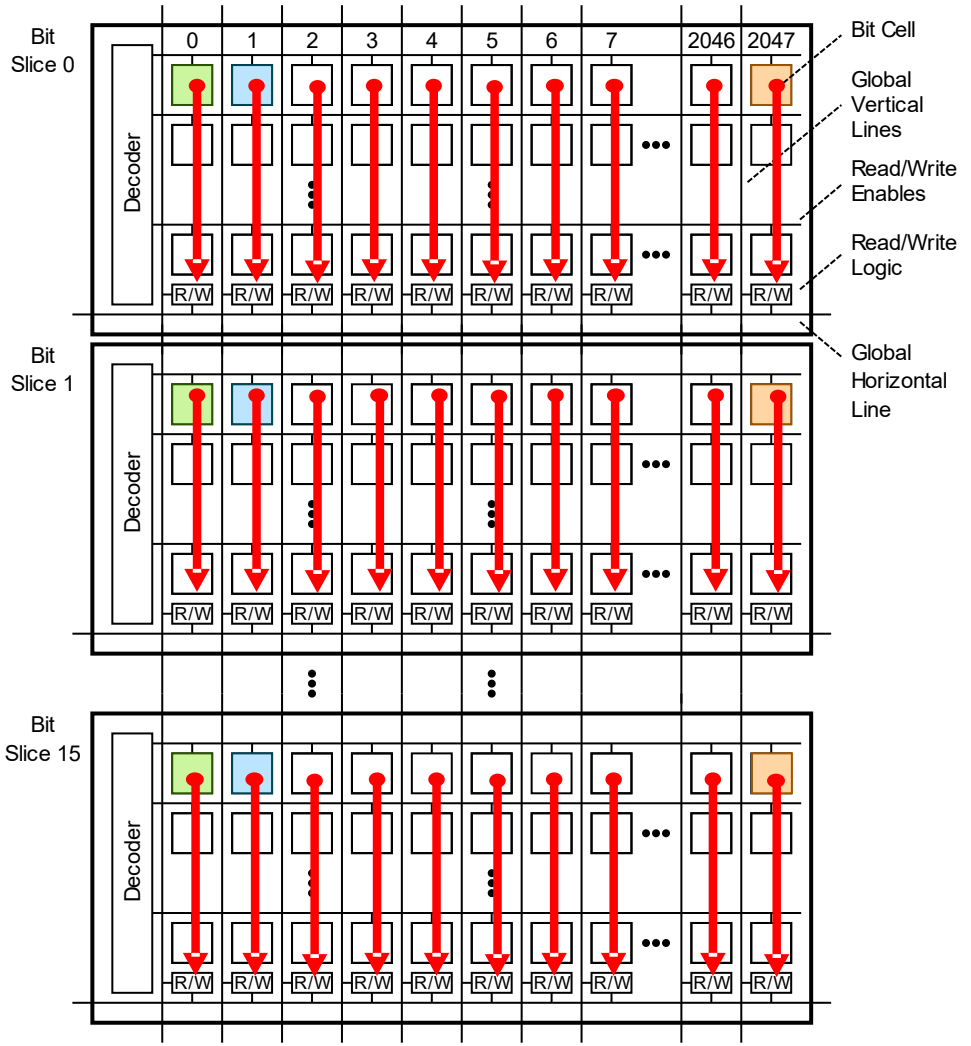
SRAM Bank Physical View



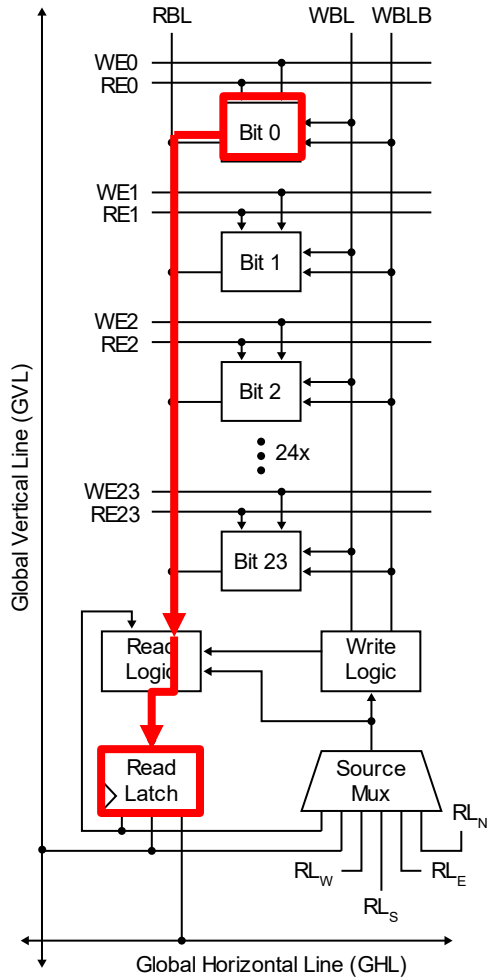
Bit Processor Circuitry

EXAMPLE #1: BITWISE OR

```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```



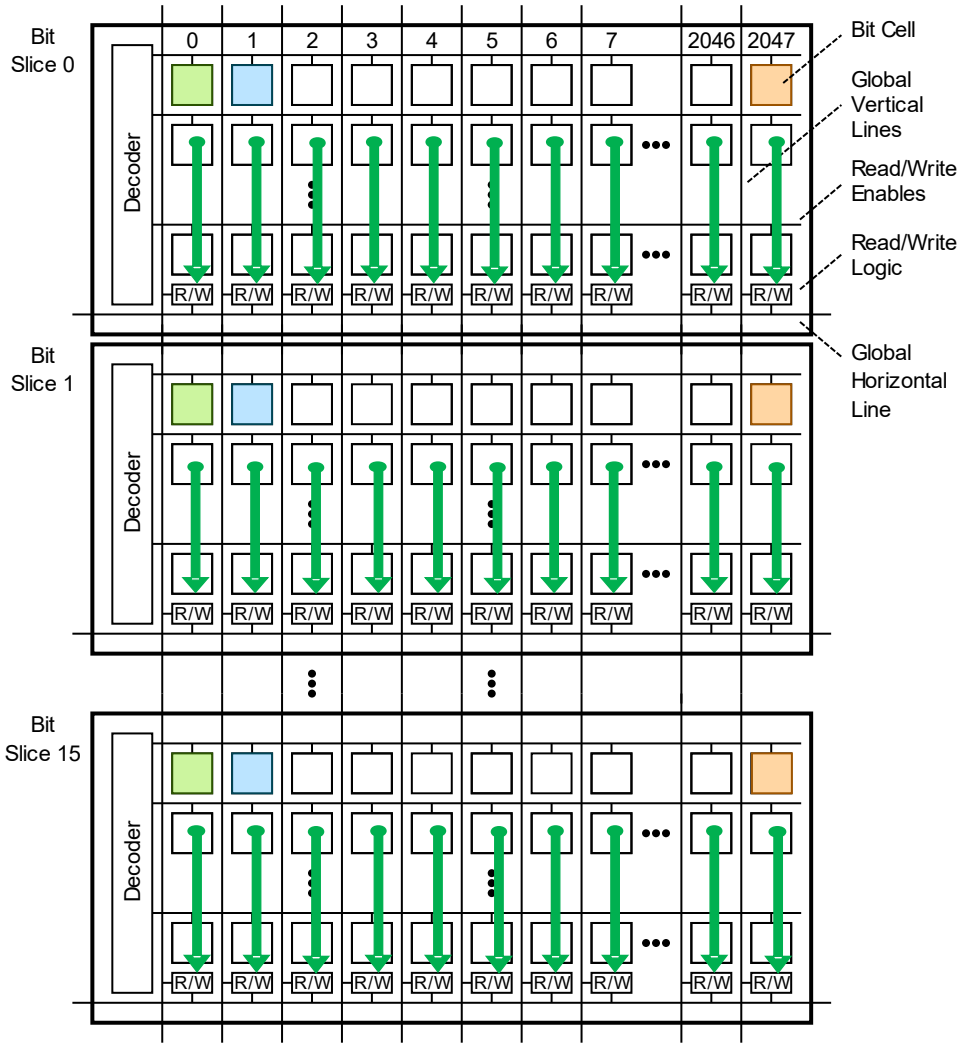
SRAM Bank Physical View



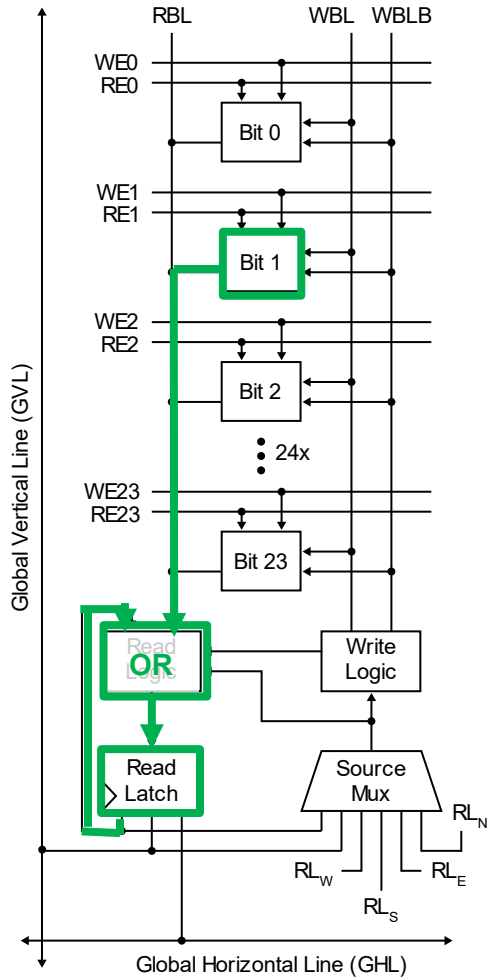
Bit Processor Circuitry

EXAMPLE #1: BITWISE OR

```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```



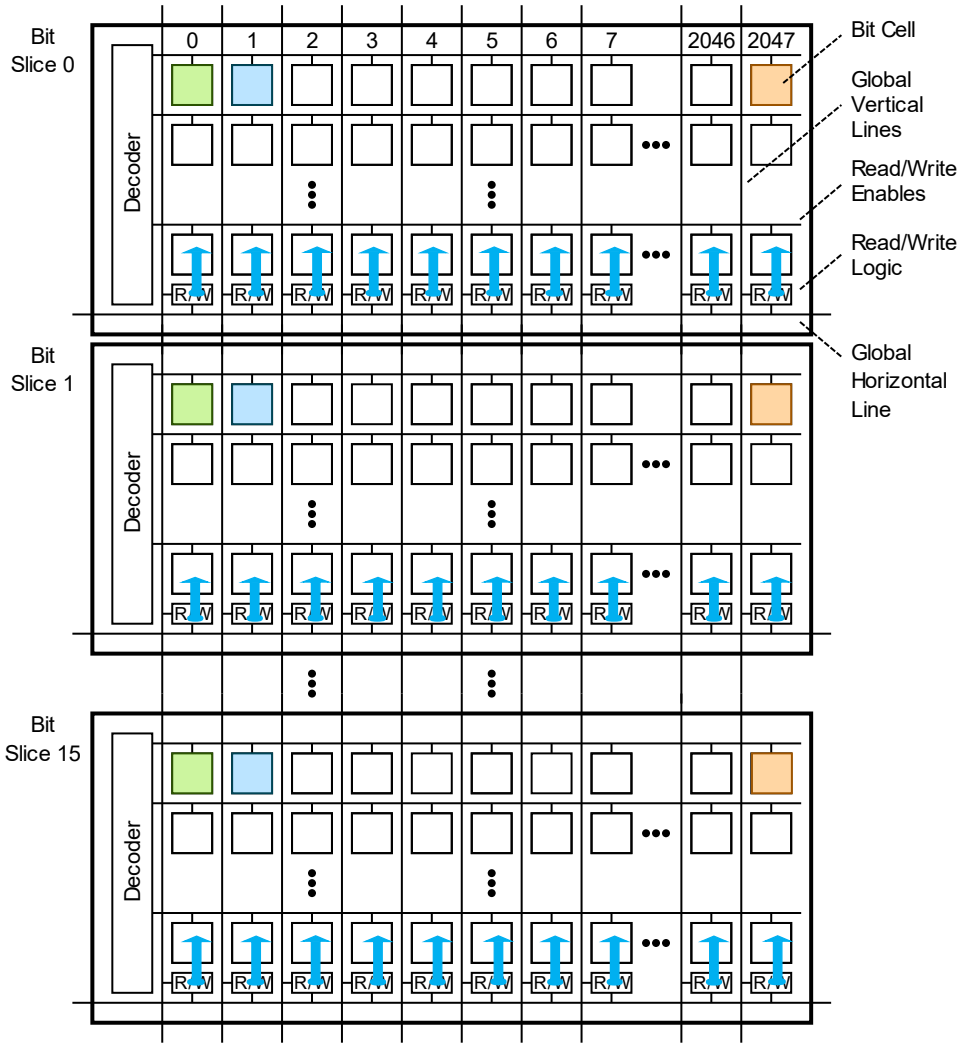
SRAM Bank Physical View



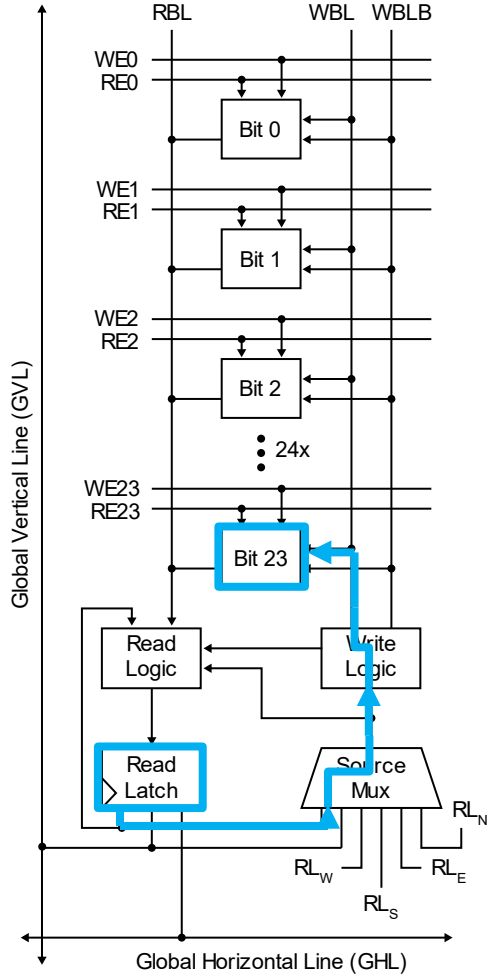
Bit Processor Circuitry

EXAMPLE #1: BITWISE OR

```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```



SRAM Bank Physical View

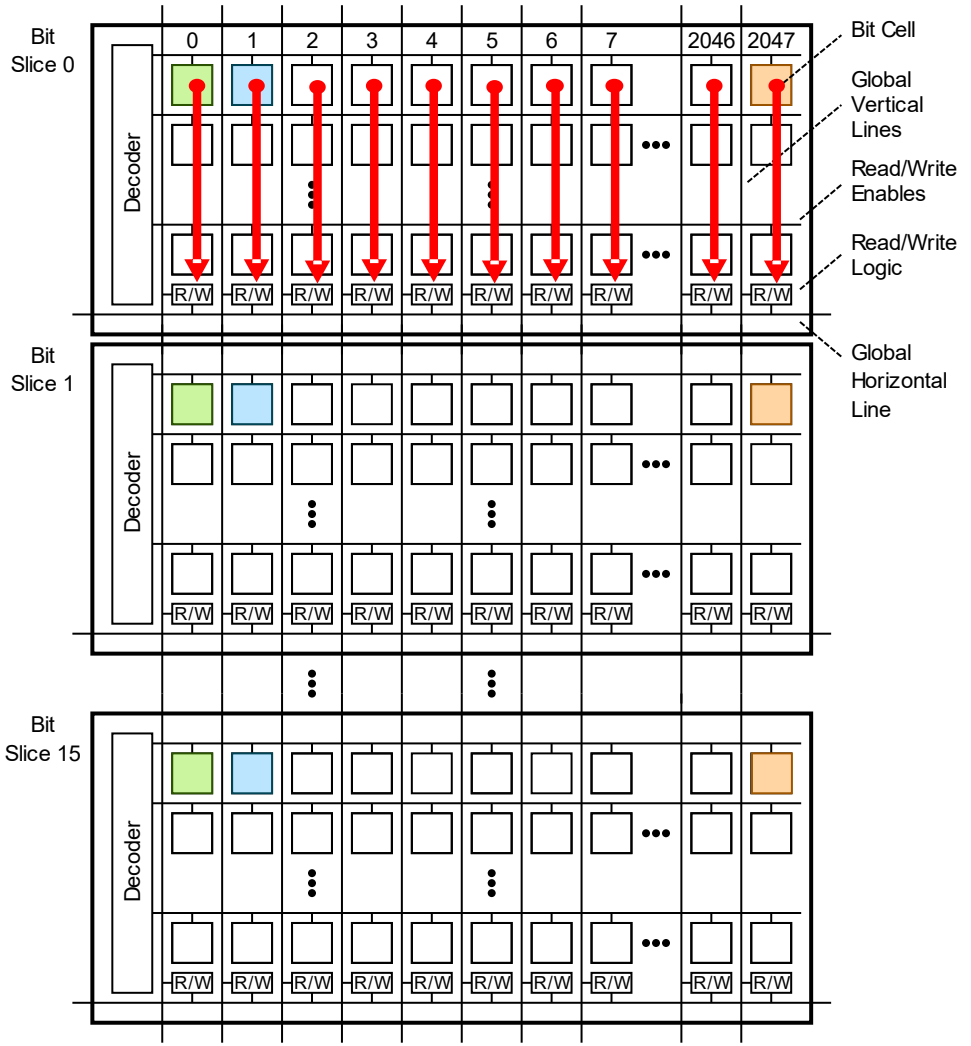


Bit Processor Circuitry

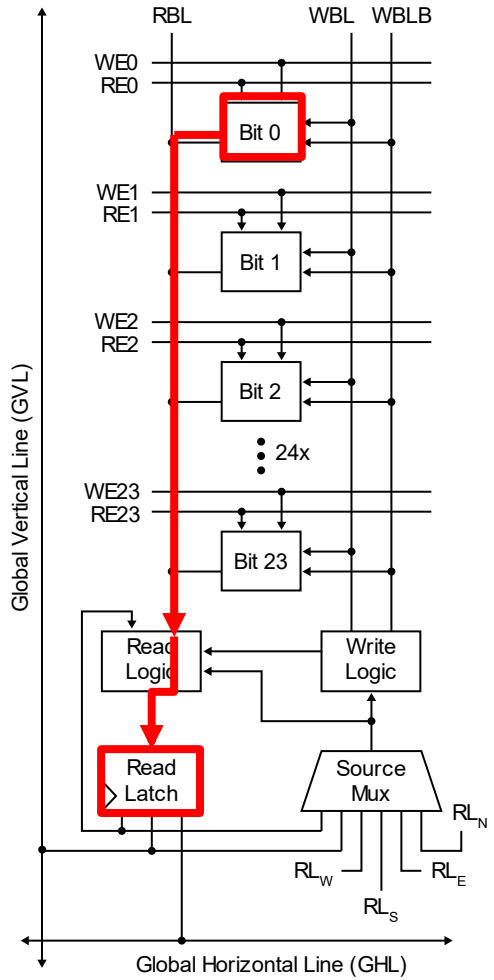
EXAMPLE #2: VECTOR-VECTOR ADD (BIT 0)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 ^ vsrc1
  0x0001:   RL = VRF[vsrc0];
  0x0001:   RL ^= VRF[vsrc1];
  0x0001:   VRF[vdst] = RL;

  // cout = vsrc0 * vsrc1
  0x0001:   RL = VRF[vsrc0, vsrc1];

  ...
```

vsrc0	vsrc1	vdst	c_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



SRAM Bank Physical View

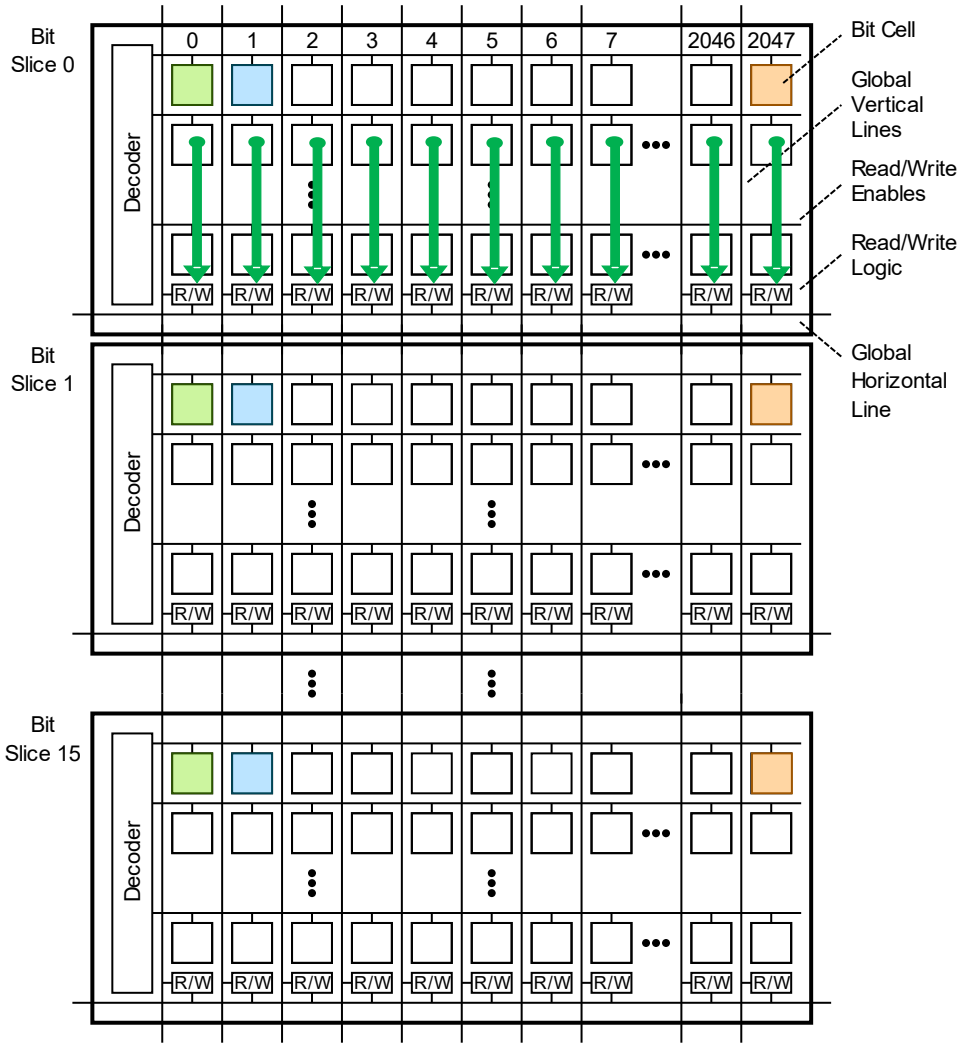


Bit Processor Circuitry

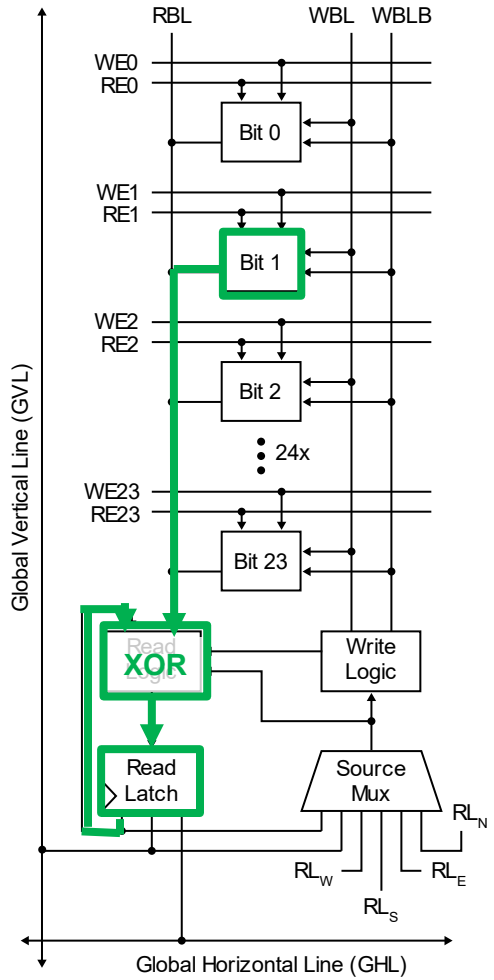
EXAMPLE #2: VECTOR-VECTOR ADD (BIT 0)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 ^ vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;

  // cout = vsrc0 * vsrc1
  0x0001:  RL = VRF[vsrc0, vsrc1];

  ...
```

vsrc0	vsrc1	vdst	c_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



SRAM Bank Physical View

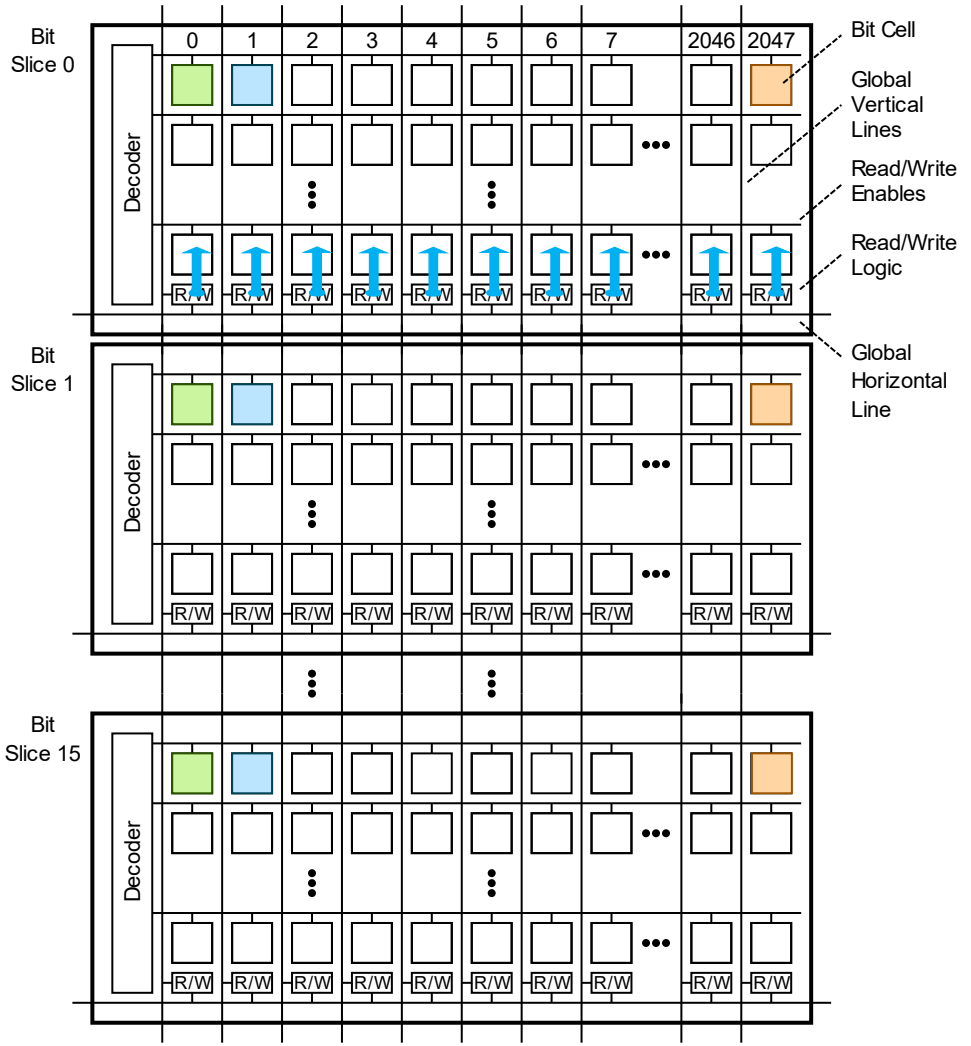


Bit Processor Circuitry

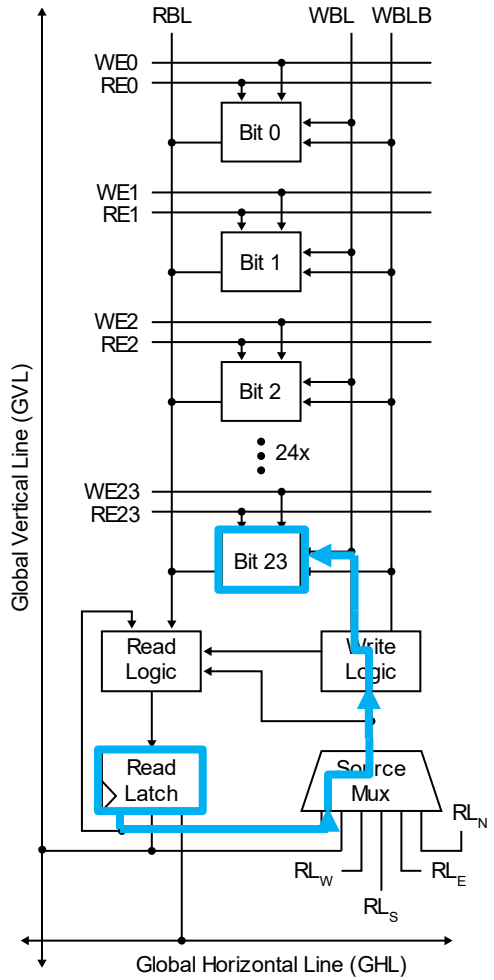
EXAMPLE #2: VECTOR-VECTOR ADD (BIT 0)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 ^ vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;

  // cout = vsrc0 * vsrc1
  0x0001:  RL = VRF[vsrc0, vsrc1];

  ...
```

vsrc0	vsrc1	vdst	c_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



SRAM Bank Physical View

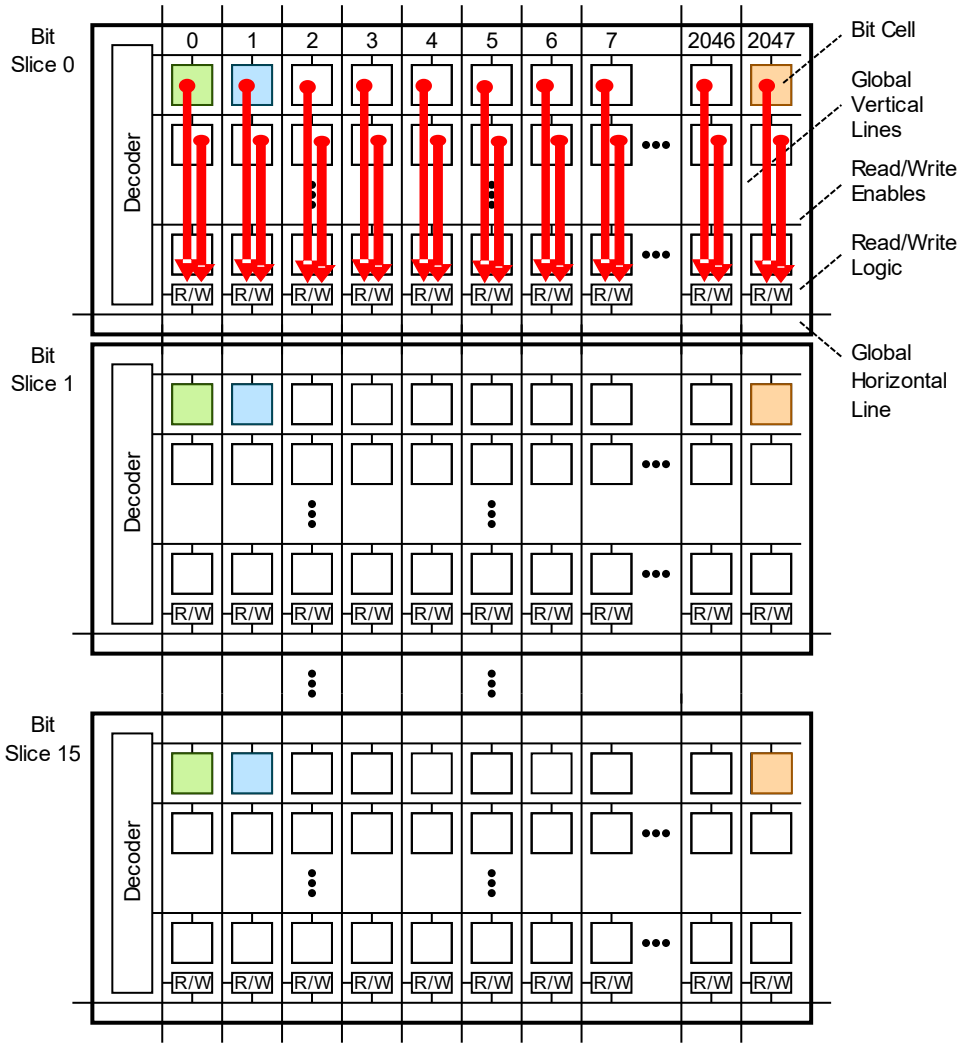


Bit Processor Circuitry

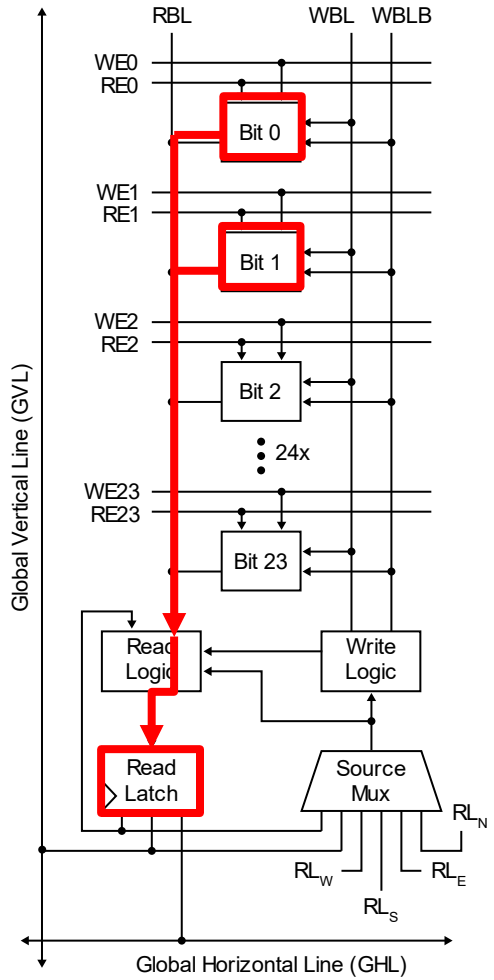
EXAMPLE #2: VECTOR-VECTOR ADD (BIT 0)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 ^ vsrc1
  0x0001:   RL = VRF[vsrc0];
  0x0001:   RL ^= VRF[vsrc1];
  0x0001:   VRF[vdst] = RL;

  // cout = vsrc0 * vsrc1
  0x0001:   RL = VRF[vsrc0, vsrc1];

  ...
```

vsrc0	vsrc1	vdst	c_out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



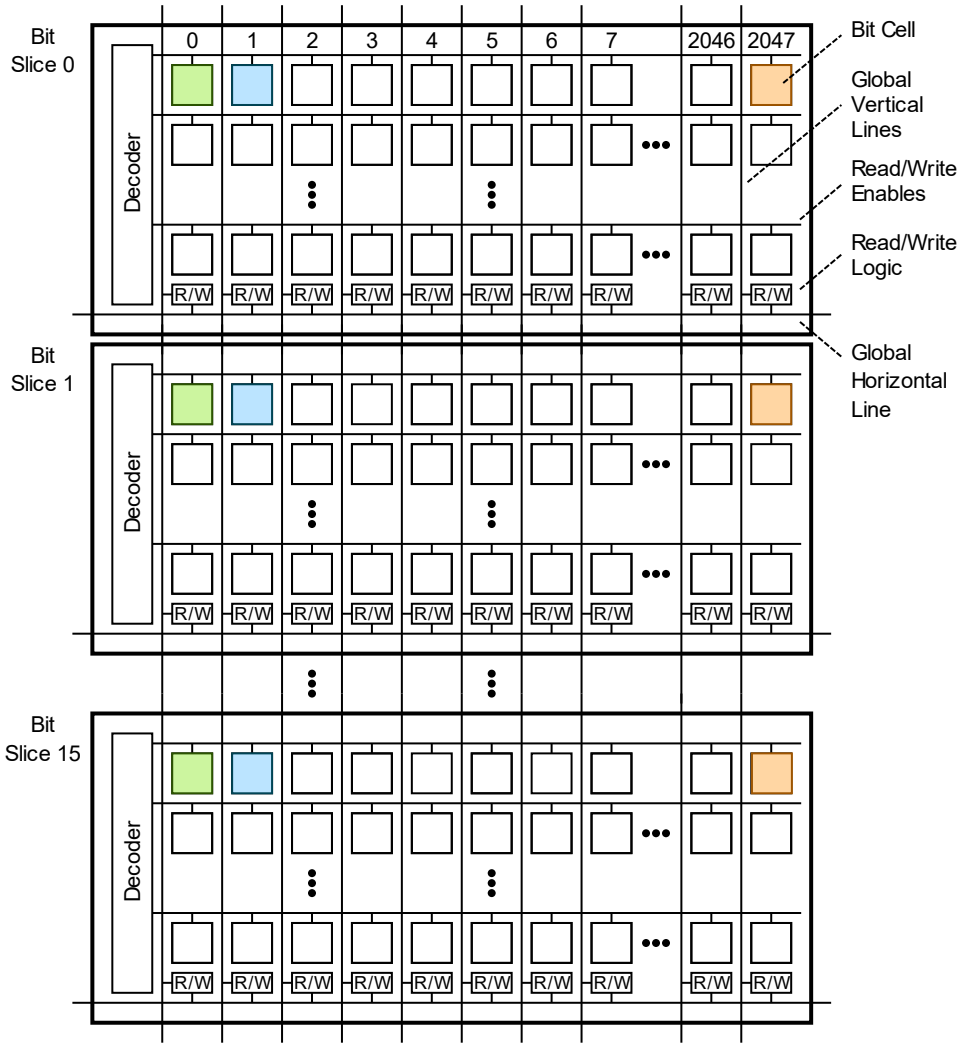
SRAM Bank Physical View



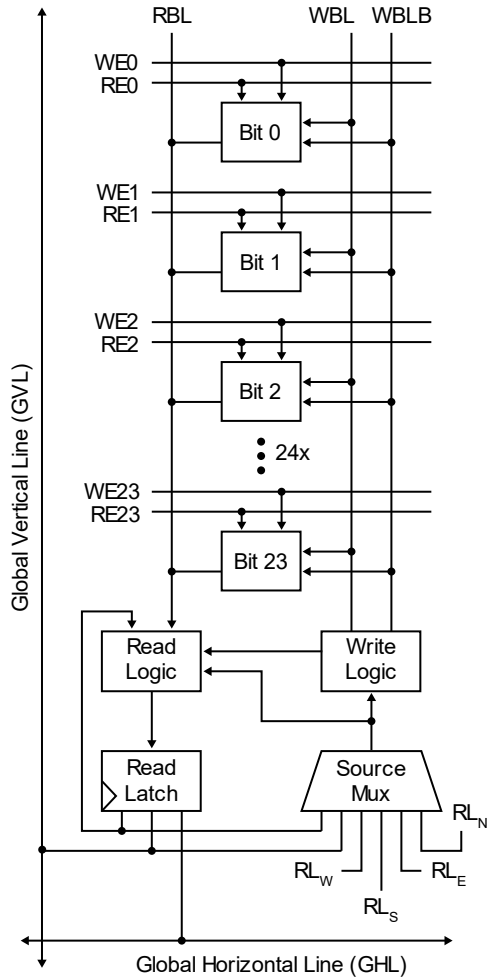
Bit Processor Circuitry

EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  ...
  // ---- bit 1 ----
  // vdst = a ^ b ^ cin
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  (0x0001<<1):  RL ^= RL_N;
  (0x0001<<1):  VRF[vdst] = RL;

  // cout = a*b + b*cin + a*cin
  (0x0001<<1):  RL = VRF[vsrc0, vsrc1];
  (0x0001<<1):  VRF[temp_0] = RL;
  (0x0001<<1):  RL = VRF[vsrc1];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  VRF[temp_1] = RL;
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  ...
```



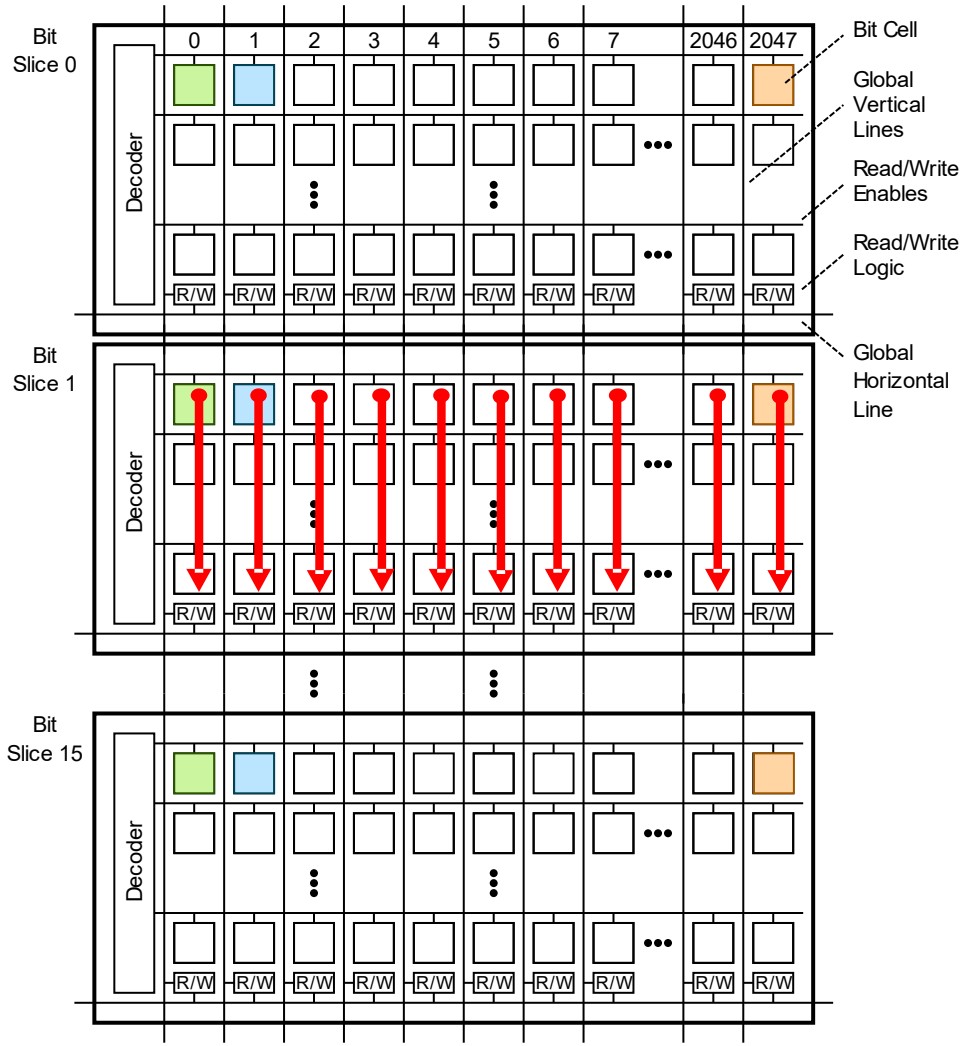
SRAM Bank Physical View



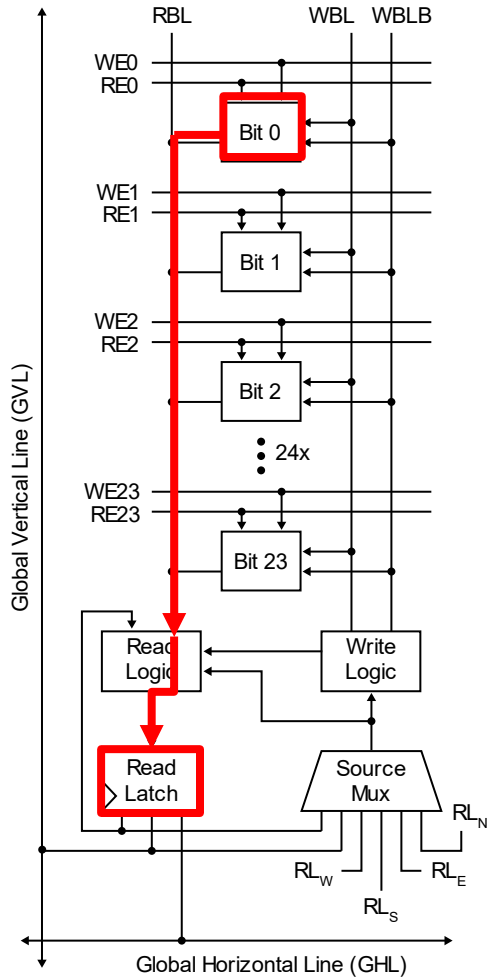
Bit Processor Circuitry

EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  ...
  // ---- bit 1 ----
  // vdst = a ^ b ^ cin
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  (0x0001<<1):  RL ^= RL_N;
  (0x0001<<1):  VRF[vdst] = RL;

  // cout = a*b + b*cin + a*cin
  (0x0001<<1):  RL = VRF[vsrc0, vsrc1];
  (0x0001<<1):  VRF[temp_0] = RL;
  (0x0001<<1):  RL = VRF[vsrc1];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  VRF[temp_1] = RL;
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  ...
```



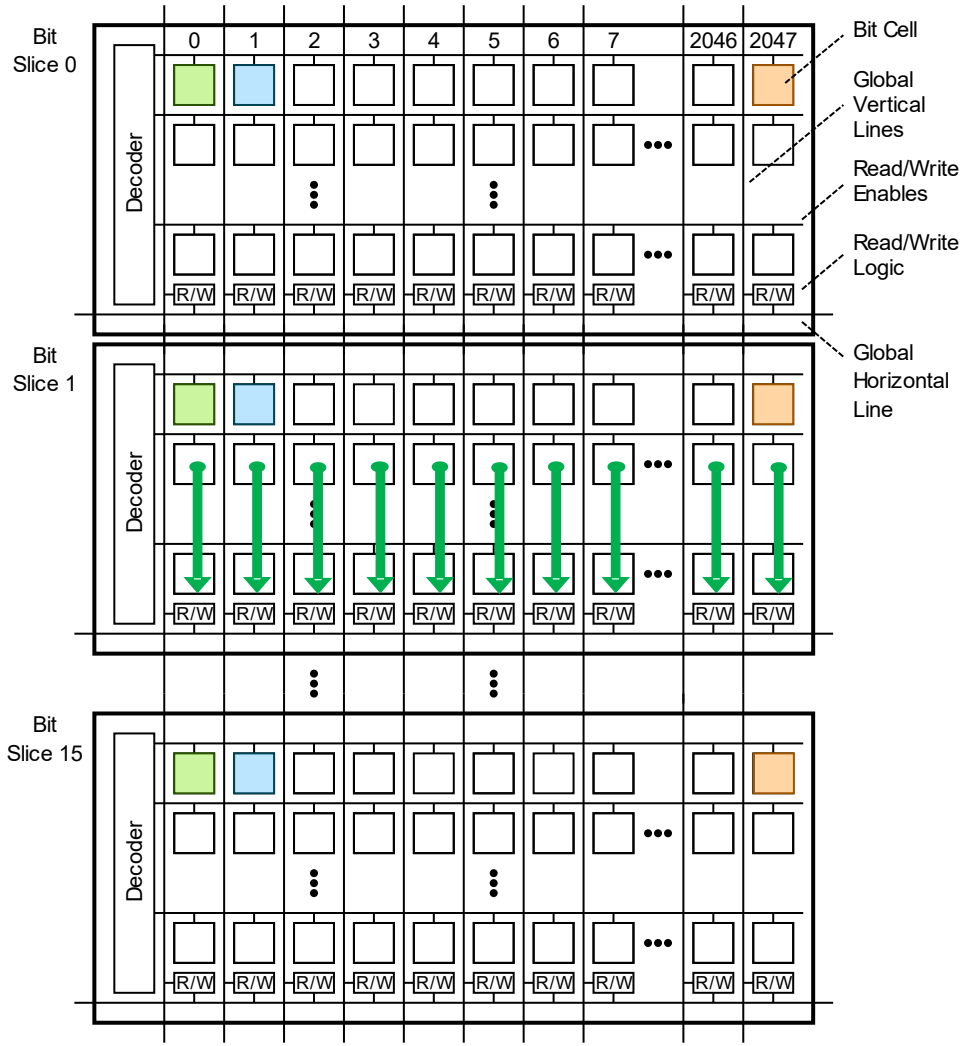
SRAM Bank Physical View



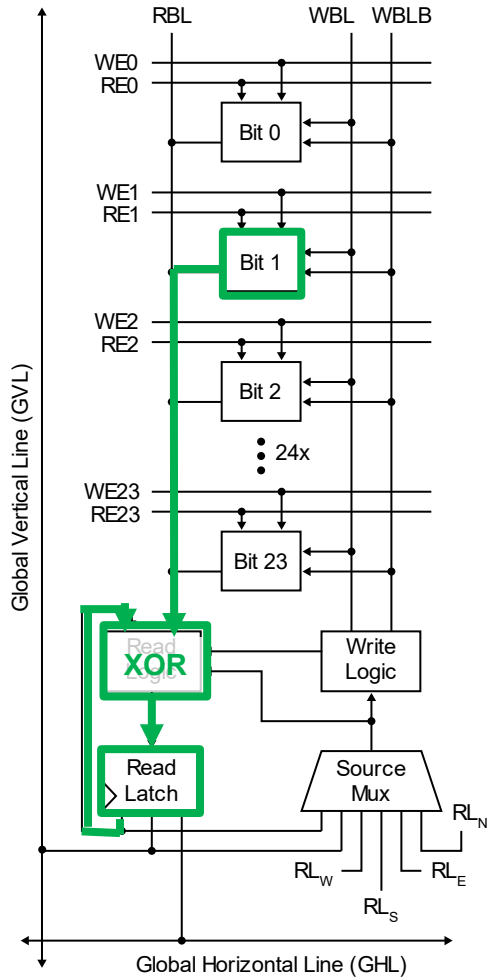
Bit Processor Circuitry

EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  ...
  // ---- bit 1 ----
  // vdst = a ^ b ^ cin
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  (0x0001<<1):  RL ^= RL_N;
  (0x0001<<1):  VRF[vdst] = RL;

  // cout = a*b + b*cin + a*cin
  (0x0001<<1):  RL = VRF[vsrc0, vsrc1];
  (0x0001<<1):  VRF[temp_0] = RL;
  (0x0001<<1):  RL = VRF[vsrc1];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  VRF[temp_1] = RL;
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  ...
```



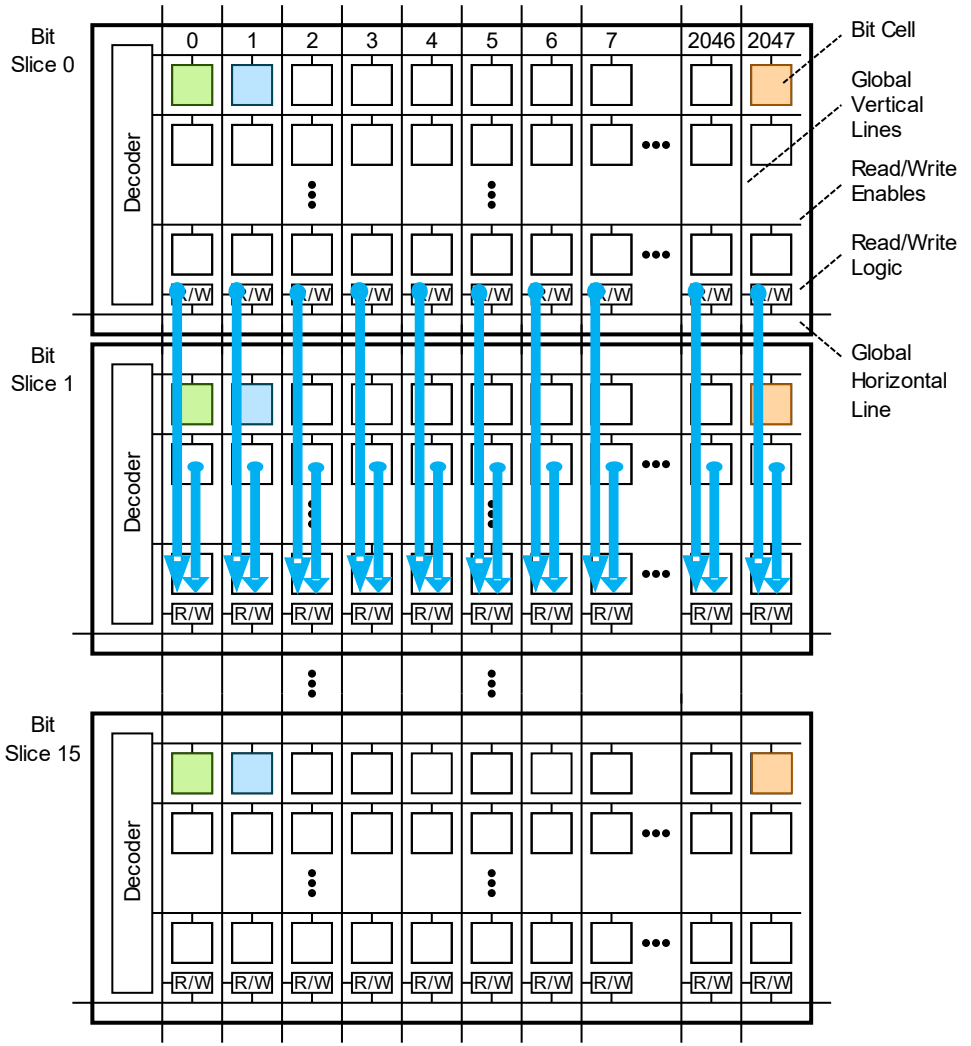
SRAM Bank Physical View



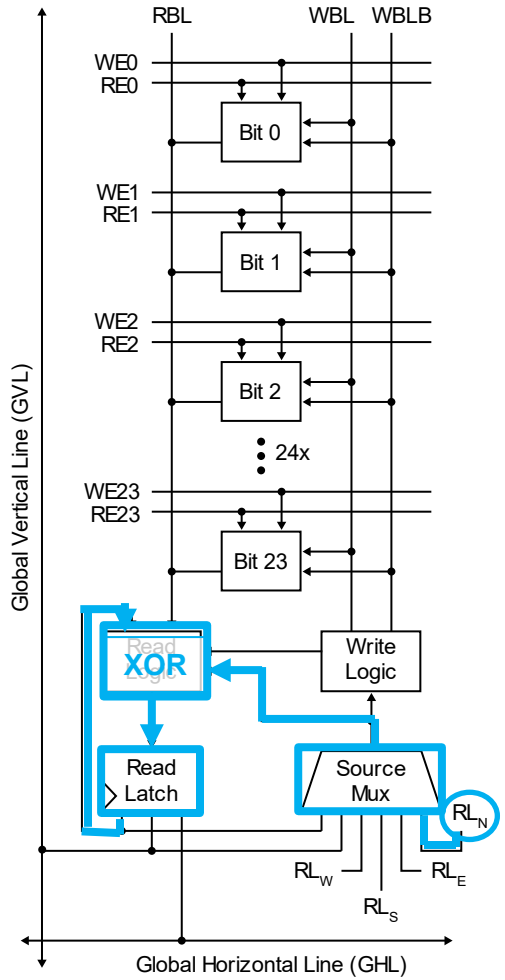
Bit Processor Circuitry

EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  ...
  // ---- bit 1 ----
  // vdst = a ^ b ^ cin
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  (0x0001<<1):  RL ^= RL_N;
  (0x0001<<1):  VRF[vdst] = RL;

  // cout = a*b + b*cin + a*cin
  (0x0001<<1):  RL = VRF[vsrc0, vsrc1];
  (0x0001<<1):  VRF[temp_0] = RL;
  (0x0001<<1):  RL = VRF[vsrc1];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  VRF[temp_1] = RL;
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  ...
```



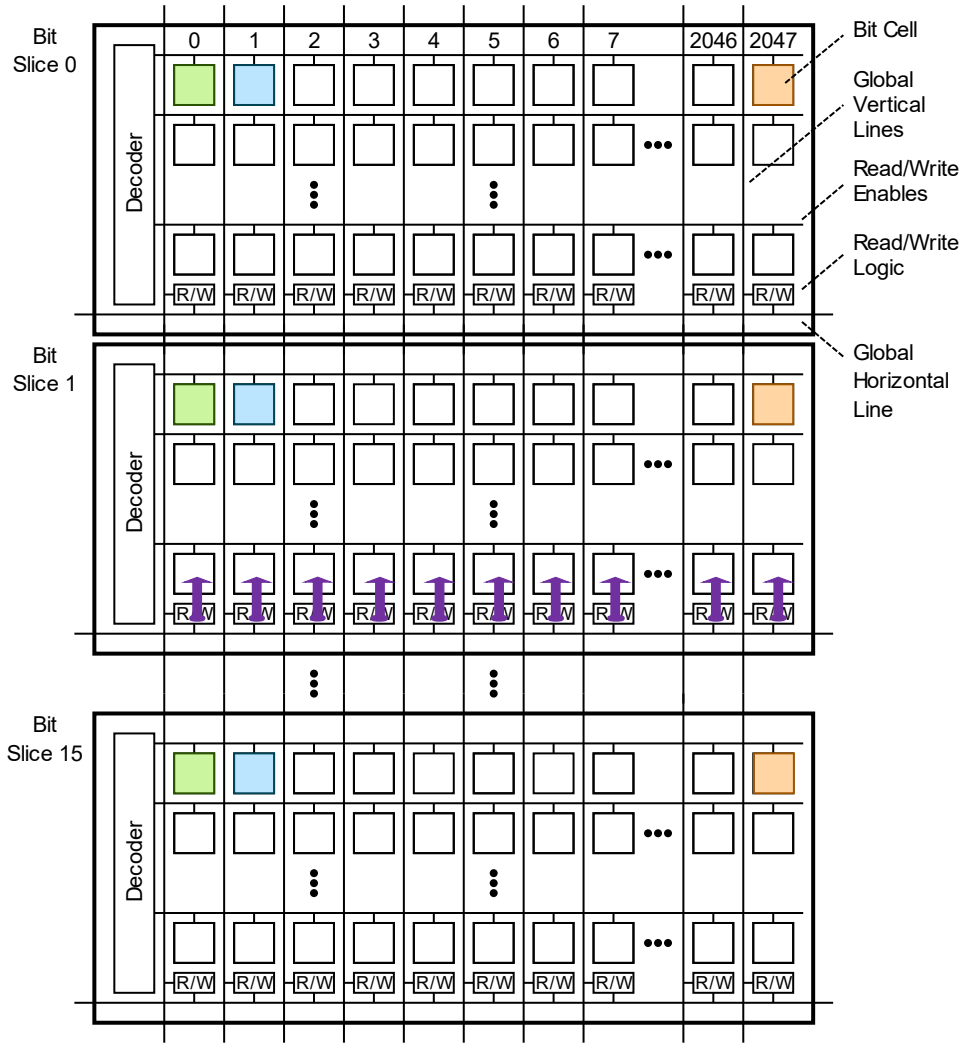
SRAM Bank Physical View



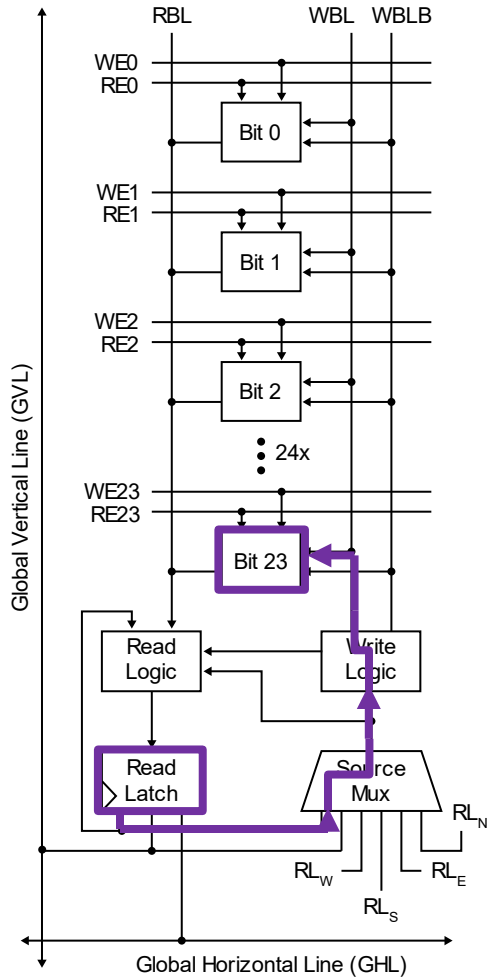
Bit Processor Circuitry

EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  ...
  // ---- bit 1 ----
  // vdst = a ^ b ^ cin
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  (0x0001<<1):  RL ^= RL_N;
  (0x0001<<1):  VRF[vdst] = RL;

  // cout = a*b + b*cin + a*cin
  (0x0001<<1):  RL = VRF[vsrc0, vsrc1];
  (0x0001<<1):  VRF[temp_0] = RL;
  (0x0001<<1):  RL = VRF[vsrc1];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  VRF[temp_1] = RL;
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  ...
```



SRAM Bank Physical View



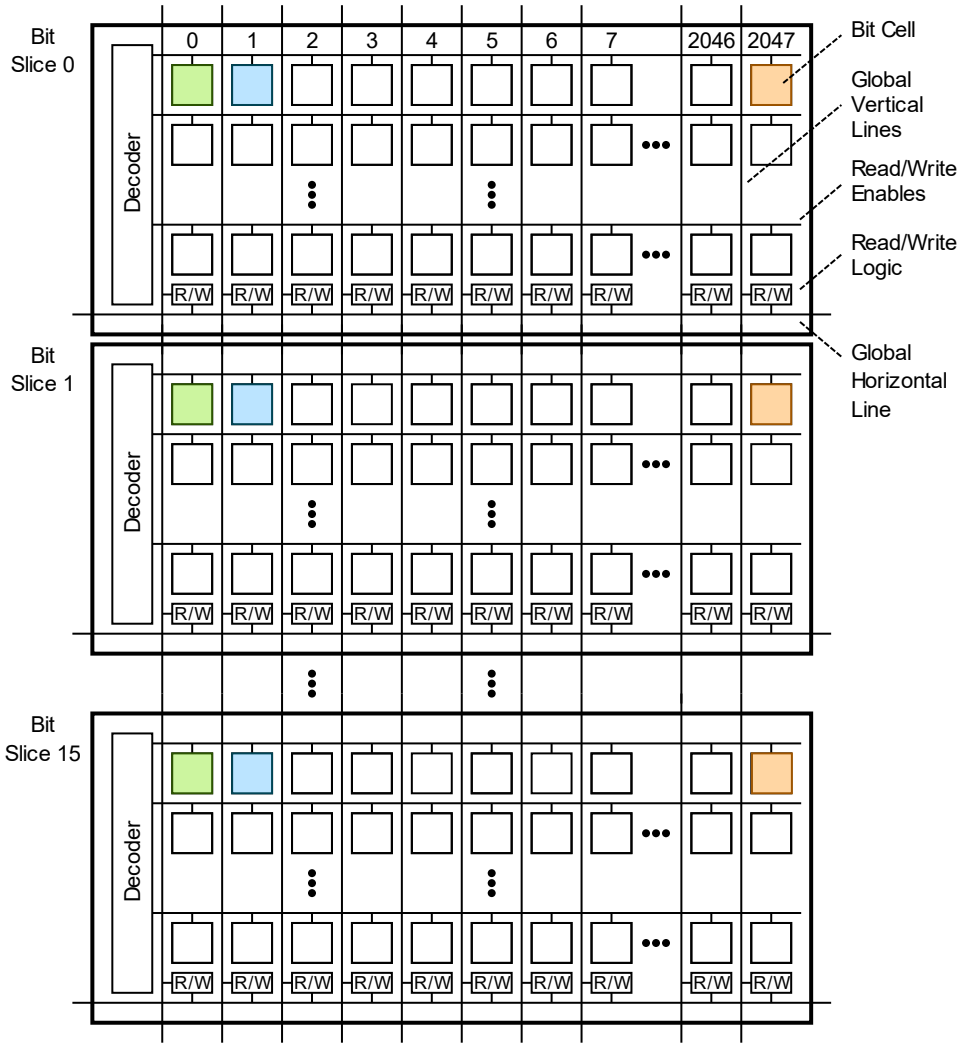
Bit Processor Circuitry

EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

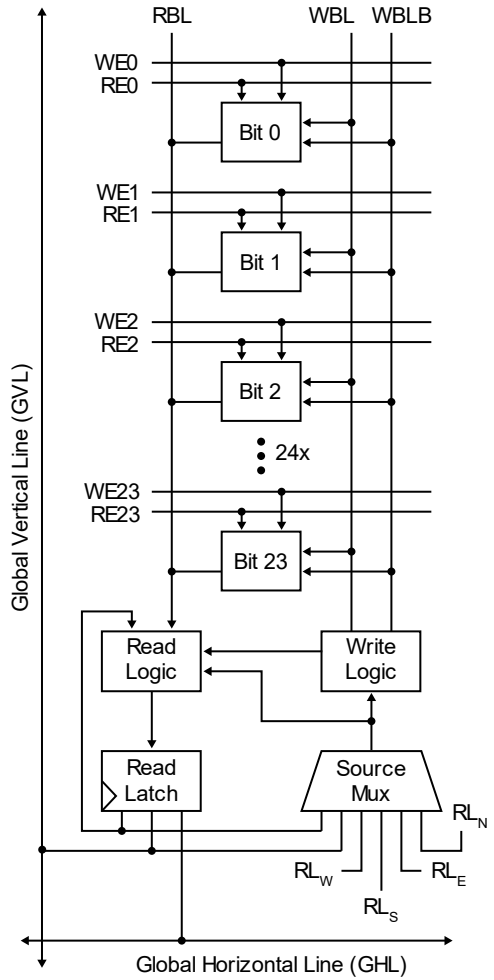
```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  ...
  // ---- bit 1 ----
  // vdst = a ^ b ^ cin
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  (0x0001<<1):  RL ^= RL_N;
  (0x0001<<1):  VRF[vdst] = RL;

  // cout = a*b + b*cin + a*cin
  (0x0001<<1):  RL = VRF[vsrc0, vsrc1];
  (0x0001<<1):  VRF[temp_0] = RL;
  (0x0001<<1):  RL = VRF[vsrc1];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  VRF[temp_1] = RL;
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL &= RL_N;
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  ...
```

Ripple-carry: 215 cycles

Carry-select: 12 cycles



SRAM Bank Physical View



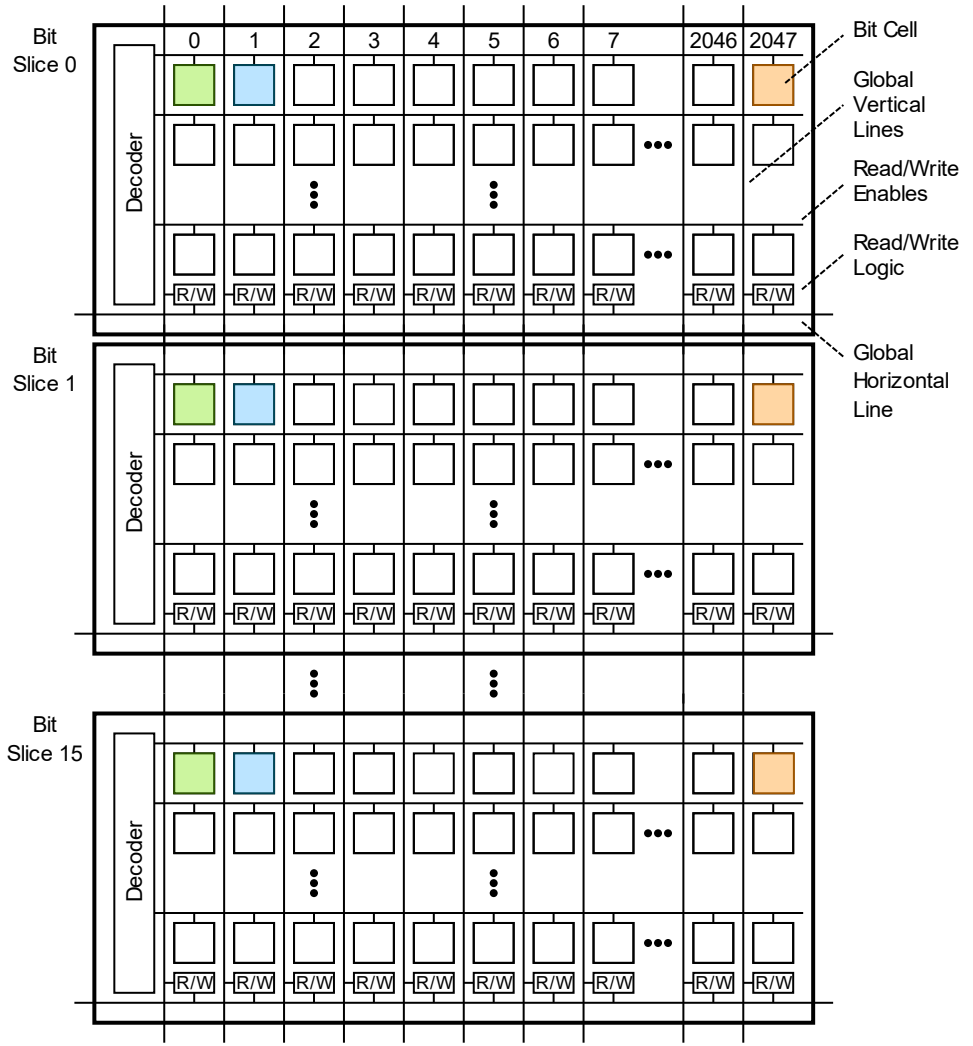
Bit Processor Circuitry

EXAMPLE #3: SEARCH

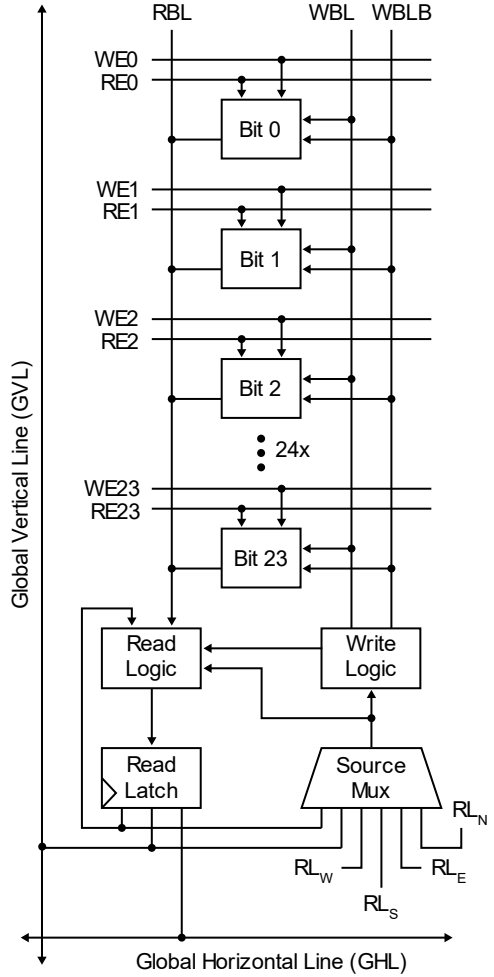
```
APL_FRAG search(vdst, vsrc, key):
  key:       RL = VRF[vsrc];
  ~key:      RL = ~VRF[vsrc];
  0xFFFF:    GVL = RL;
  0xFFFF:    VRF[vdst] = GVL;
```

Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0



SRAM Bank Physical View



Bit Processor Circuitry

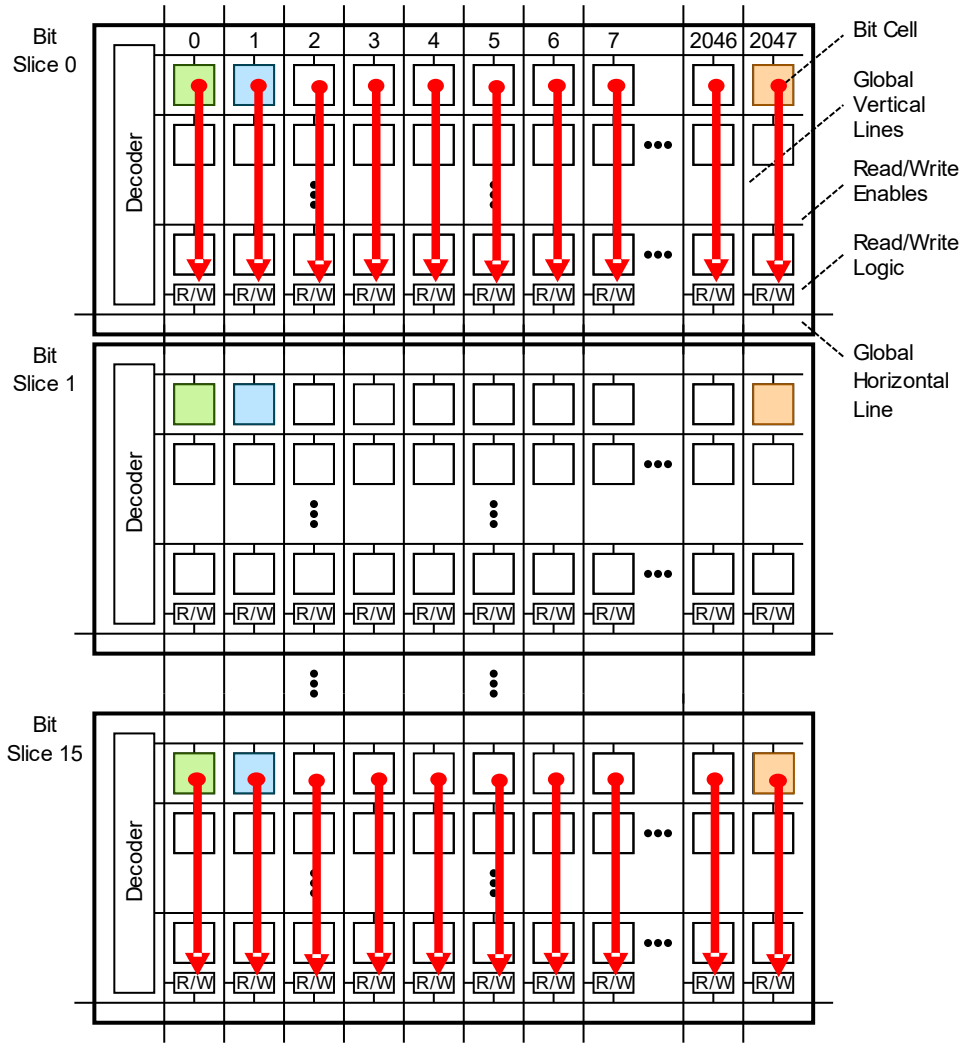
EXAMPLE #3: SEARCH

```
APL_FRAG search(vdst, vsrc, key):
  key:        RL = VRF[vsrc];
  ~key:       RL = ~VRF[vsrc];
  0xFFFF:     GVL = RL;
  0xFFFF:     VRF[vdst] = GVL;
```

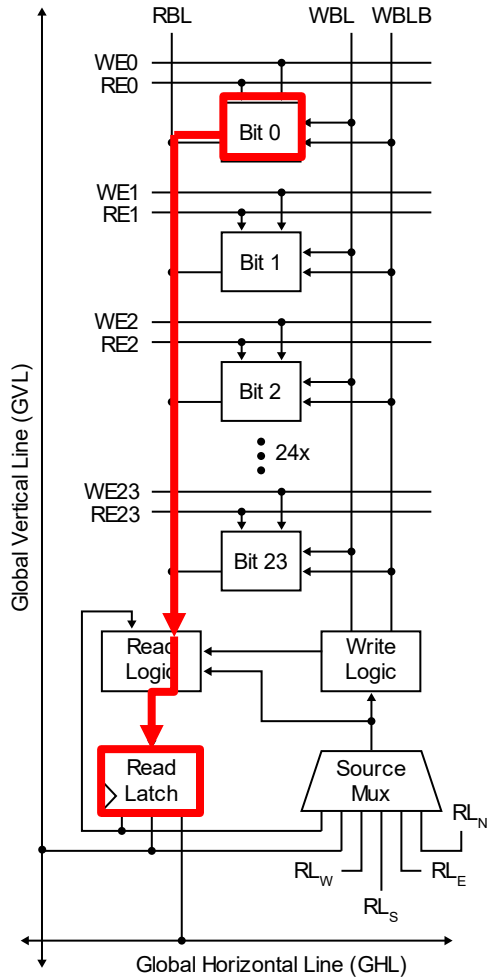
Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0

Result: 1 0 0 0 1 0 1 0



SRAM Bank Physical View



Bit Processor Circuitry

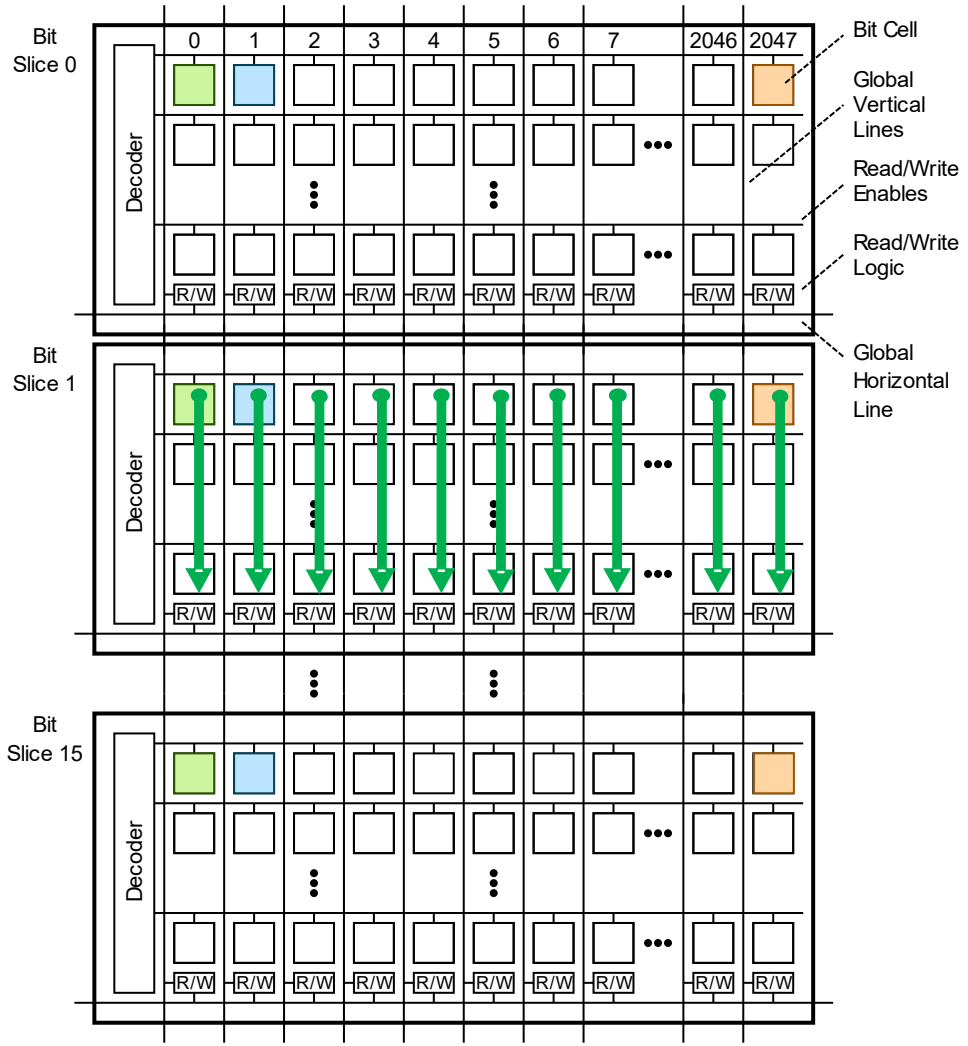
EXAMPLE #3: SEARCH

```
APL_FRAG search(vdst, vsrc, key):
  key:       RL = VRF[vsrc];
  ~key:      RL = ~VRF[vsrc];
  0xFFFF:    GVL = RL;
  0xFFFF:    VRF[vdst] = GVL;
```

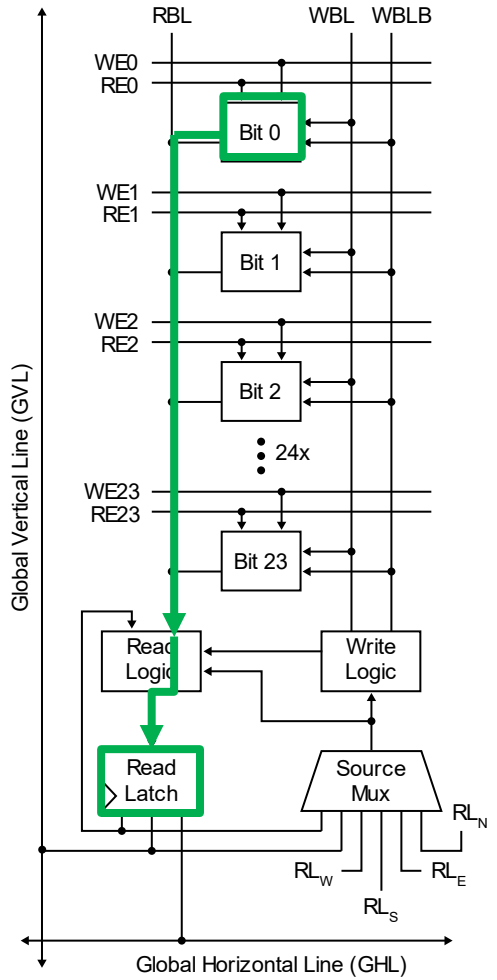
Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0

Result: 1 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0



SRAM Bank Physical View



Bit Processor Circuitry

EXAMPLE #3: SEARCH

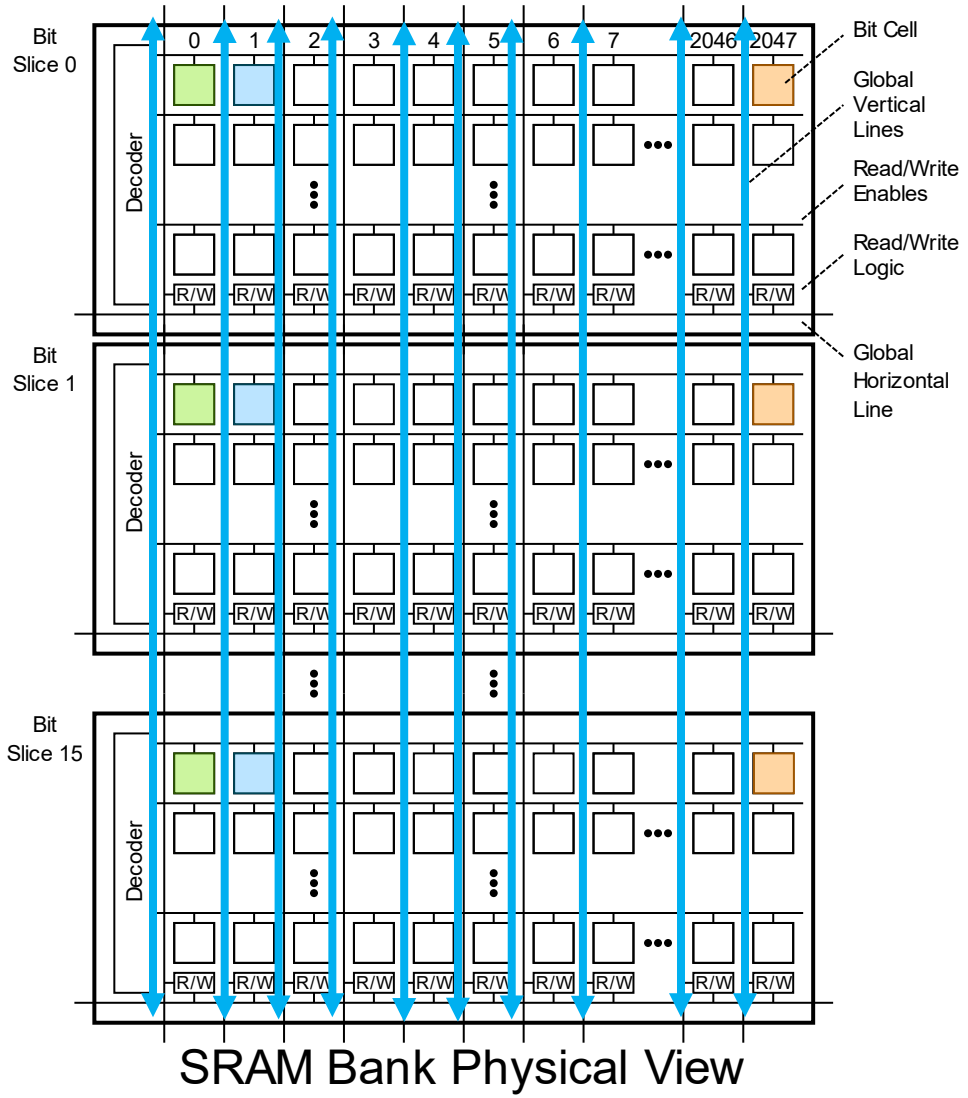
```
APL_FRAG search(vdst, vsrc, key):
  key:       RL = VRF[vsrc];
  ~key:      RL = ~VRF[vsrc];
  0xFFFF:    GVL = RL;
  0xFFFF:    VRF[vdst] = GVL;
```

Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

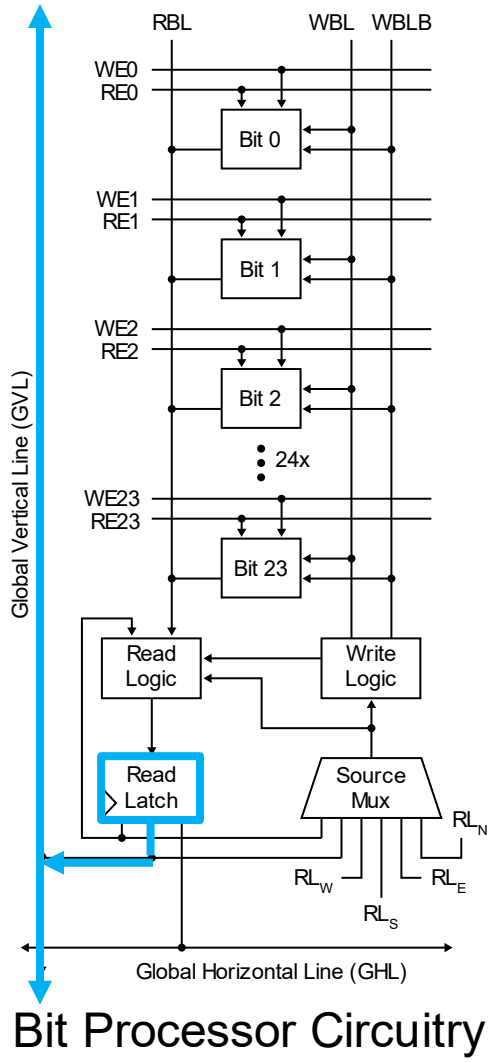
Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0

Result: 1 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0

AND



SRAM Bank Physical View



Bit Processor Circuitry

EXAMPLE #3: SEARCH

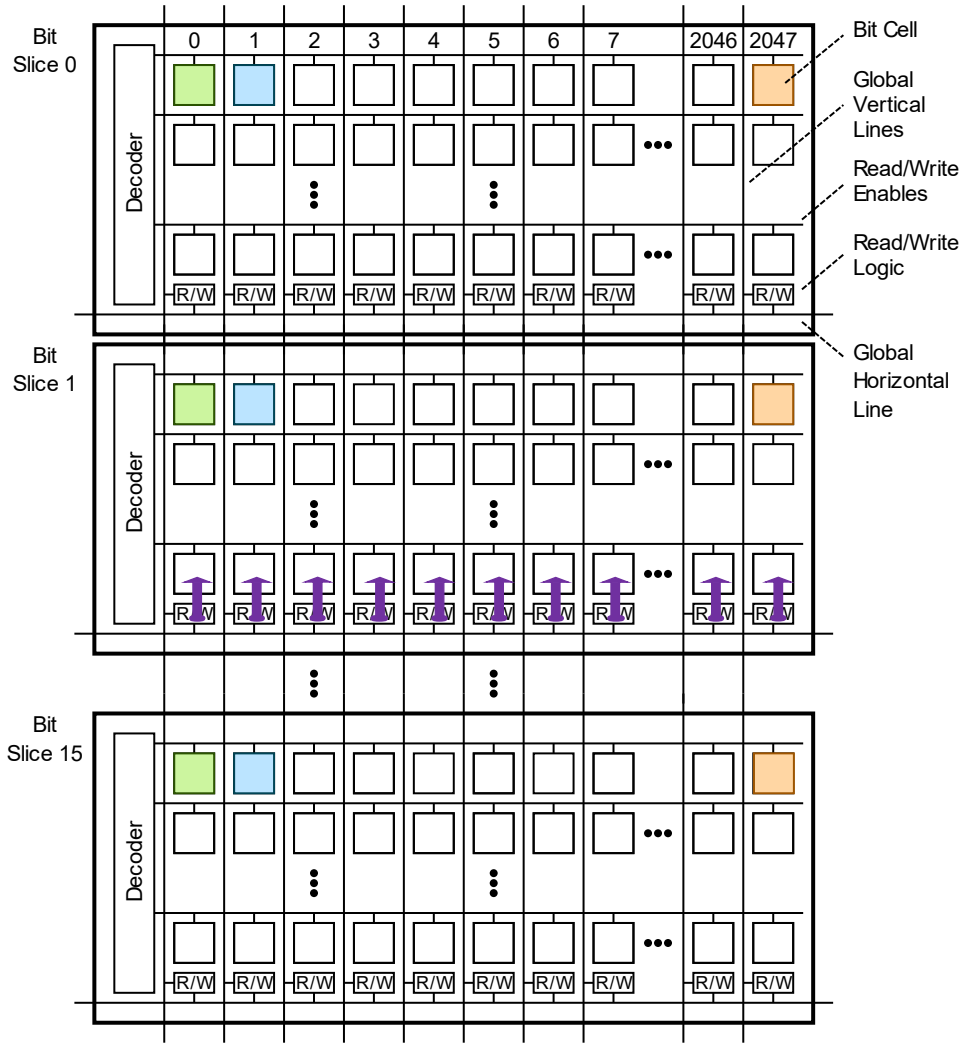
```
APL_FRAG search(vdst, vsrc, key):
  key:       RL = VRF[vsrc];
  ~key:      RL = ~VRF[vsrc];
  0xFFFF:    GVL = RL;
  0xFFFF:    VRF[vdst] = GVL;
```

Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

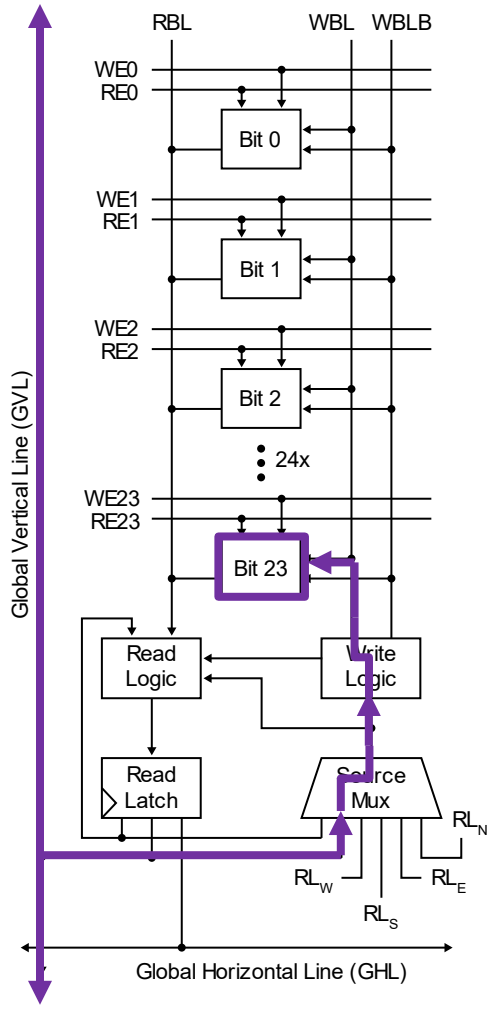
Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0

Result: 1 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0

AND = 0



SRAM Bank Physical View



Bit Processor Circuitry

EXAMPLE #3: SEARCH

```
APL_FRAG search(vdst, vsrc, key):
  key:       RL = VRF[vsrc];
  ~key:      RL = ~VRF[vsrc];
  0xFFFF:    GVL = RL;
  0xFFFF:    VRF[vdst] = GVL;
```

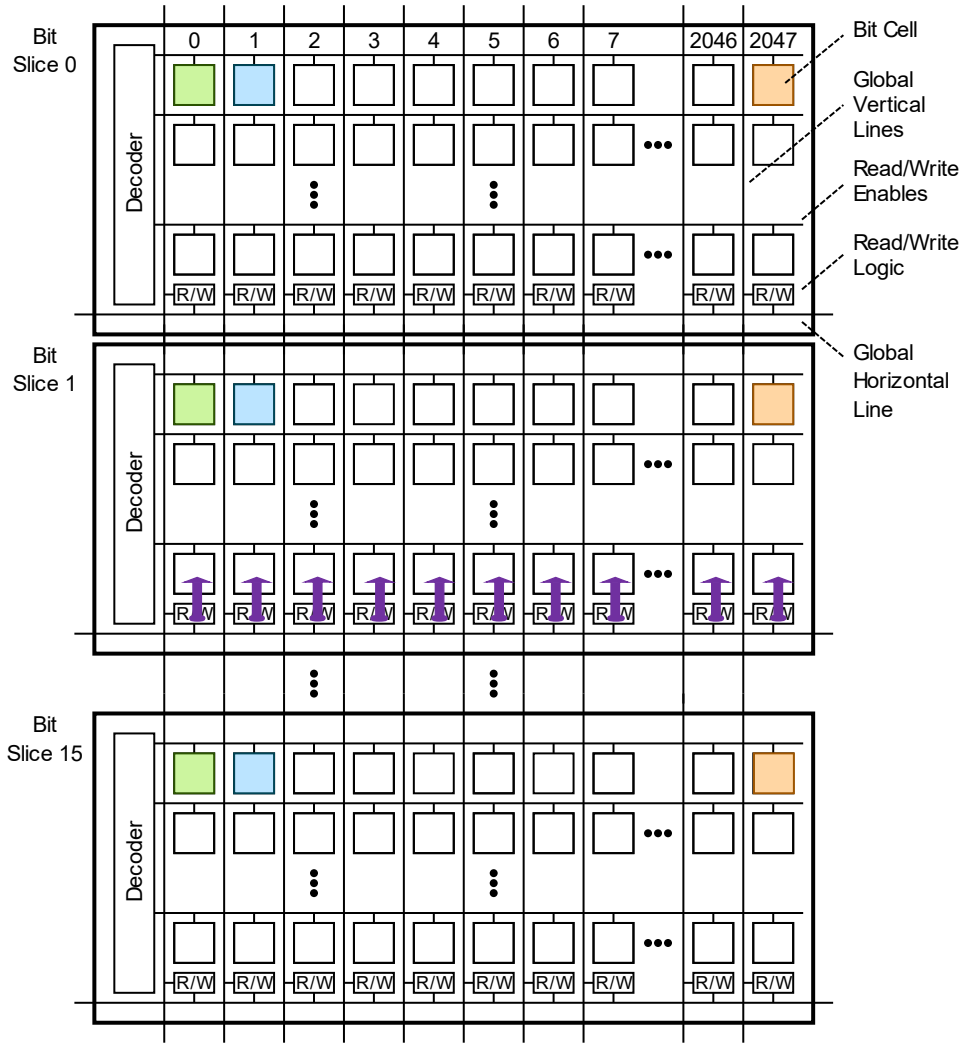
Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0

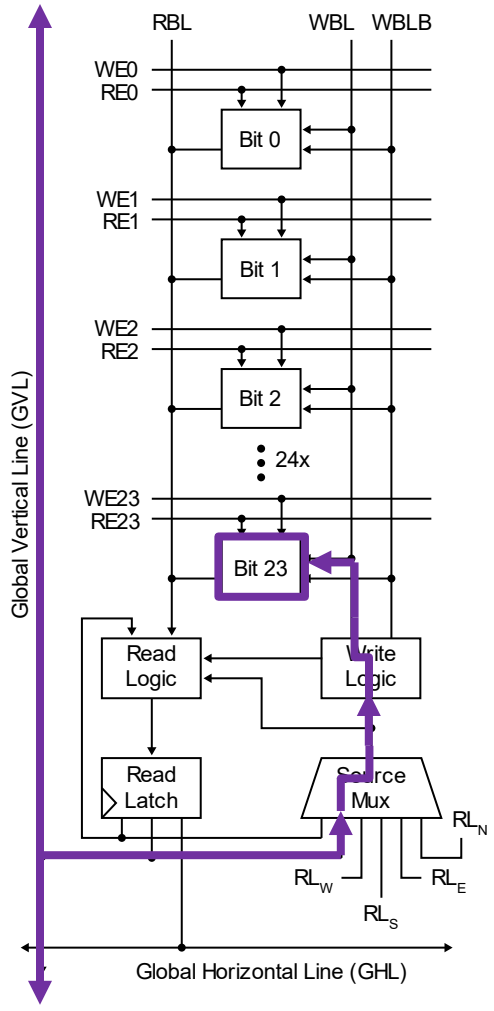
Result: 1 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0

AND = 0

- Architecture doesn't directly support ~VRF[] operation
- More compact mask encodings are possible



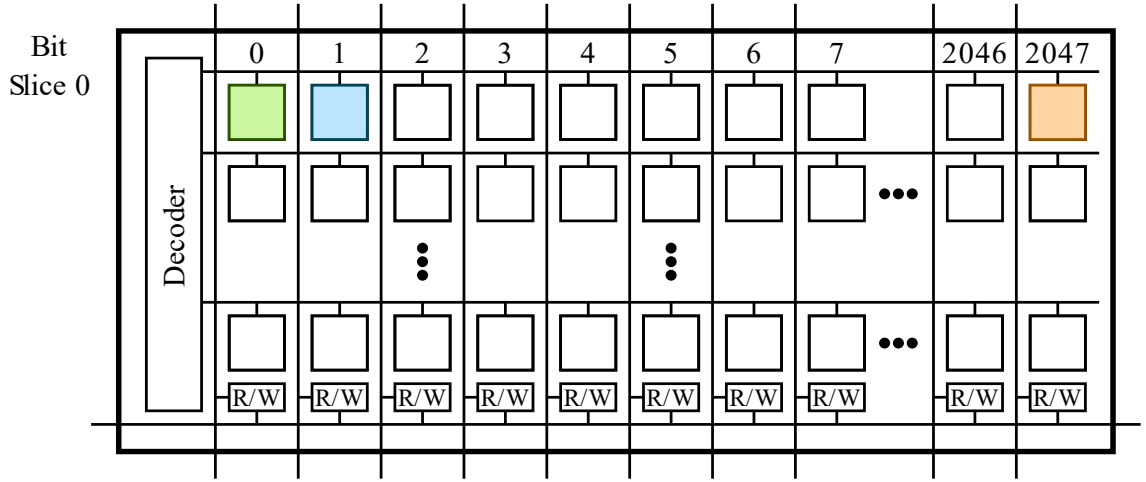
SRAM Bank Physical View



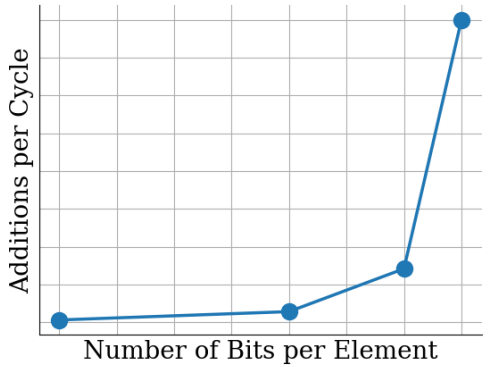
Bit Processor Circuitry

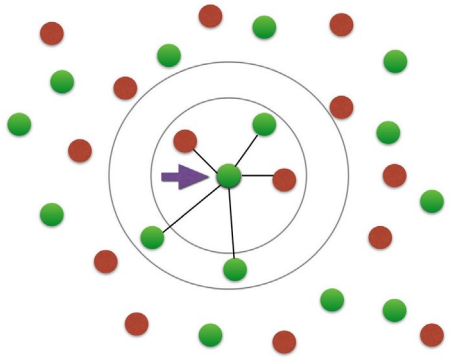
Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

- Motivation
- APU Microarchitecture
- APU Microcoding
- **Virtual Vector Instruction Set**
- Microbenchmarking Results




```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```





RISC-V Virtual Vector Instruction Set

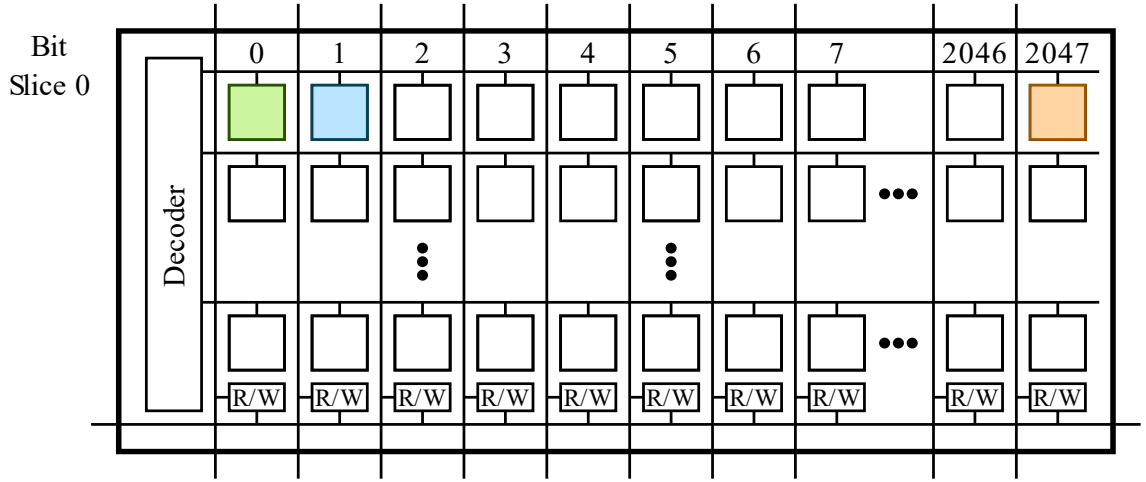
Instruction	Description	Operation Type				
		Element-Wise	Cross-Element	Mask	Memory	Configuration
`vor_vv`	bit-wise or	✓				
`vadd_vv`	element-wise addition	✓				
`vsub_vx`	vector-scalar subtraction	✓				
`vmul_vv`	element-wise multiplication	✓				
`vsll`	bitwise left-shift logical	✓				
`vsrl`	bitwise right-shift logical	✓				
`vmv_vv`	copy vector	✓				
`vmseq_vx`	set-if-equal		✓			
`vmseq_vx_m`	set-if-equal (masked)		✓	✓		
`vredmin_vs`	find min element		✓			
`vredmin_vs_m`	find min element (masked)		✓	✓		
`vmv_vx`	broadcast scalar to vector		✓			
`vmv_xs`	extract element 0 of vector		✓			
`vfirst`	index of first set mask bit		✓	✓		
`vmsof`	set only first set mask bit		✓	✓		
`vmset`	set all elements of mask		✓	✓		
`vmnot`	invert mask	✓		✓		
`vmand`	bitwise-and of two masks	✓		✓		
`vmv_m`	copy mask	✓		✓		
`vload`	loads vector from DRAM				✓	
`vstore`	stores vector to DRAM				✓	
`vsetvl`	set application vector length					✓

Limitations:

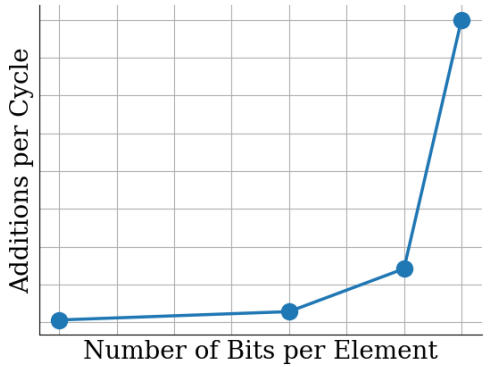
- Unsigned 16-bit integers
- No floating-point support
- Vector lengths up to 32,768 elements
- Assumes source operands are distinct
- Behavior on elements past the specified vector length is undefined

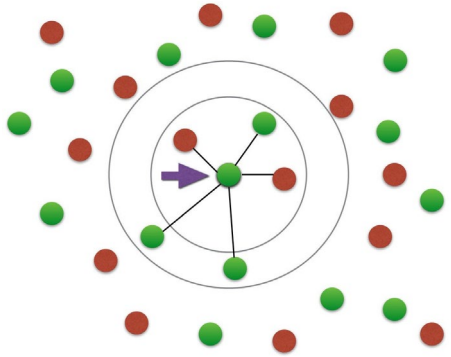
Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

- Motivation
- APU Microarchitecture
- APU Microcoding
- Virtual Vector Instruction Set
- **Microbenchmarking Results**




```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```



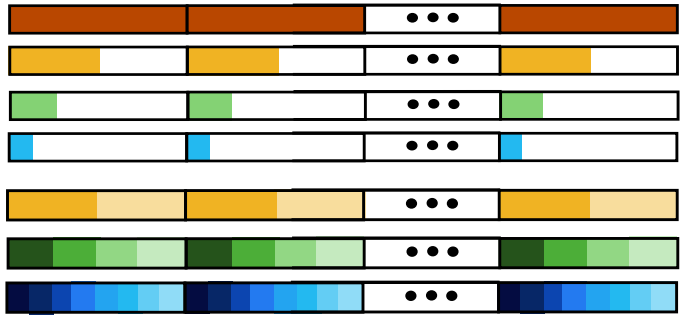


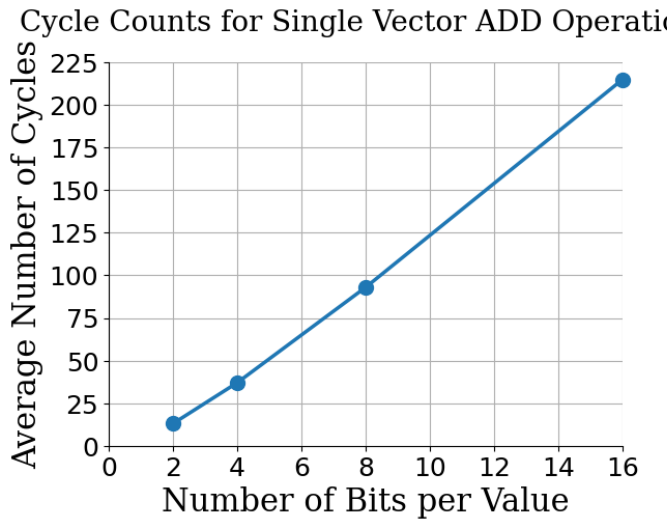
MICROBENCHMARKING NARROW-BITWIDTH VECTOR OPERATIONS

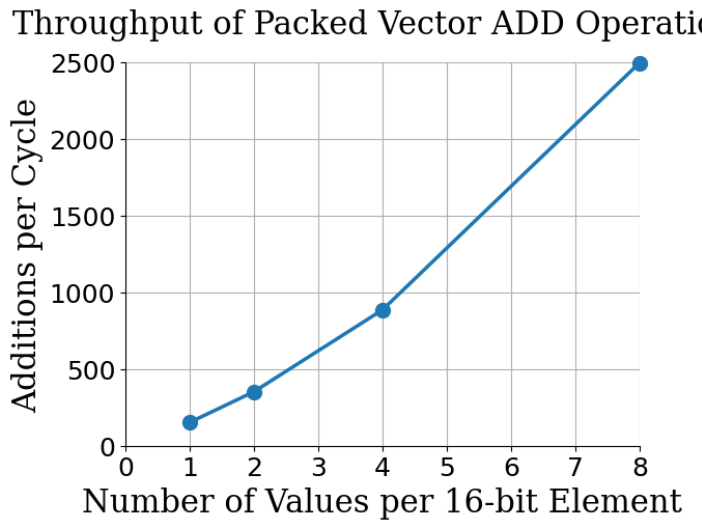
```
APL_FRAG _frag_vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  0x0001:       RL = VRF[vsrc0];
  0x0001:       RL ^= VRF[vsrc1];
  0x0001:       VRF[vdst] = RL;
  0x0001:       RL = VRF[vsrc0, vsrc1];
  // ---- bit 1 ----
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  ...
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  // ---- bit 2 ----
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  ...
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
  // ---- bit 3 ----
  (0x0001<<1):  RL = VRF[vsrc0];
  (0x0001<<1):  RL ^= VRF[vsrc1];
  ...
  (0x0001<<1):  RL |= VRF[temp_0];
  (0x0001<<1):  RL |= VRF[temp_1];
...
```

vadd

packed operation	avg # cycles	ops/cycle
one 16-bit ADD	215	152
one 8-bit ADD	93	352
one 4-bit ADD	37	886
one 2-bit ADD	13	2497
two 8-bit ADDs	93	705
four 4-bit ADDs	37	3542
eight 2-bit ADDs	13	20010

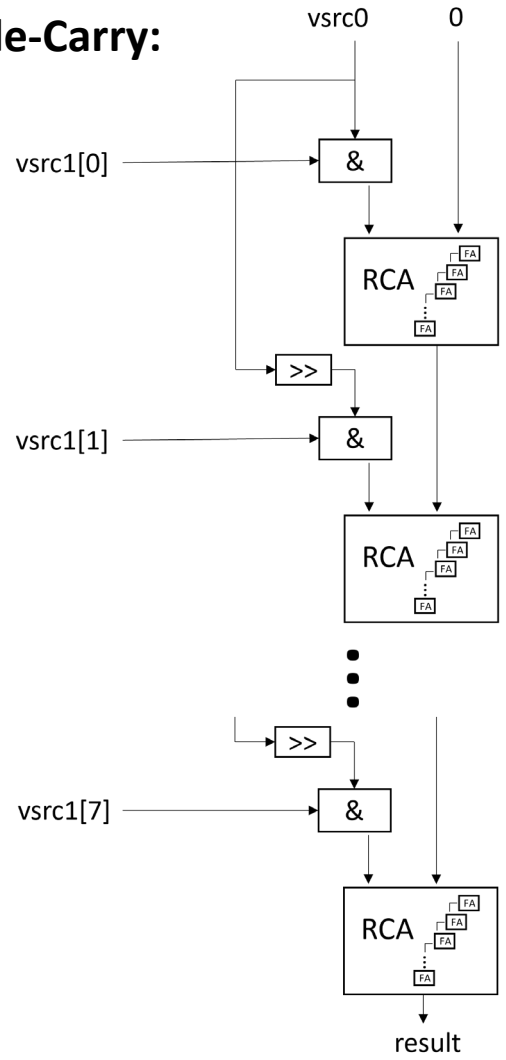




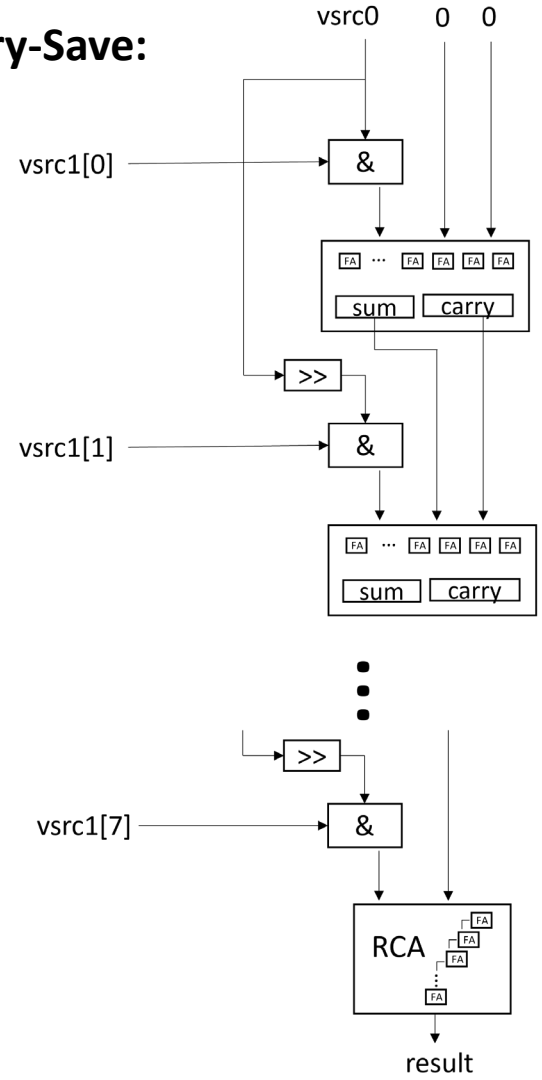


Microbenchmarking Vector Multiply Instructions

Ripple-Carry:



Carry-Save:

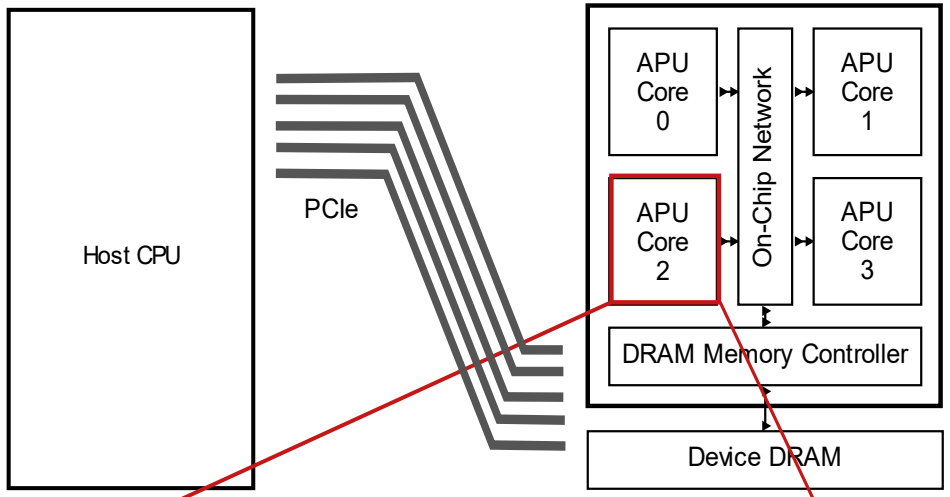



```
For each bit:
    S_out = (X * Y_i) ^ S_in ^ C_in
    C_out = (X * Y_i) * C_in
                      + S_in * C_in
                      + X * Y_i * S_in
```

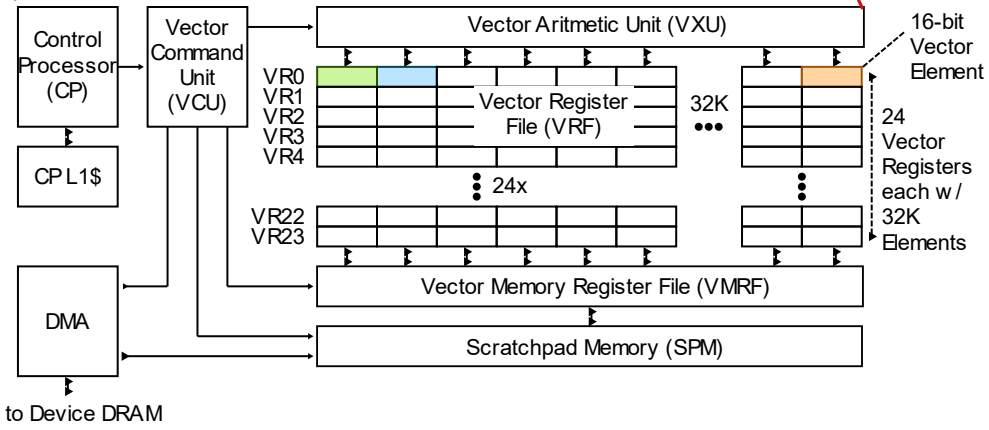
Ripple-carry: 3540 cycles
Carry-save: 438 cycles

8.3x speedup

Microbenchmarking Data Movement and Fragment Overhead



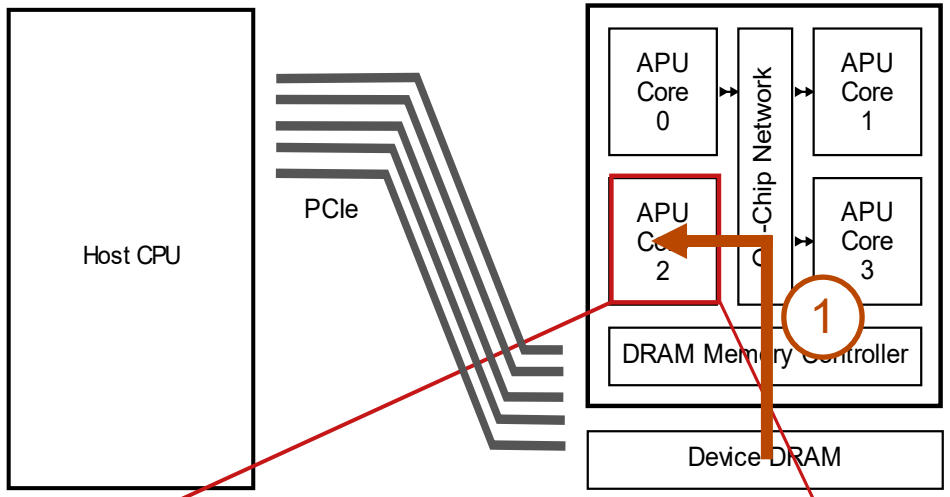
System Overview



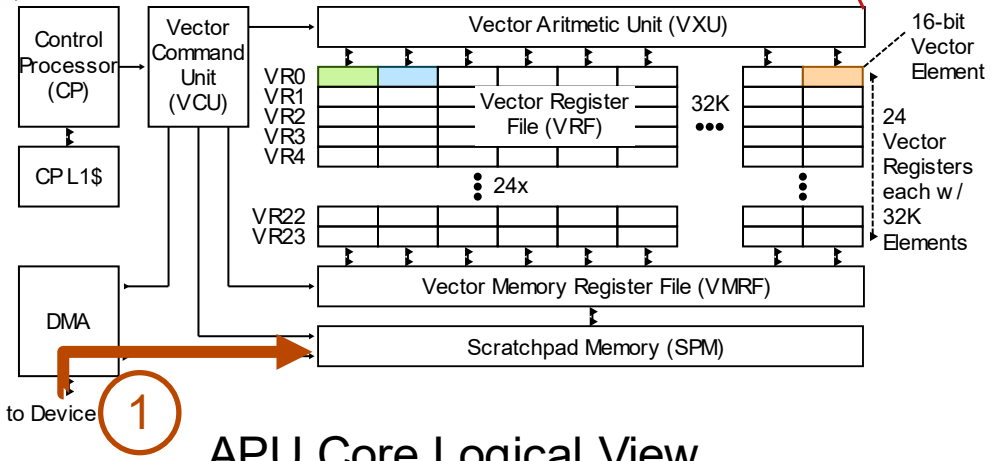
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle

Microbenchmarking Data Movement and Fragment Overhead



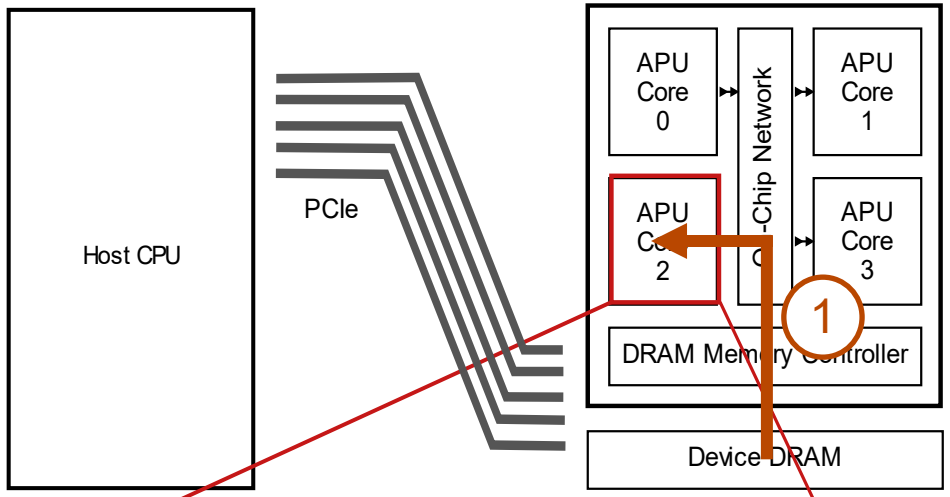
System Overview



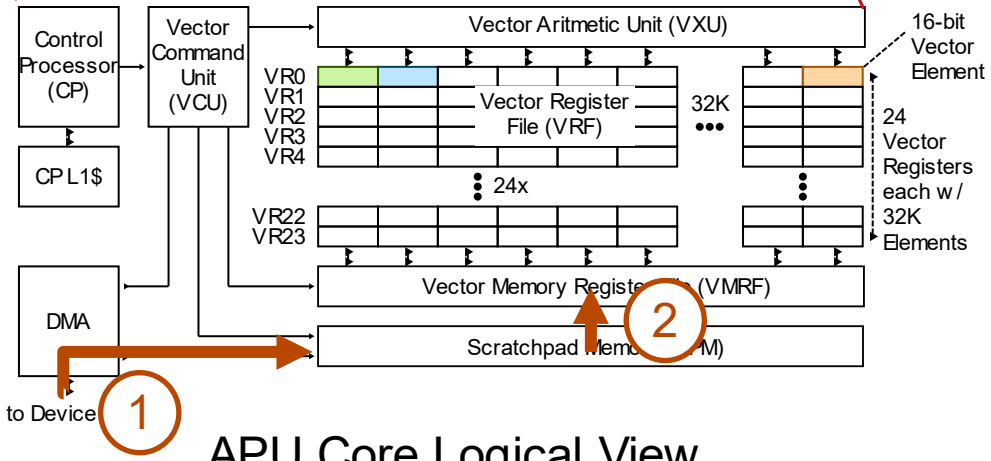
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle

MICROBENCHMARKING DATA MOVEMENT AND FRAGMENT OVERHEAD



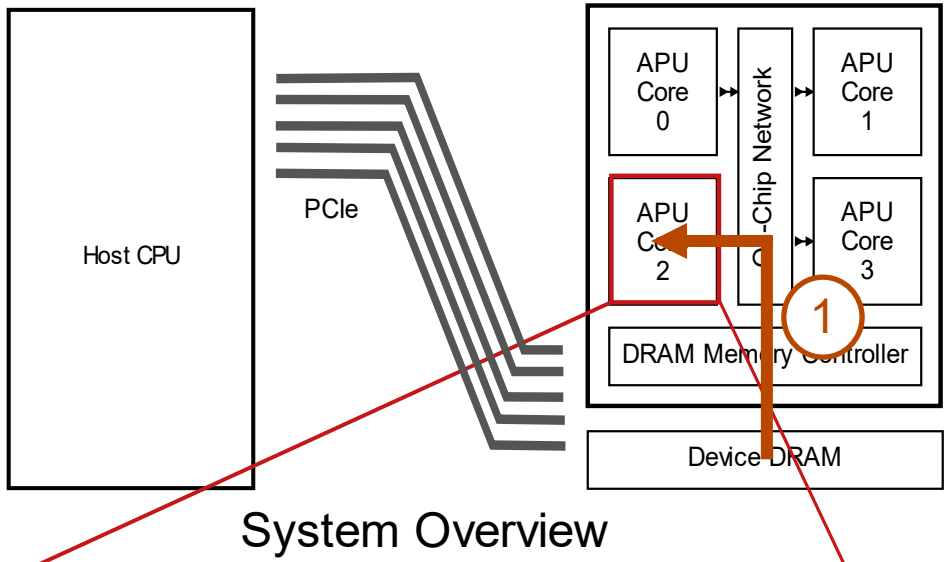
System Overview



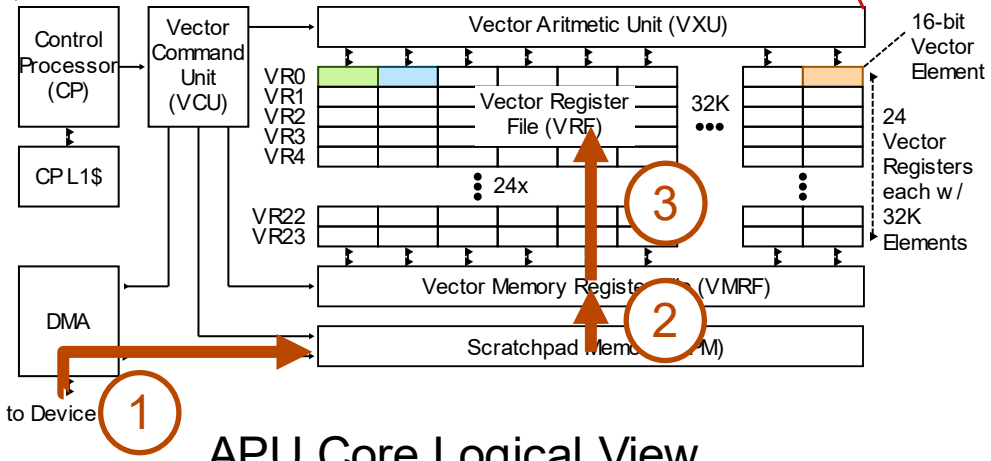
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle

Microbenchmarking Data Movement and Fragment Overhead



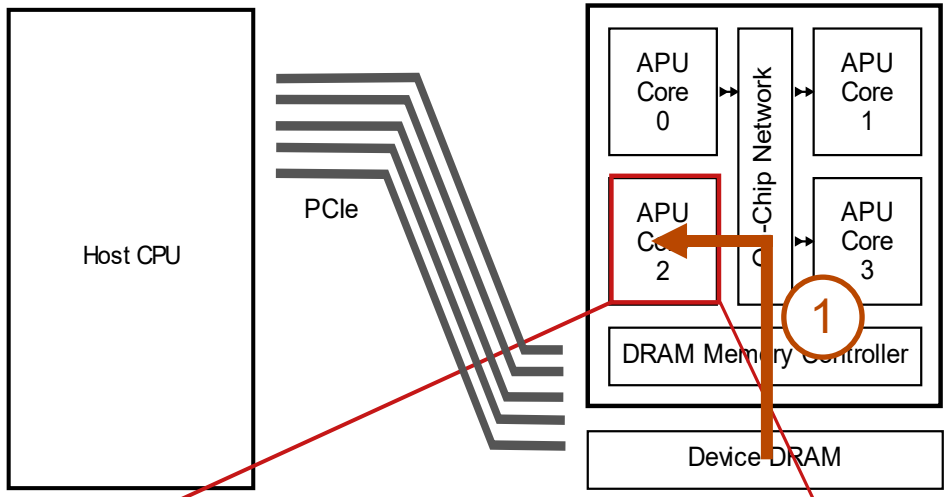
System Overview



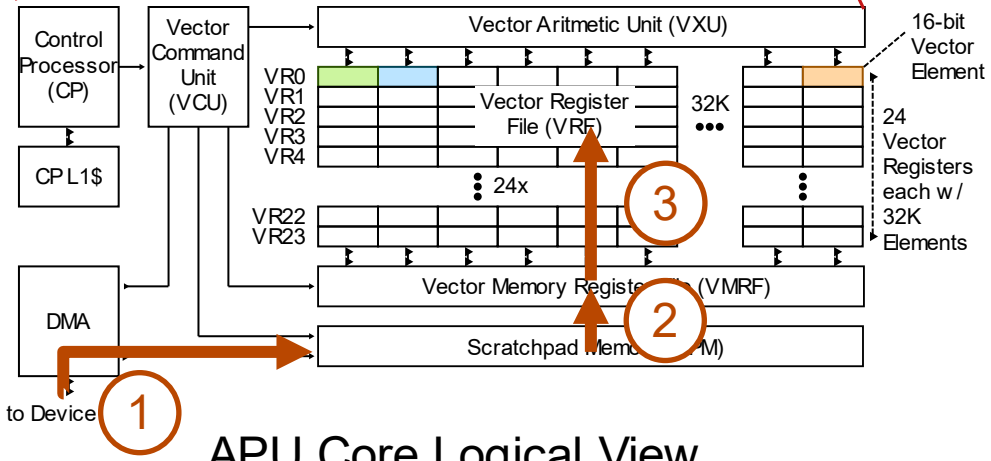
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle

Microbenchmarking Data Movement and Fragment Overhead



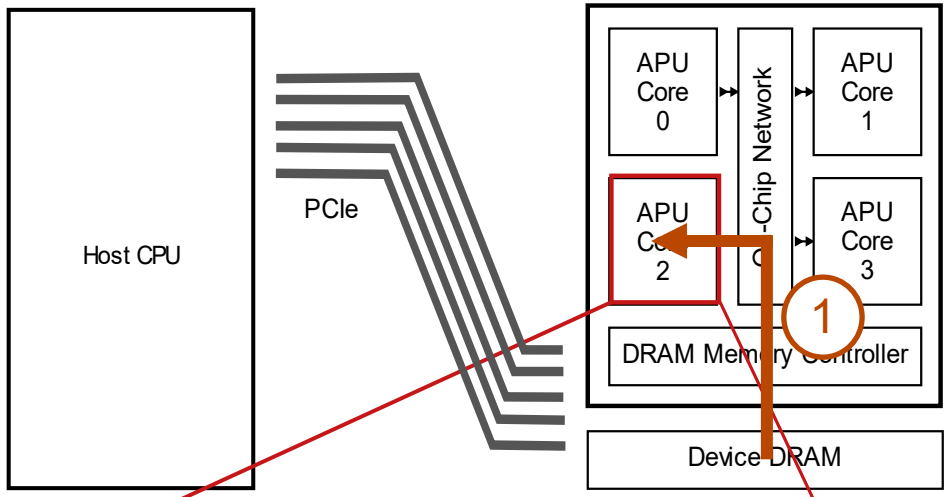
System Overview



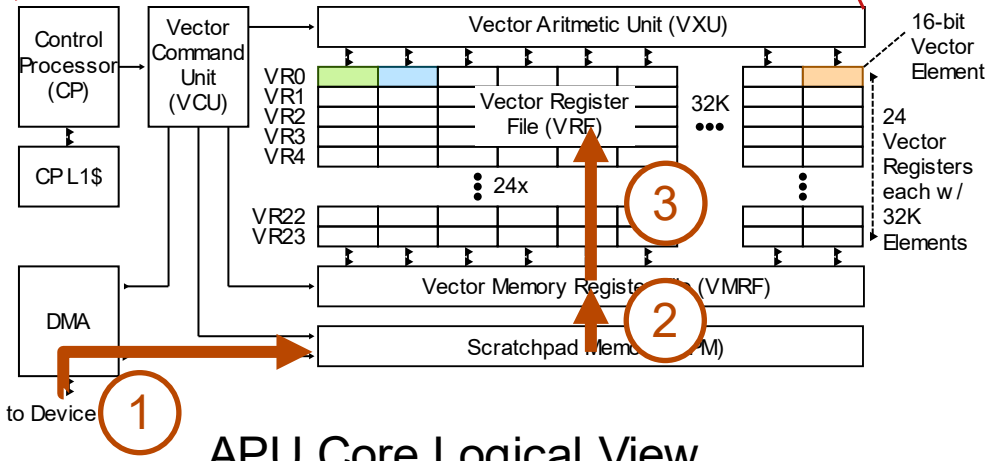
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle
Total	22,066	1.4

Microbenchmarking Data Movement and Fragment Overhead



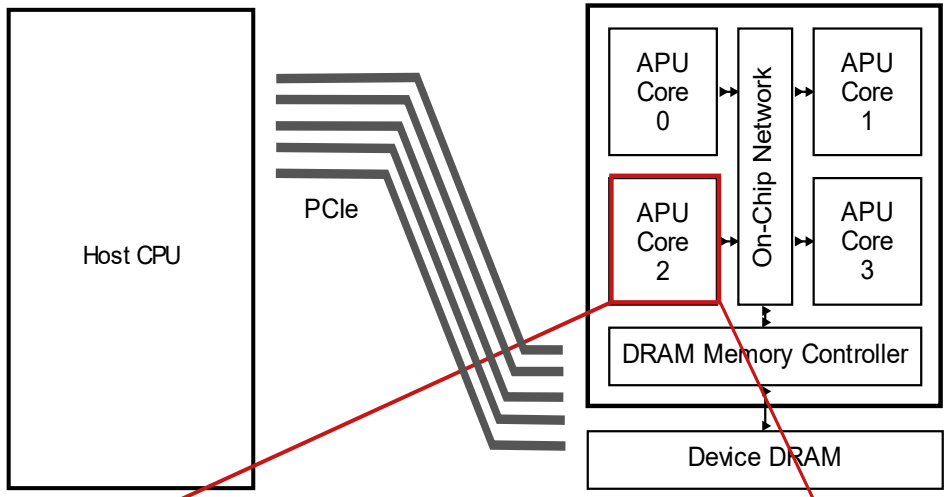
System Overview



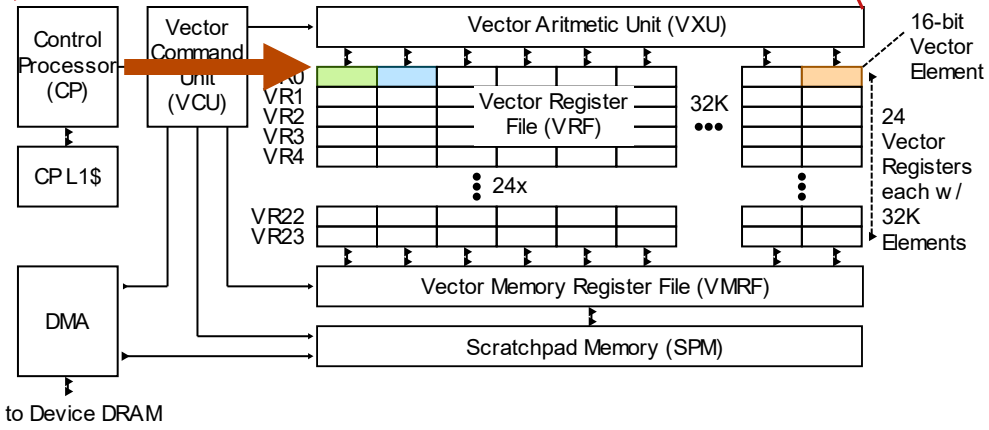
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle
DRAM → SPM	21,575	1.5
SPM → VMRF	470	69.7
VMRF → VRF	21	1568.9
Total	22,066	1.4

Microbenchmarking Data Movement and Fragment Overhead



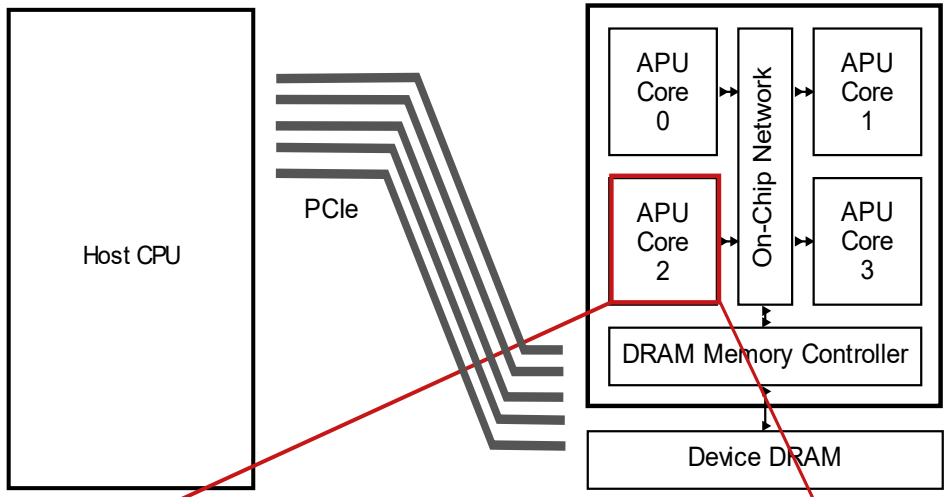
System Overview



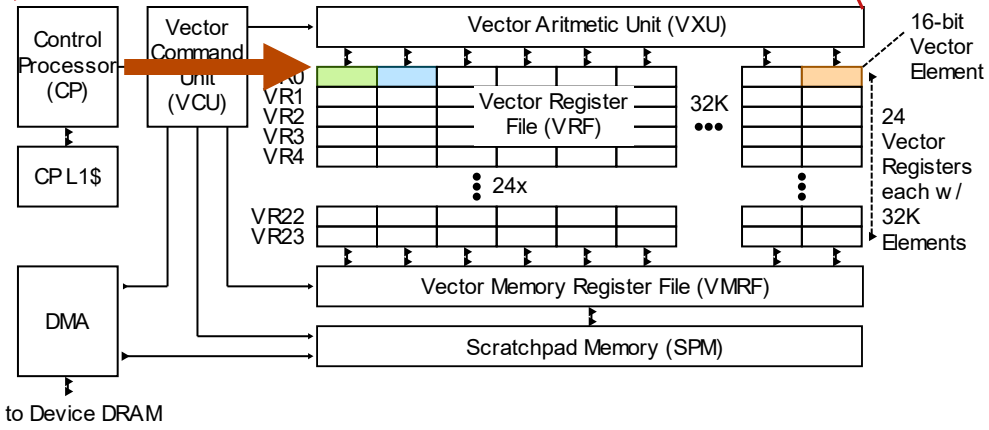
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle
DRAM → SPM	21,575	1.5
SPM → VMRF	470	69.7
VMRF → VRF	21	1568.9
Total	22,066	1.4

Microbenchmarking Data Movement and Fragment Overhead



System Overview



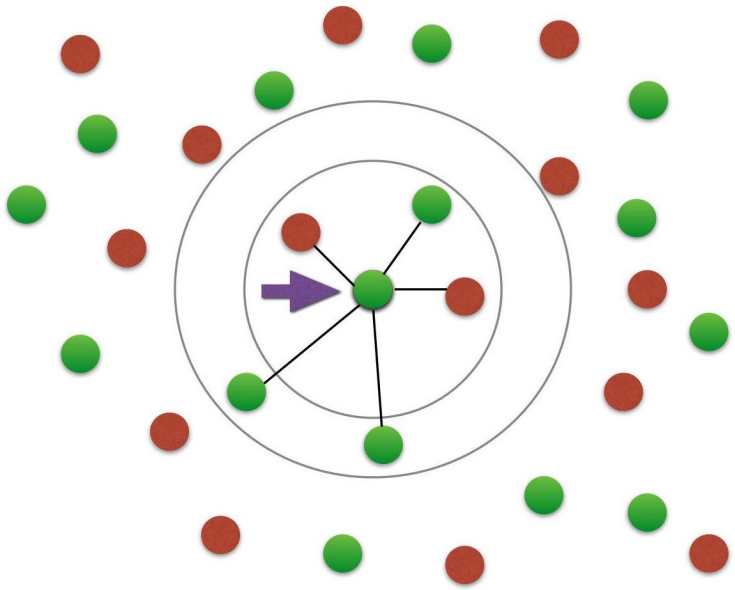
APU Core Logical View

Data Transfer	avg # cycles	elements/cycle
DRAM → SPM	21,575	1.5
SPM → VMRF	470	69.7
VMRF → VRF	21	1568.9
Total	22,066	1.4

Instruction	Description	Execution Time (cycles) Theoretical (b=16)	Measured
`vor_vv`	bit-wise or	6	15
`vmseq_vx`	set-if-equal	7	13
`vmseq_vx_m`	set-if-equal (masked)	9	15
`vmset`	set all elements of mask	3	10
`vmnot`	invert mask	7	12
`vmand`	bitwise-and of two masks	9	13
`vmv_vx`	broadcast scalar to vector	5	12
`vmv_vv`	copy vector	4	11
`vmv_m`	copy mask	4	11
`vsll`	bitwise left-shift logical	3	13
`vsrl`	bitwise right-shift logical	3	13

4-10 cycles

KNN Case Study



ANN_SIFT10 dataset, 32K points

89 searches/second

Fusing operations to skip unnecessary computation

97 searches/second

Fusing operations to re-use intermediates

112 searches/second

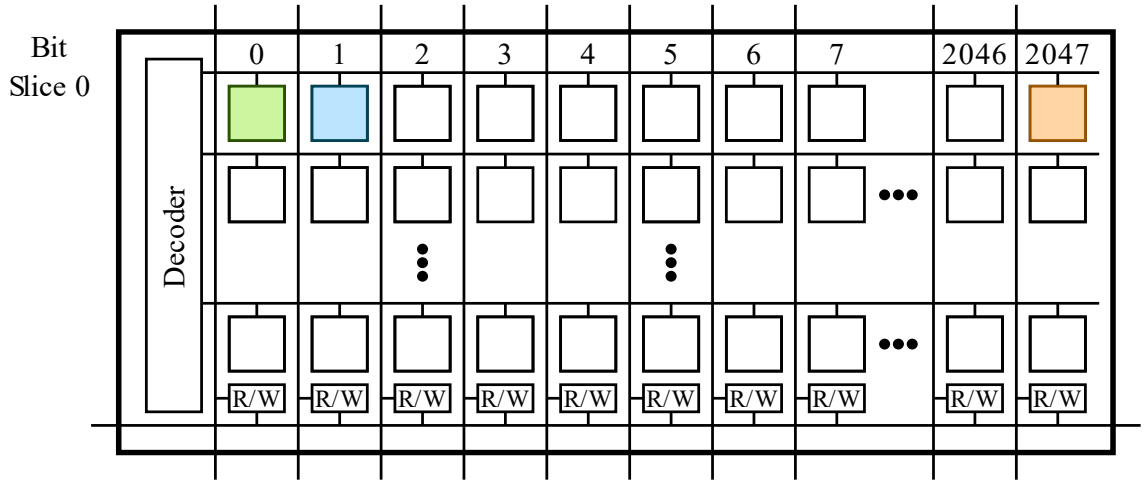
Asynchronous partial load from DRAM

129 searches/second

1.4x speedup, but 68% more LOC

Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

1. APU is an interesting platform for applications with massive parallelism, bitwise operations, narrow bitwidths, and high data reuse
2. Proof-of-concept support for a RISCV-like virtual vector instruction set
3. First step toward general-purpose computing on commercial domain-specific compute-in-SRAM accelerators




```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;
```