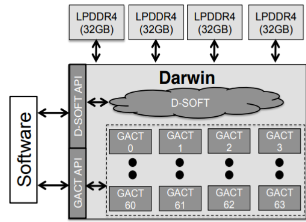
Accelerating Seed Location Filtering in DNA Read Mapping Using a Commercial Compute-in-SRAM Architecture

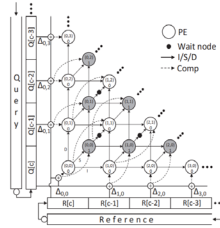
Courtney Golden[1], Dan Ilan[2], Nicholas Cebry[1], and Christopher Batten[1]

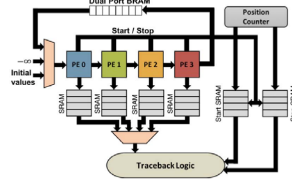
[1]Cornell University, [2]GSI Technologies Inc.

Genomics Acceleration

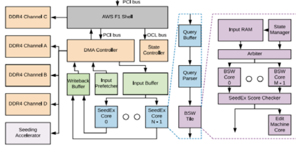


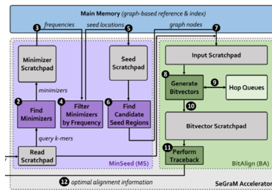
Darwin

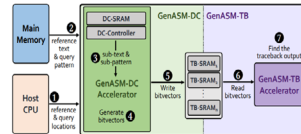

GenAx

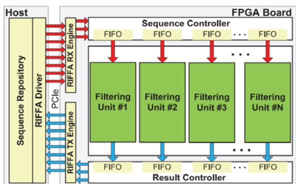


Darwin-WGA


SeedEx


SeGraM

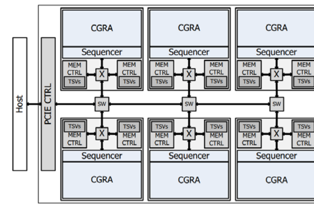

GenASM



Shouji

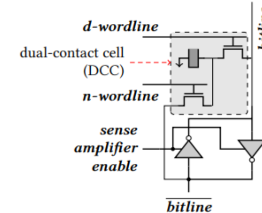
1: Declare register variables: M^u, M^d, V_M, V_I, V_D
(note: M^u refers to $M(i-1, j-1)$ and M^d refers to $M(i-1, j)$)
2: Initialize tables D and M according to the base cases.
3: For $j = 1 \to m$ do
4: $M^d \leftarrow 0, V_M \leftarrow 0, V_I \leftarrow -\infty$
5: For $j = 1 \to n$ do
6: $M^u \leftarrow M^d$
7: $M^d \leftarrow M(i-1, j)$
8: $V_I \leftarrow \max\{V_M + S_{GO}, V_I + S_{GE}\}$
9: $V_D \leftarrow \max\{M^d + S_{GO}, D(i-1, j) + S_{GE}\}$
10: $V_M \leftarrow \max\{M^u + \delta(R[i], T[j]), V_I, V_D\}$
11: $D(i,j) \leftarrow V_D$
12: $M(i,j) \leftarrow V_M$
13: End for
14: End for

SOAP3

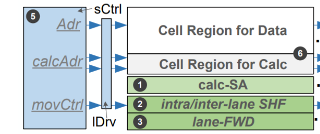


3D Manycore

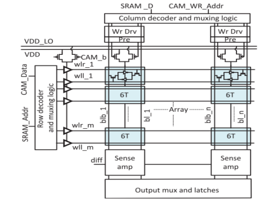
Compute-in-Memory

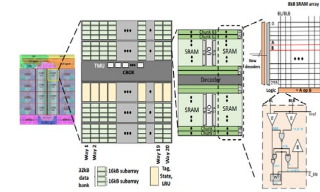


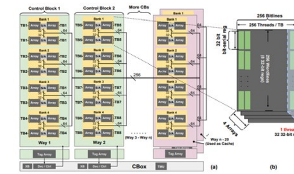
Ambit

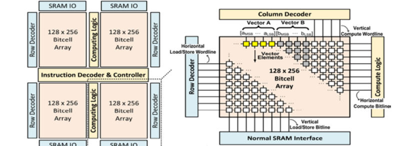


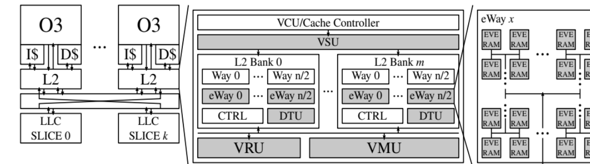
DRISA


28nm 6T SRAM

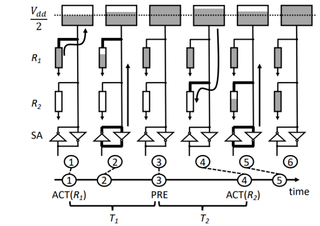

Neural Cache


Duality Cache


28nm Vector CRAM



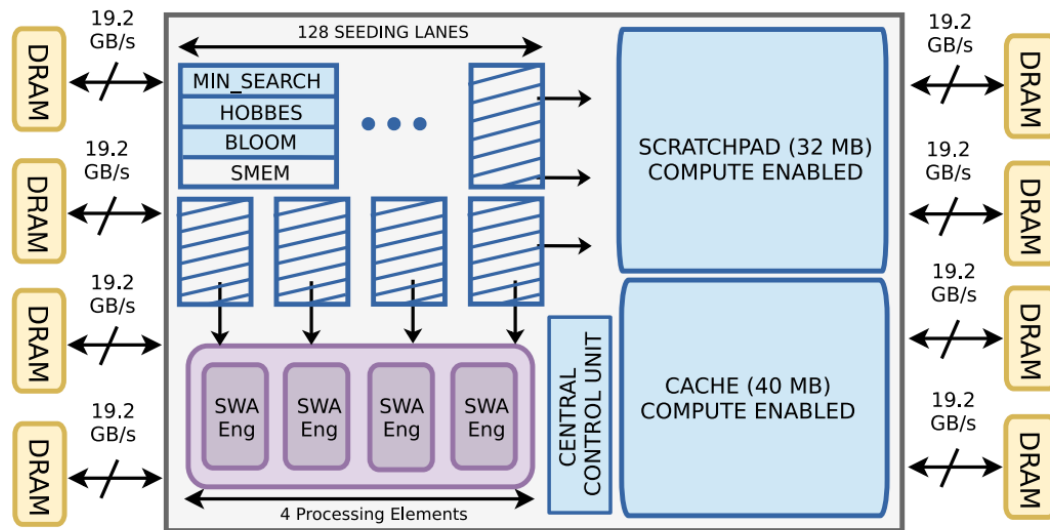
EVE


ComputeDRAM

USING COMPUTE-IN-MEMORY FOR GENOMICS

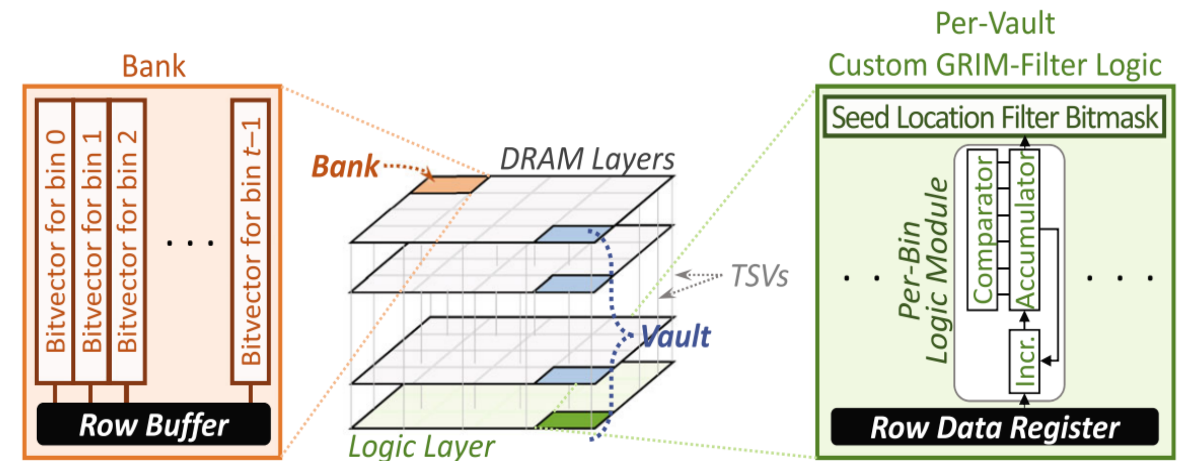
CSL

GenCache [MICRO '19]



- Proposes using tightly-coupled compute-in-SRAM accelerator with hardware extensions for filtering
- Simulation-based study showing 5.26x speedup over basic accelerator with no in-cache operations

GRIM-Filter [BMC Genomics '18]



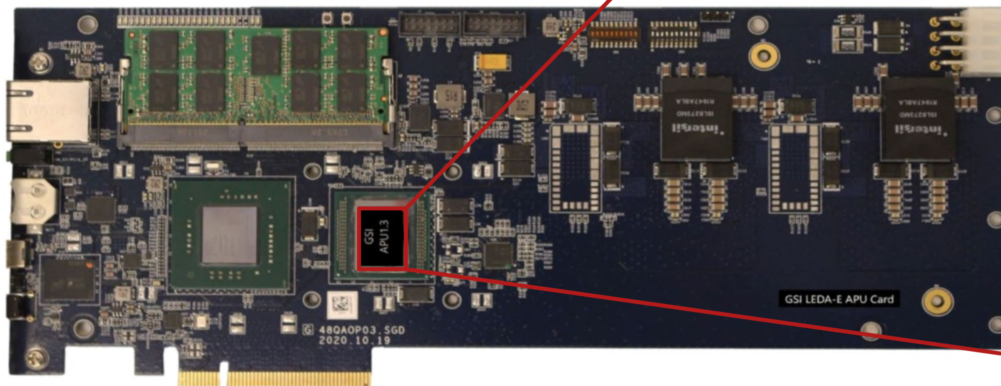
- Proposes using loosely-coupled compute-in-DRAM accelerator with custom hardware for filtering
- Simulation-based study showing 1.81-3.65x speedup over CPU

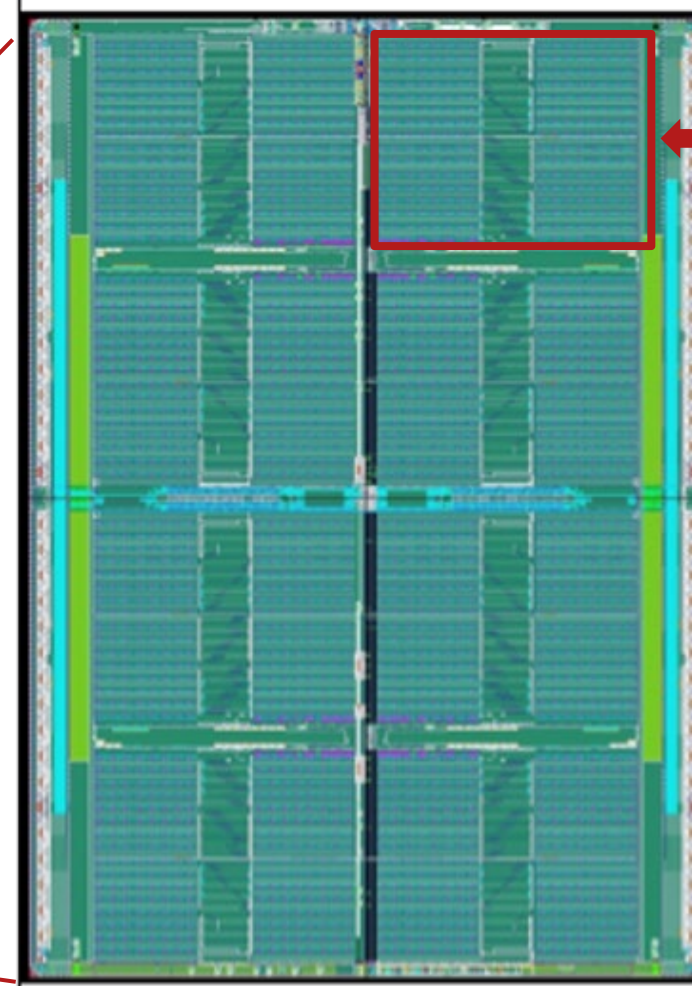
Our Goal: Explore the potential for accelerating seed location filtering in DNA read mapping using a general-purpose commercial-scale compute-in-SRAM architecture

GSI Gemini Platform: Associative Processing Unit (APU)

CSL

Device DRAM: 16 GB
Compute-Enabled SRAM: 262 KB
On-Chip Memory: 12 MB
On-Chip Memory Bandwidth: 26 TB/s
Compute Speed: 400 MHz
Power (TDP): 60 W
IC Process: TSMC 28nm



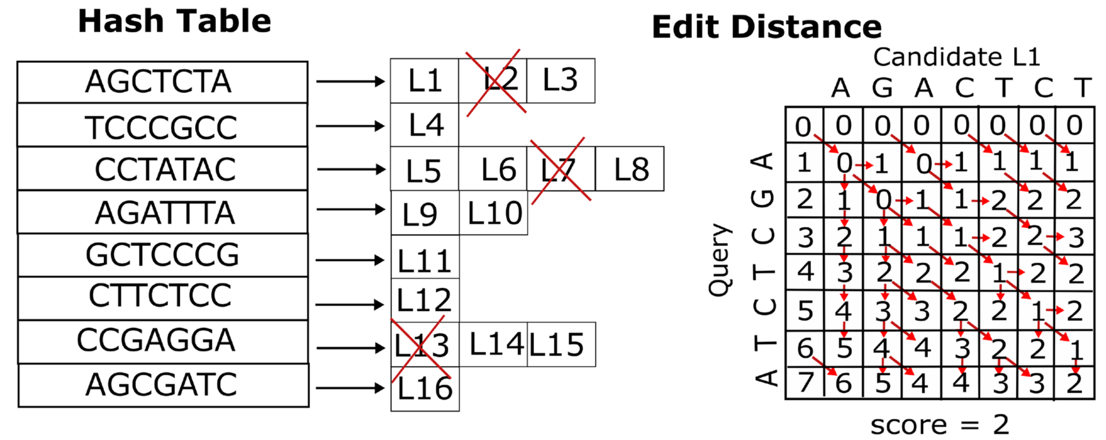


compute-enabled SRAM array

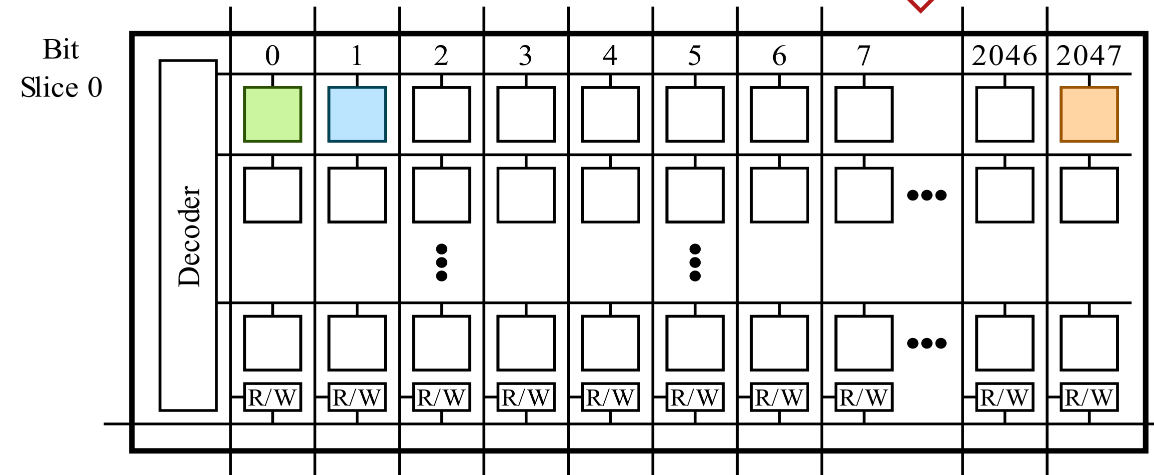
Leverages memory bitlines, peripheral logic, and highly-efficient search and update operations

Accelerating Seed Location Filtering in DNA Read Mapping Using a Commercial Compute-in-SRAM Architecture

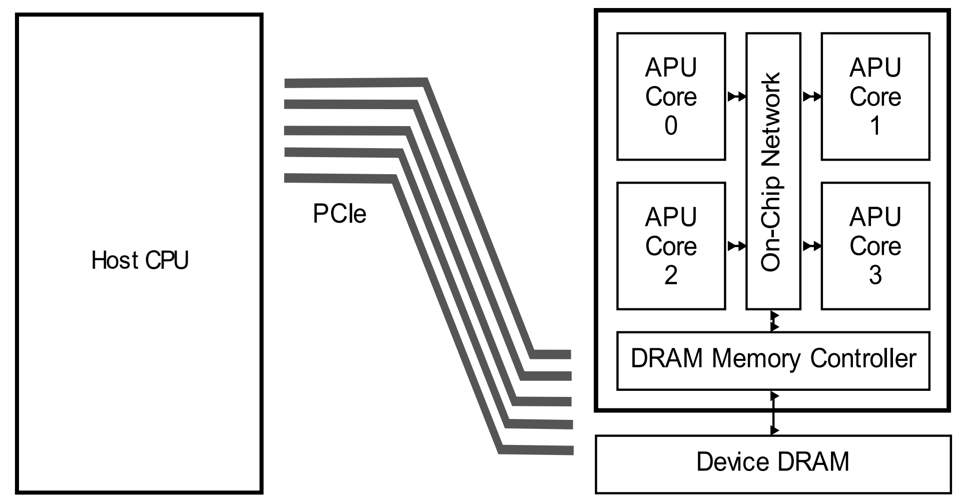
- Motivation
- **APU Microarchitecture**
- APU Microcoding
- APU for Myers' Acceleration




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

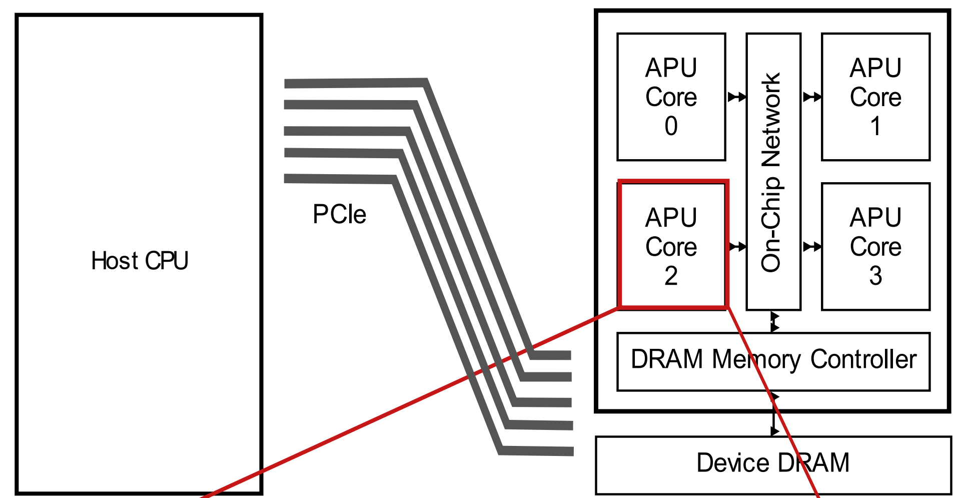


APU MICROARCHITECTURE

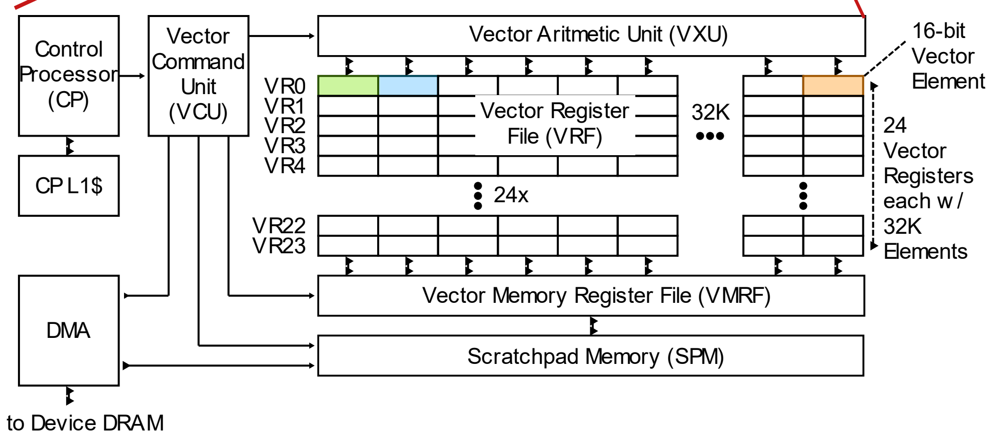


System Overview

APU MICROARCHITECTURE



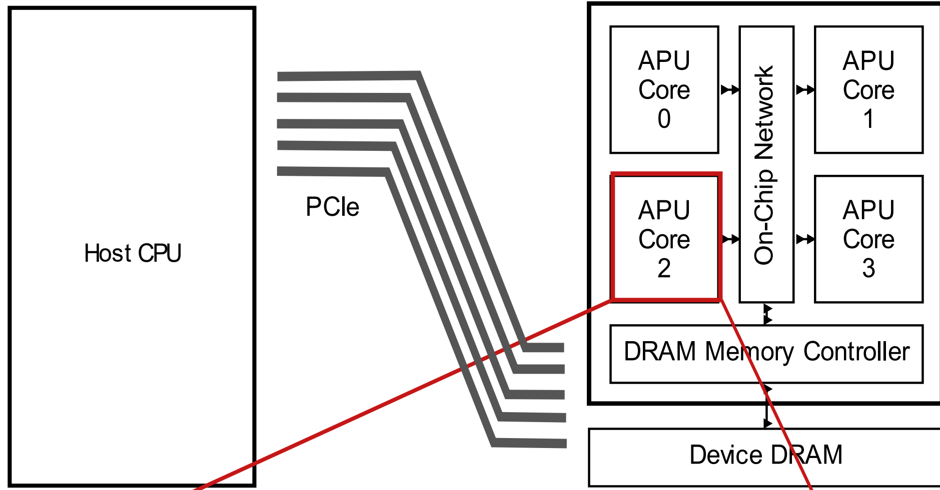
System Overview



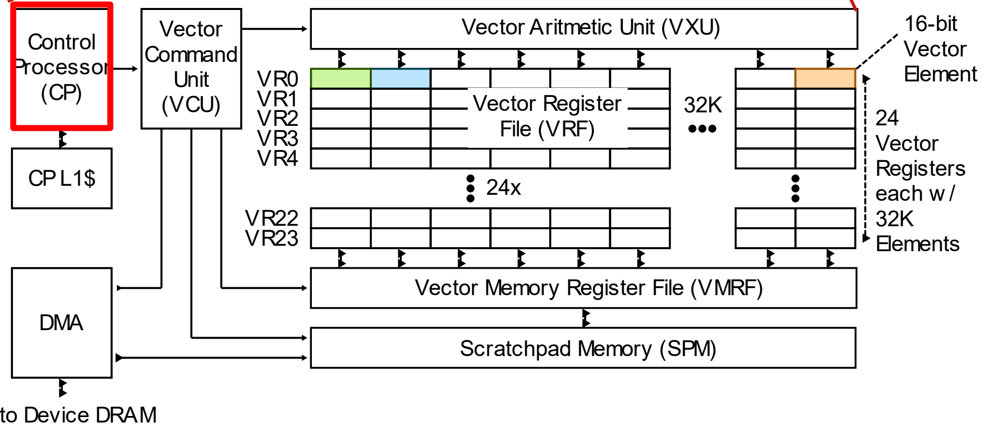
APU Core Logical View

APU MICROARCHITECTURE

CSL



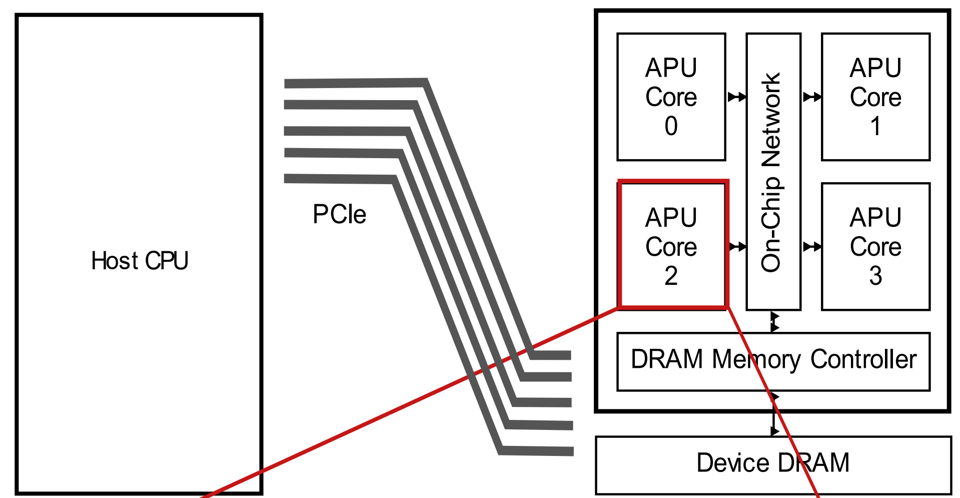
System Overview



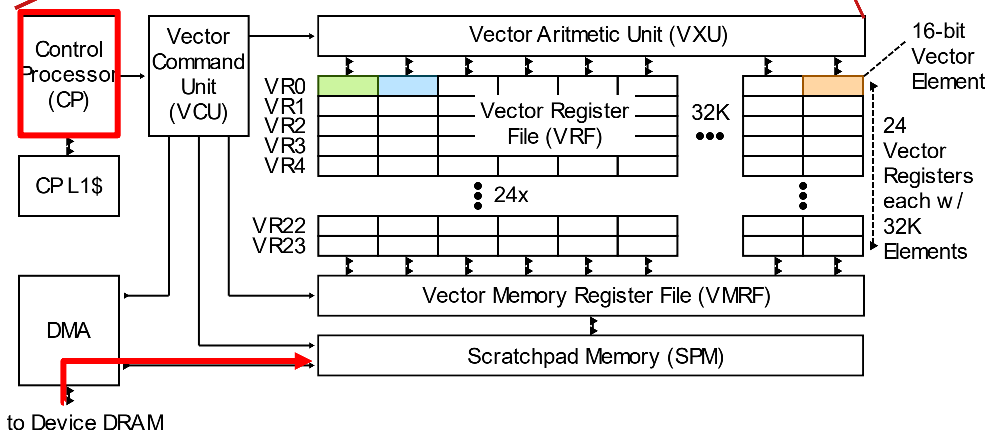
APU Core Logical View

APU MICROARCHITECTURE

CSL

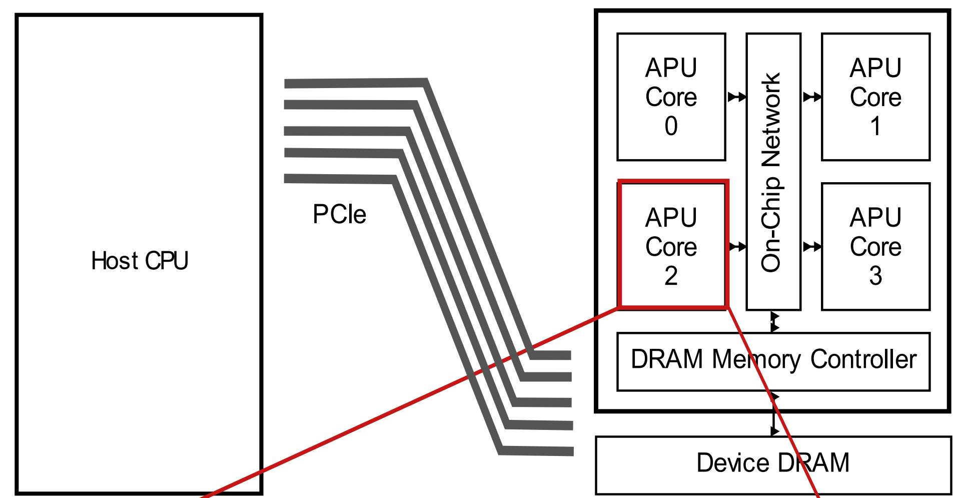


System Overview

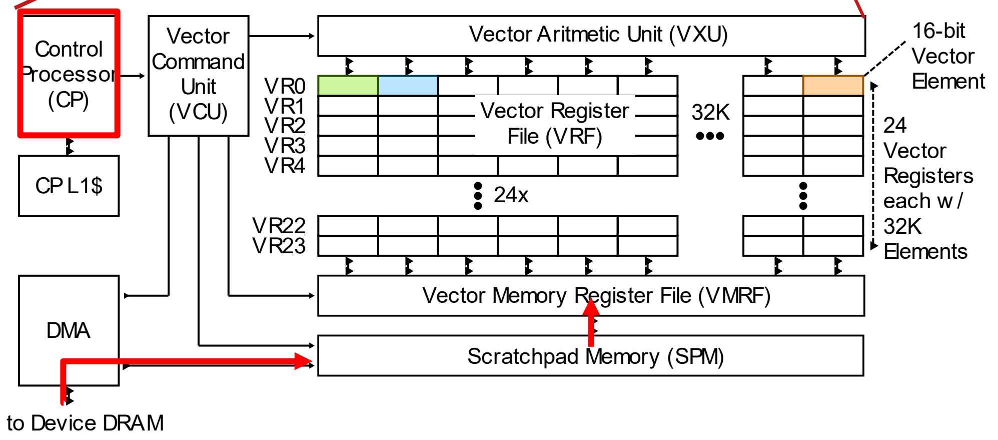


APU Core Logical View

APU Microarchitecture



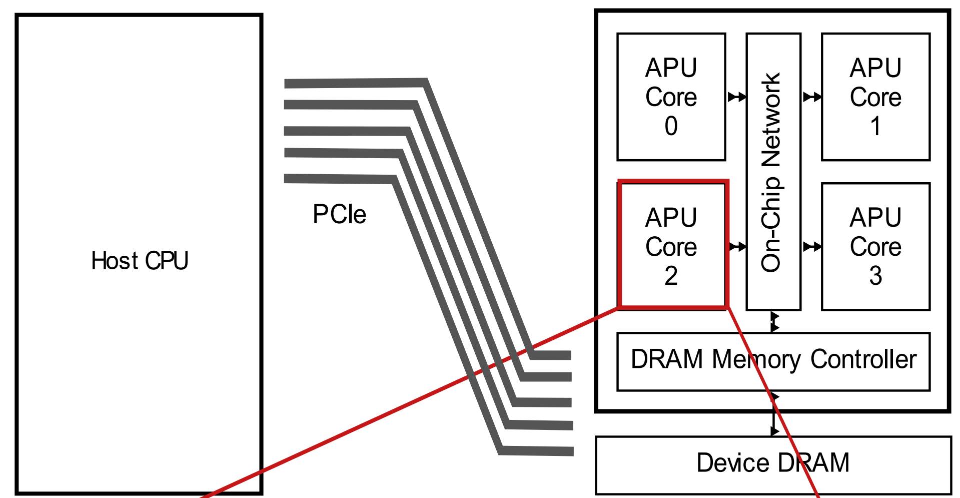
System Overview



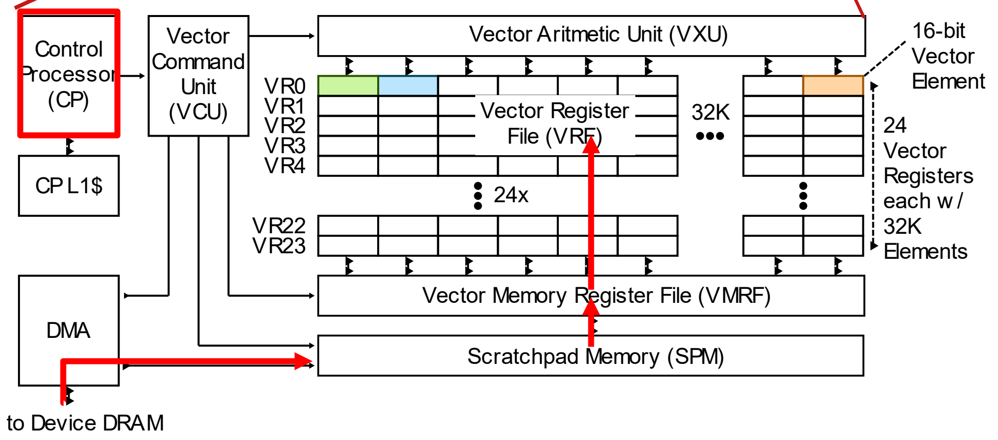
APU Core Logical View

APU MICROARCHITECTURE

CSL

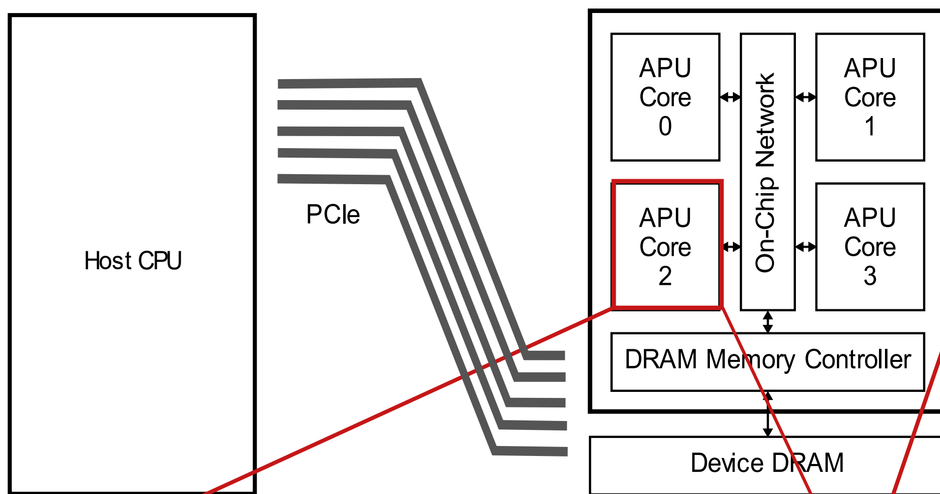


System Overview

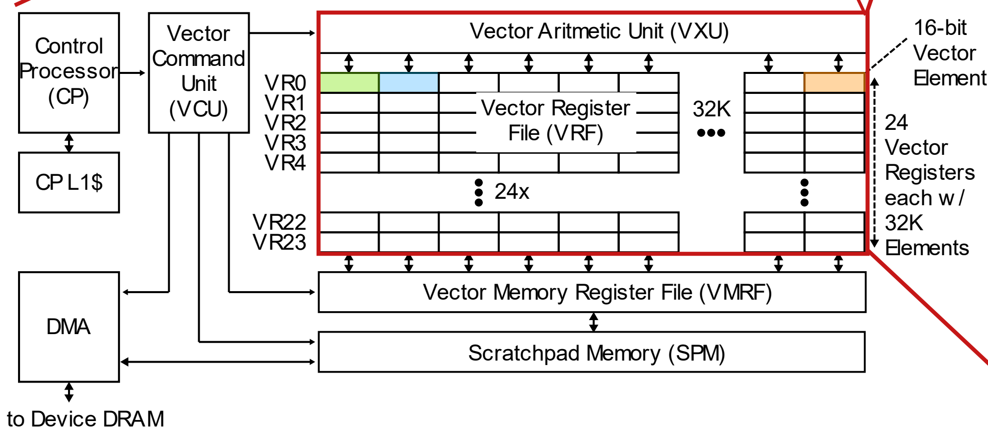


APU Core Logical View

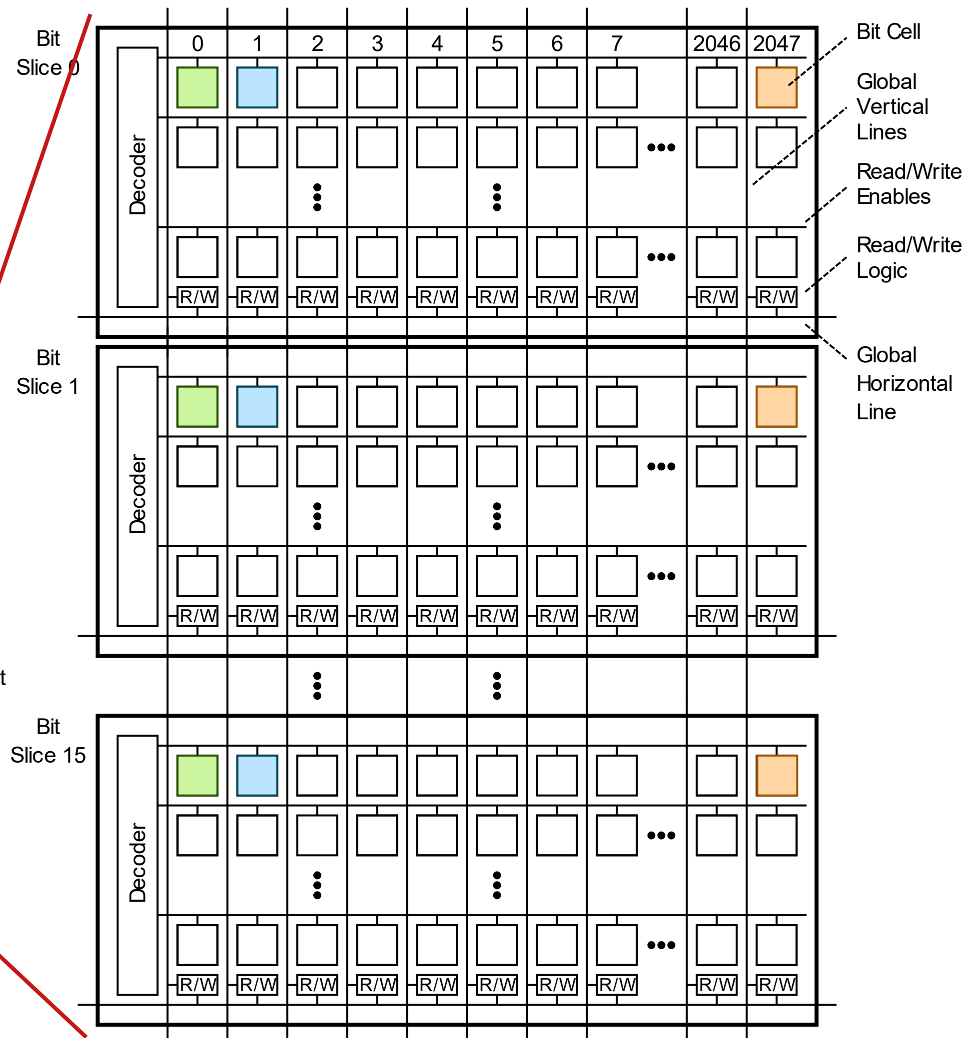
APU Microarchitecture



System Overview



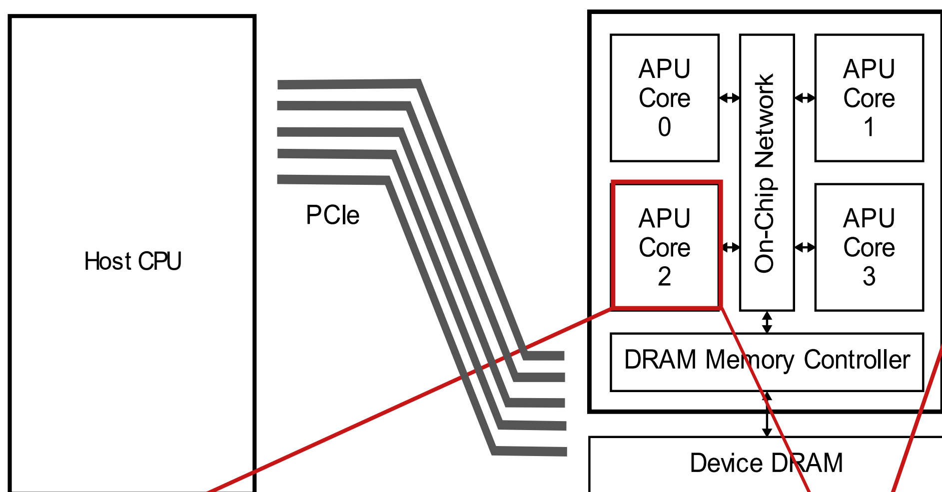
APU Core Logical View



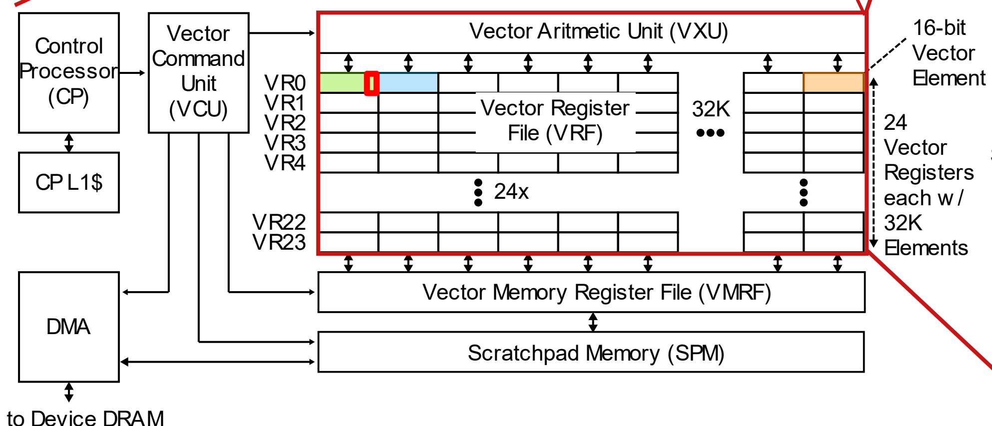
SRAM Bank Physical View

APU MICROARCHITECTURE

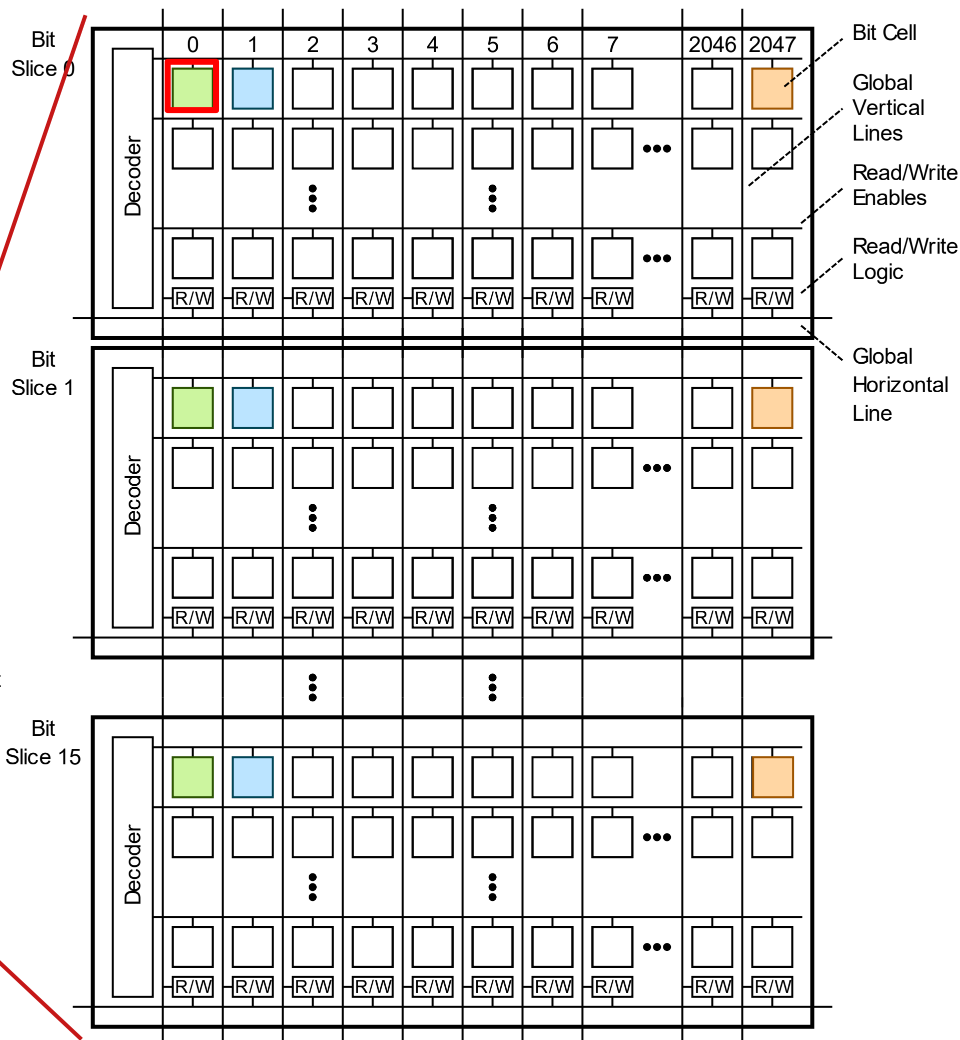
CSL



System Overview



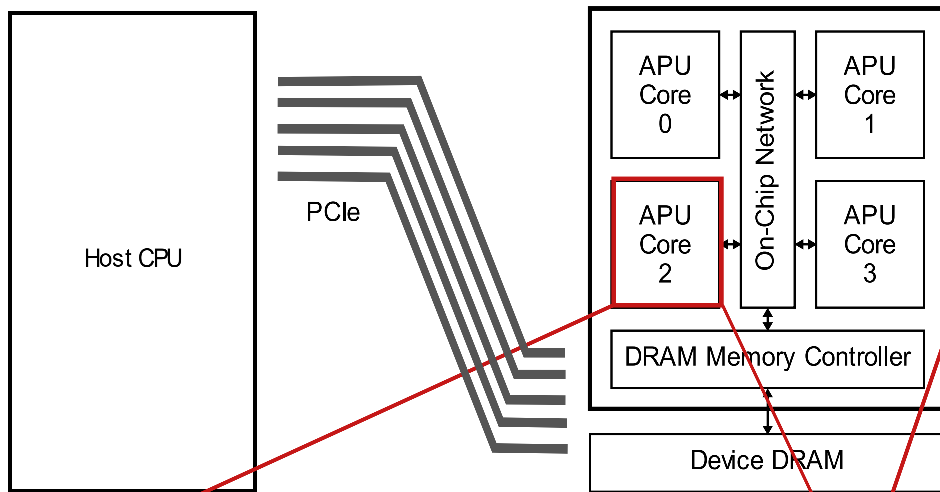
APU Core Logical View



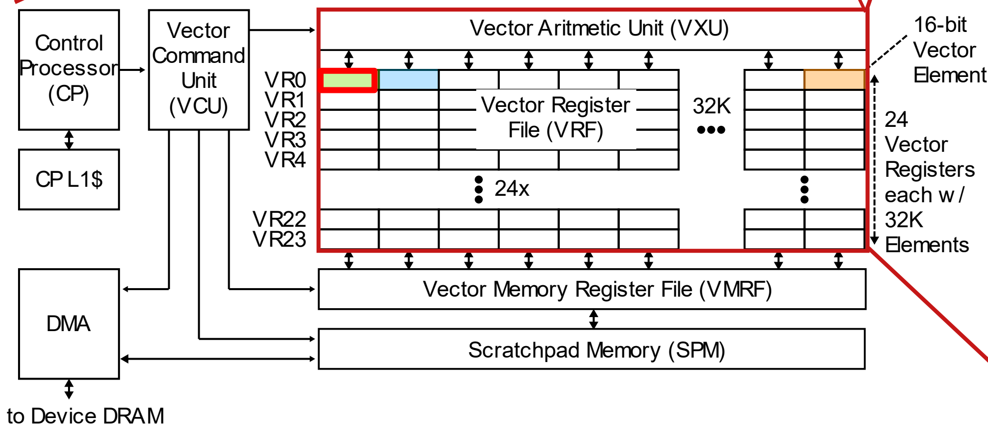
SRAM Bank Physical View

APU MICROARCHITECTURE

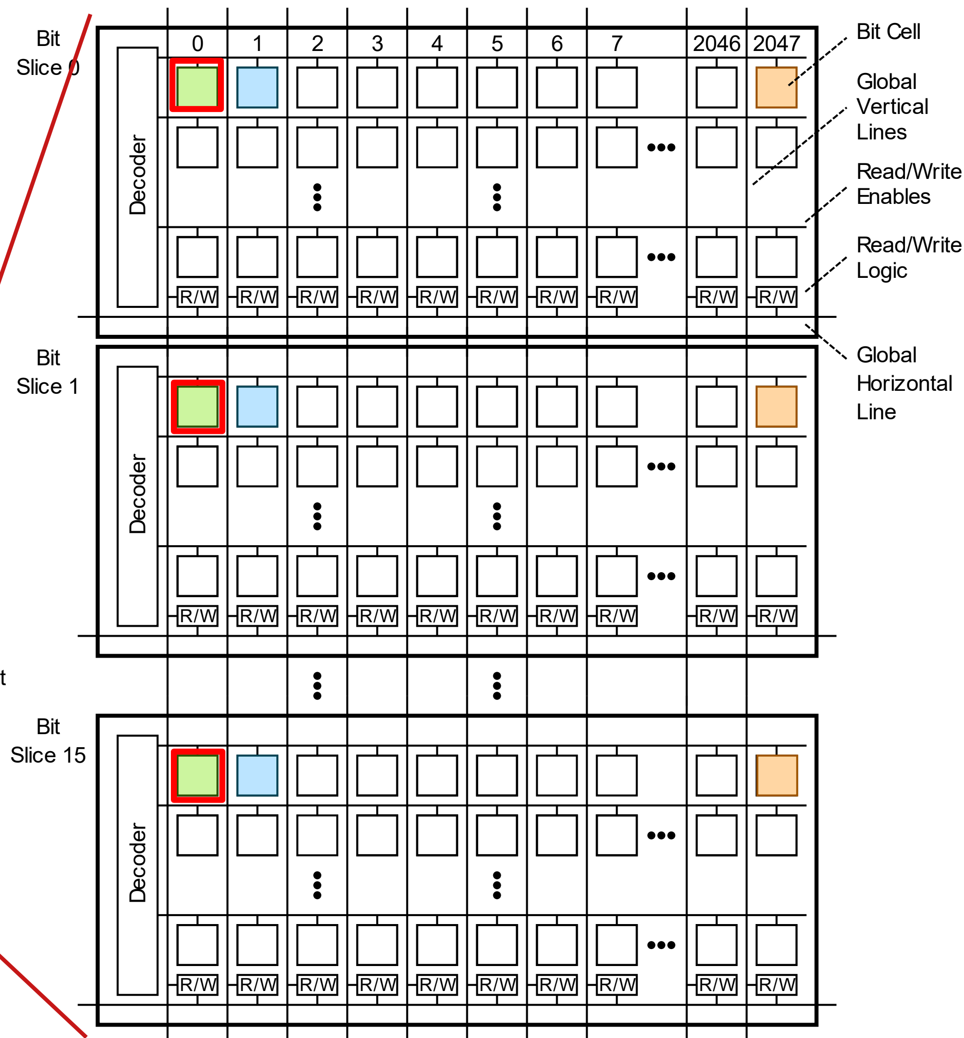
CSL



System Overview



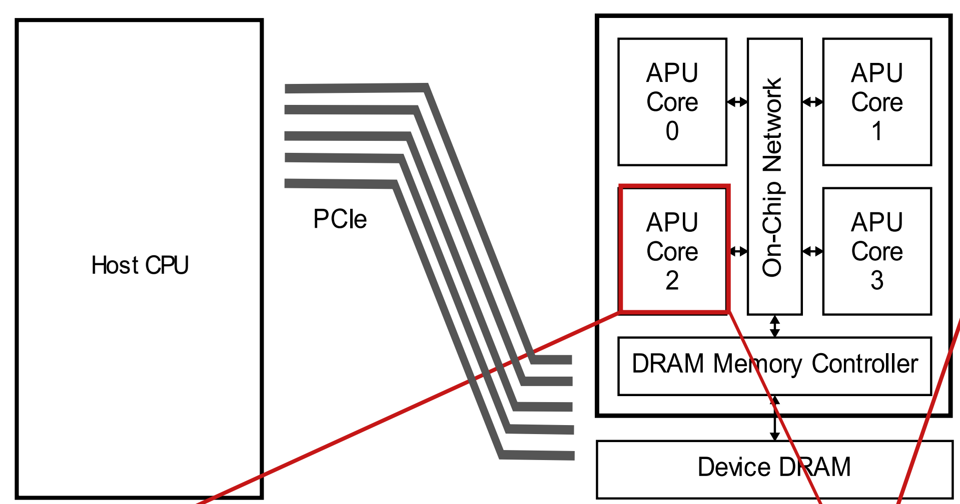
APU Core Logical View



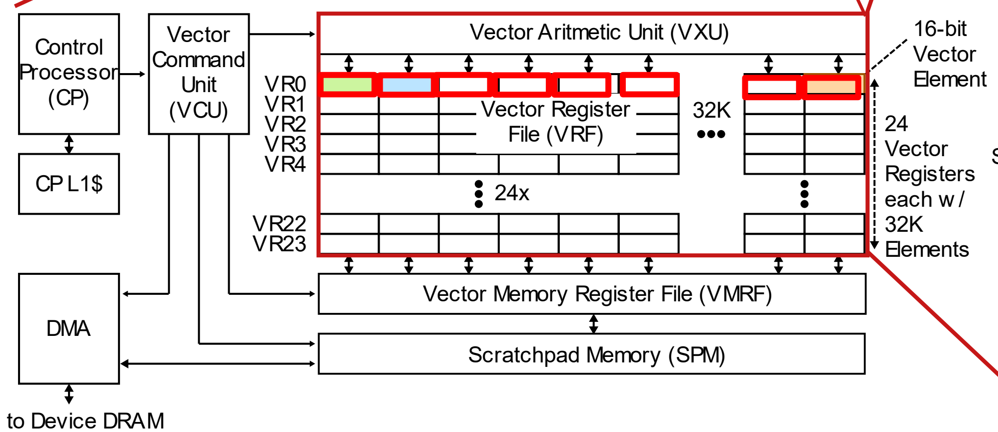
SRAM Bank Physical View

APU Microarchitecture

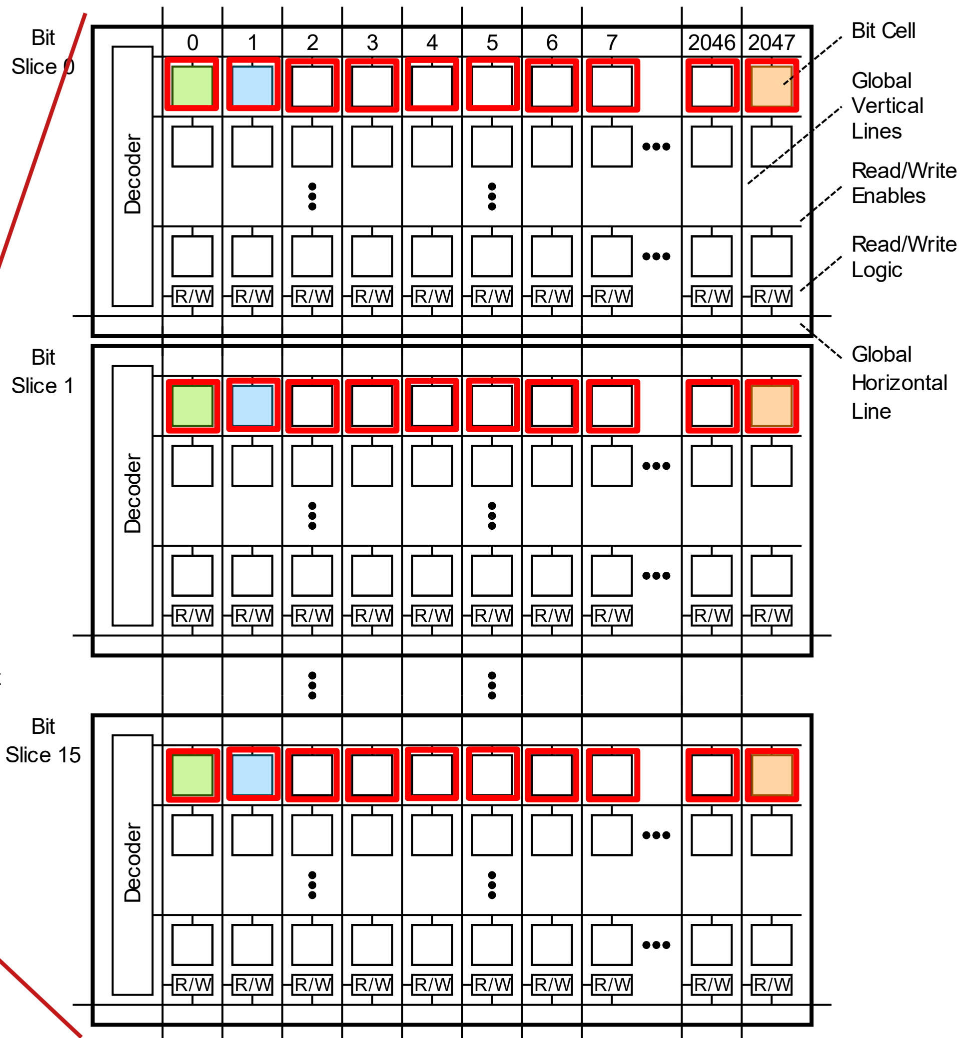
CSL



System Overview



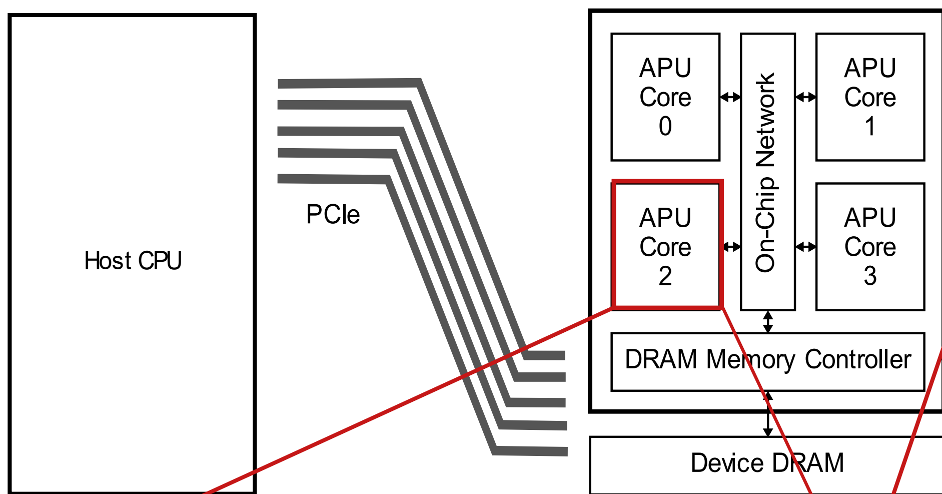
APU Core Logical View



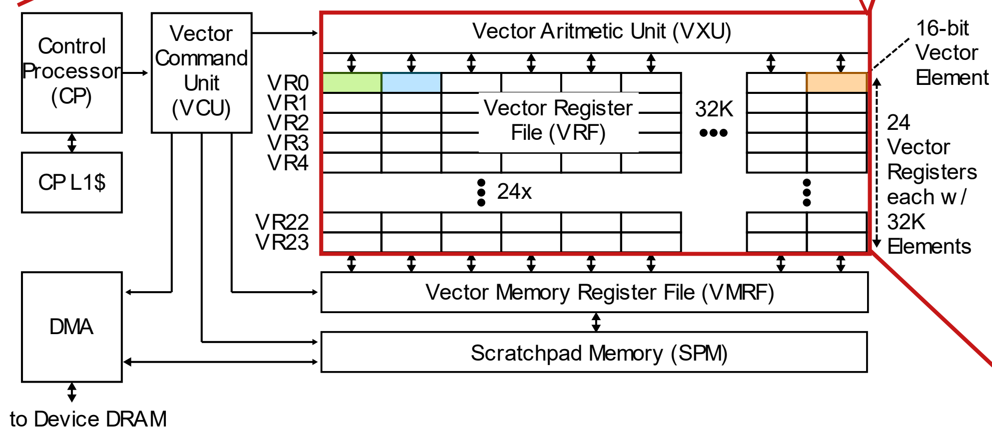
SRAM Bank Physical View

APU MICROARCHITECTURE

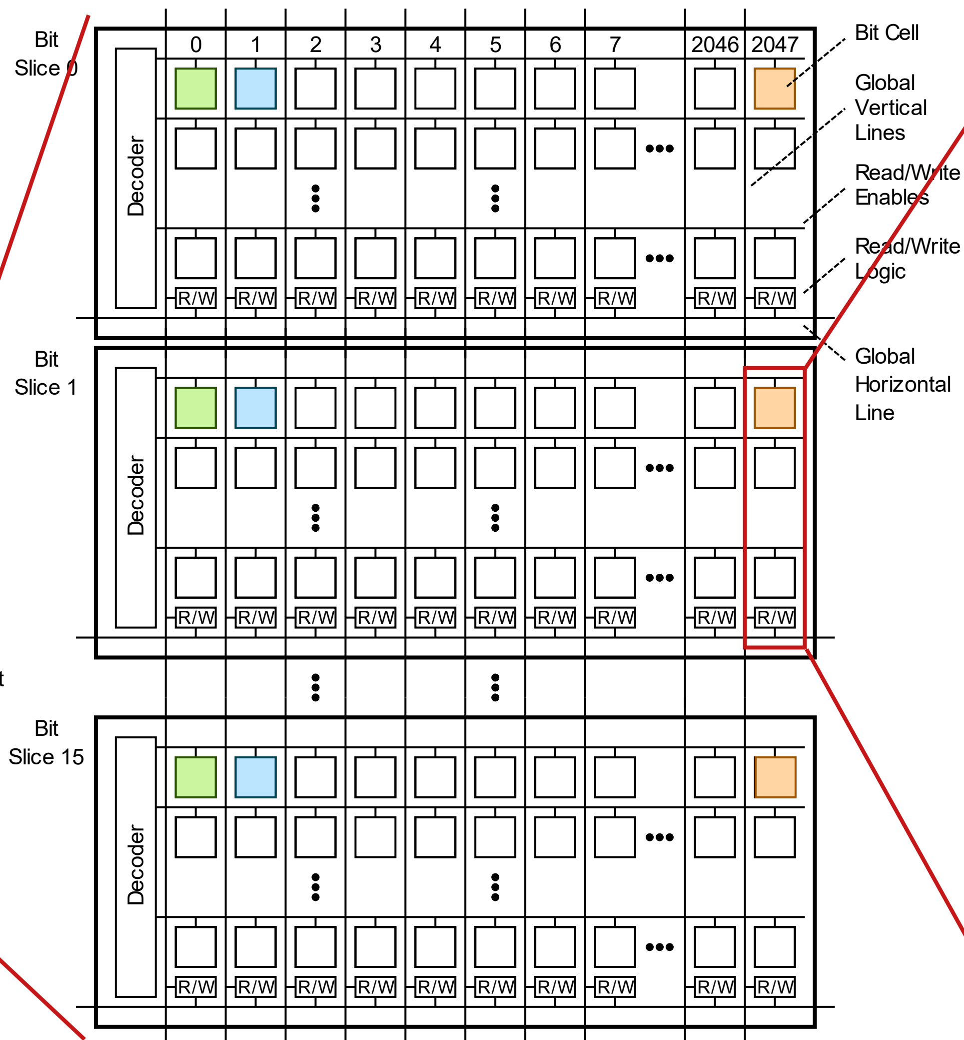
CSL



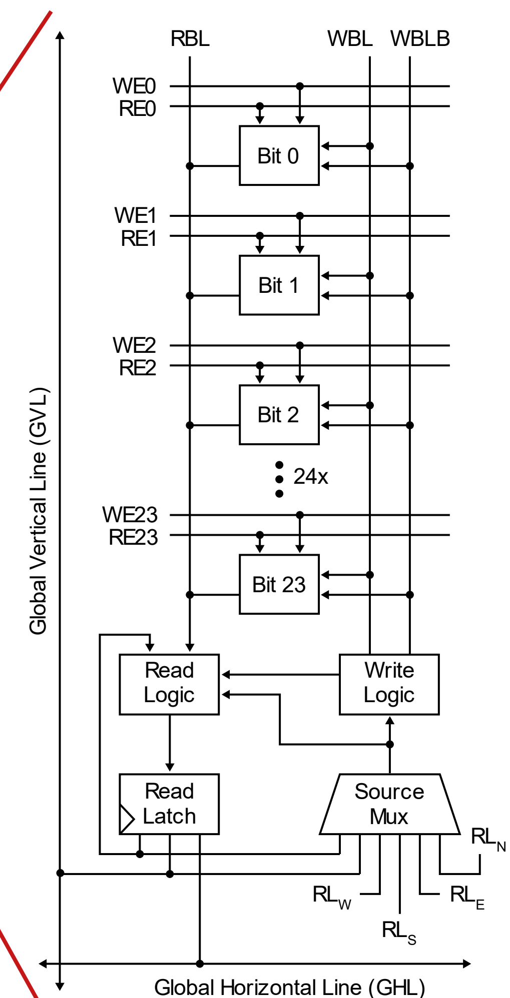
System Overview



APU Core Logical View

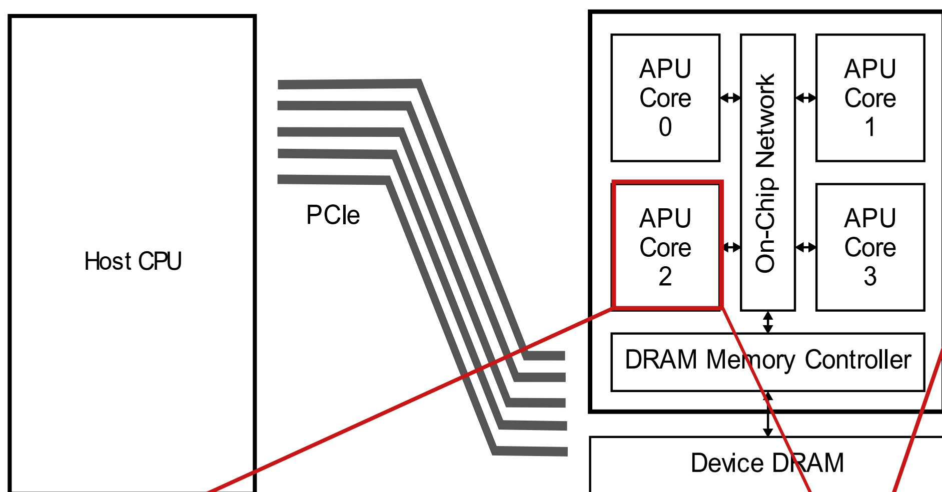


SRAM Bank Physical View

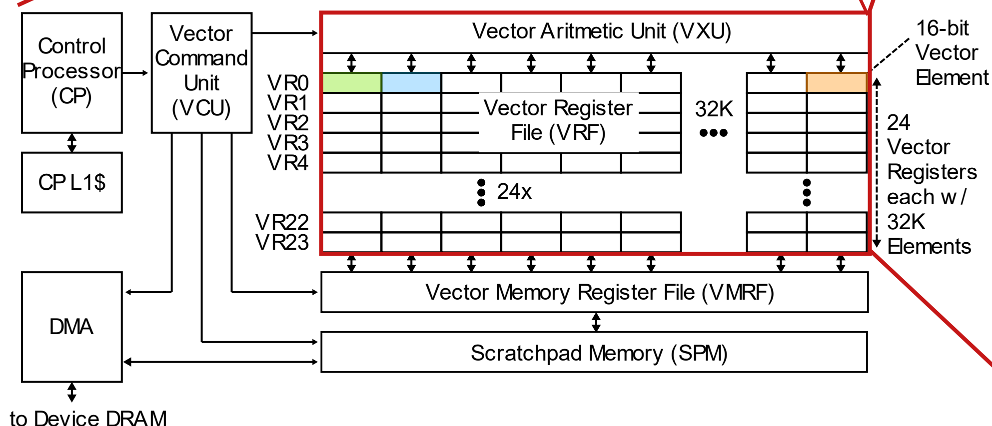


Bit Processor Circuitry

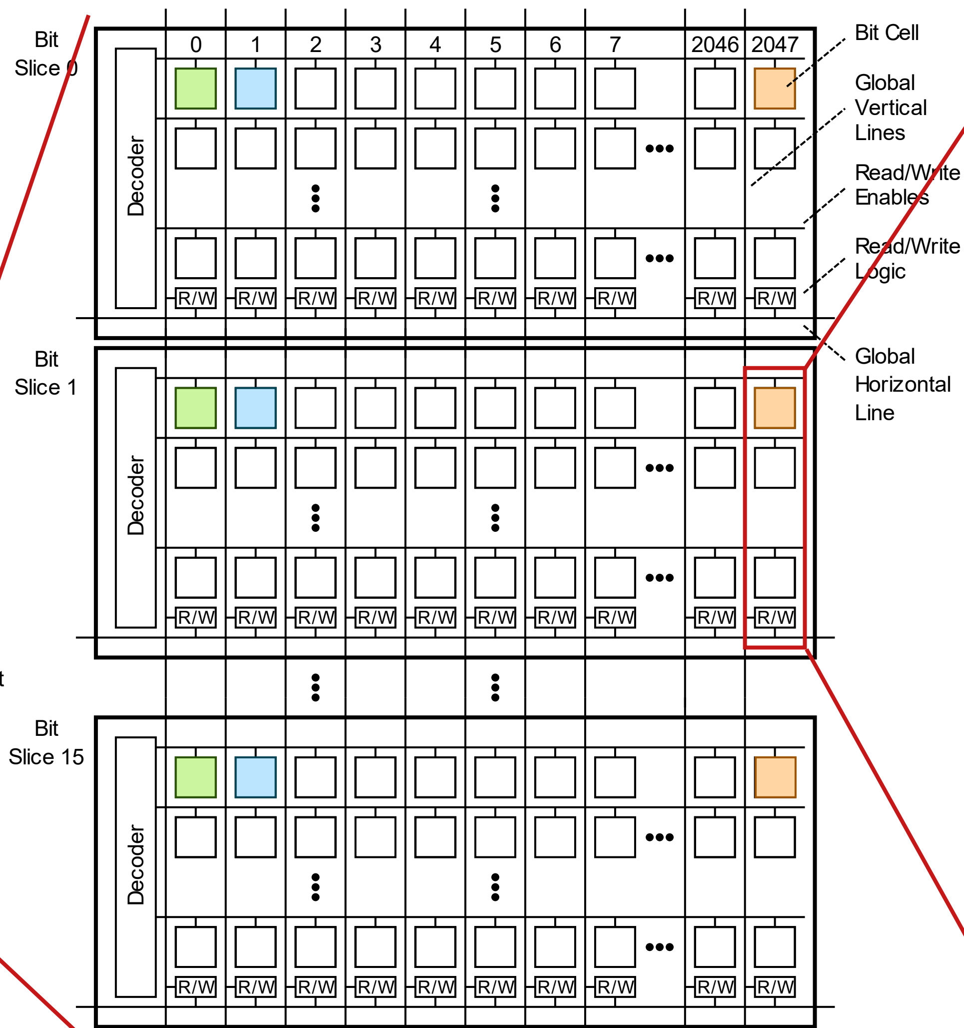
APU Microarchitecture



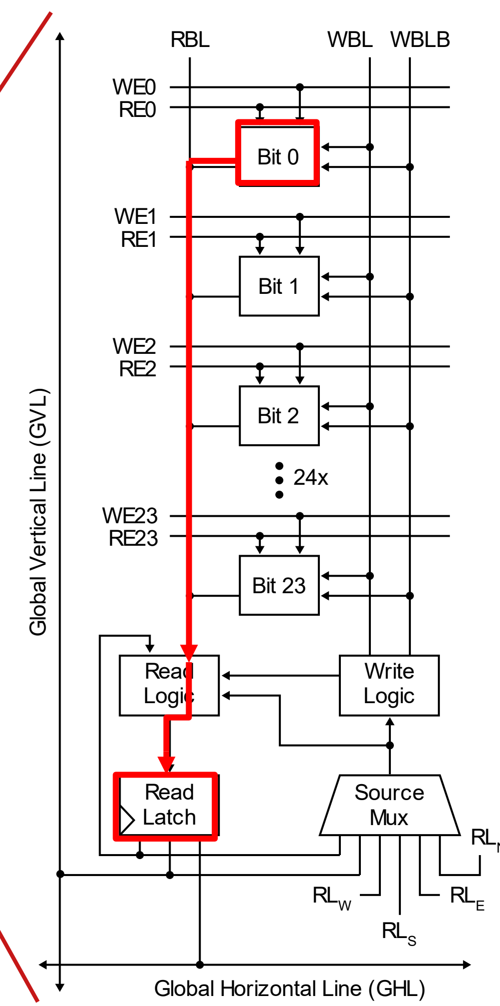
System Overview



APU Core Logical View



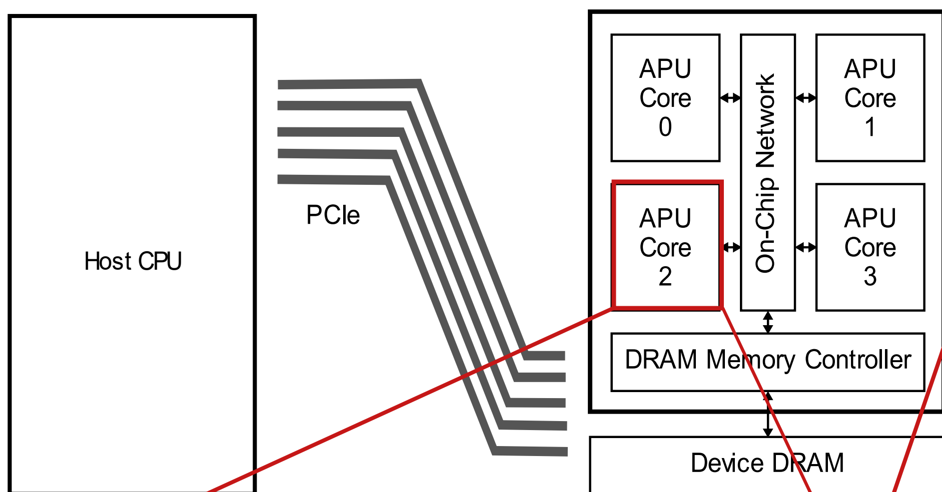
SRAM Bank Physical View



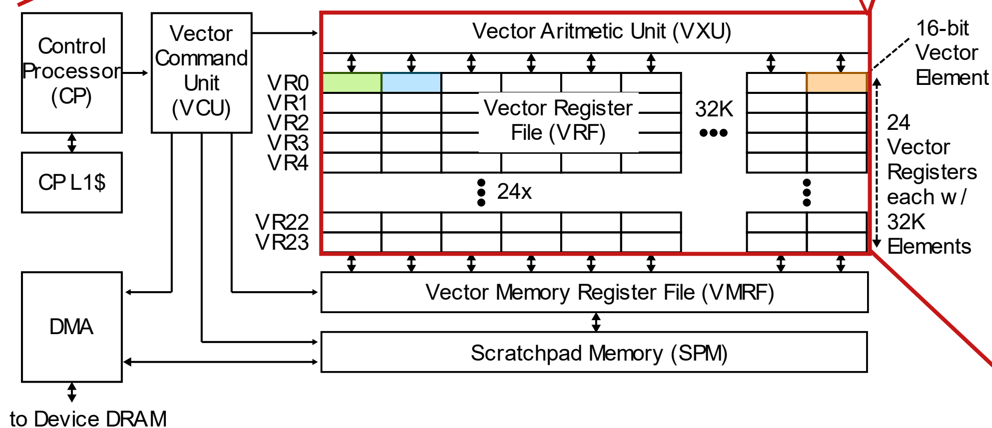
Bit Processor Circuitry

APU Microarchitecture

CSL



System Overview



APU Core Logical View

Bit Slice 0

Bit Slice 1

Bit Slice 15

Decoder

0 1 2 3 4 5 6 7 2046 2047

R/W

Bit Cell

Global Vertical Lines

Read/Write Enables

Read/Write Logic

Global Horizontal Line

SRAM Bank Physical View

RBL WBL WBLB

WE0 RE0

Bit 0

WE1 RE1

Bit 1

WE2 RE2

Bit 2

24x

WE23 RE23

Bit 23

Global Vertical Line (GVL)

Read Logic

Write Logic

Read Latch

Source Mux

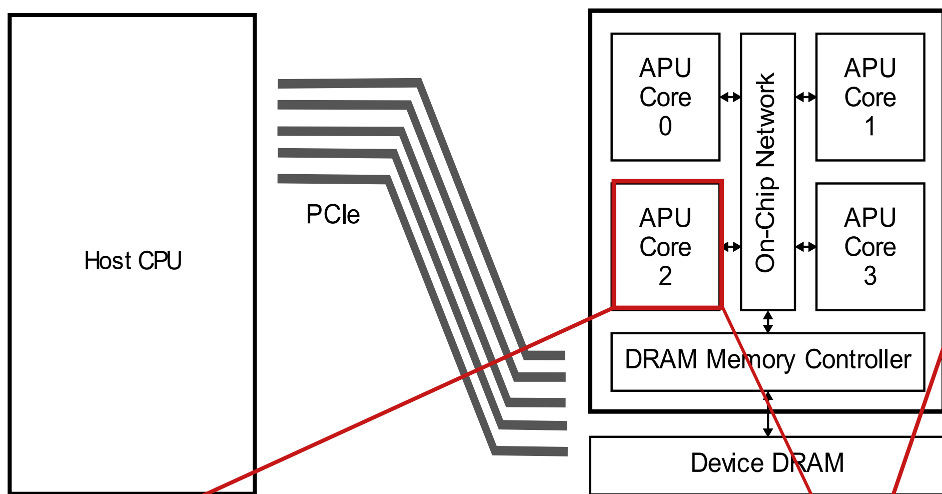
RL_N RL_W RL_E RL_S

Global Horizontal Line (GHL)

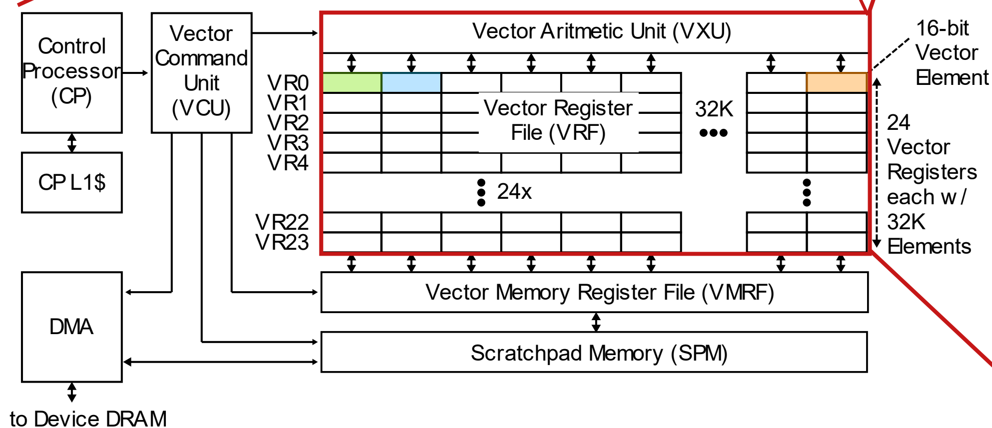
Bit Processor Circuitry

APU MICROARCHITECTURE

CSL

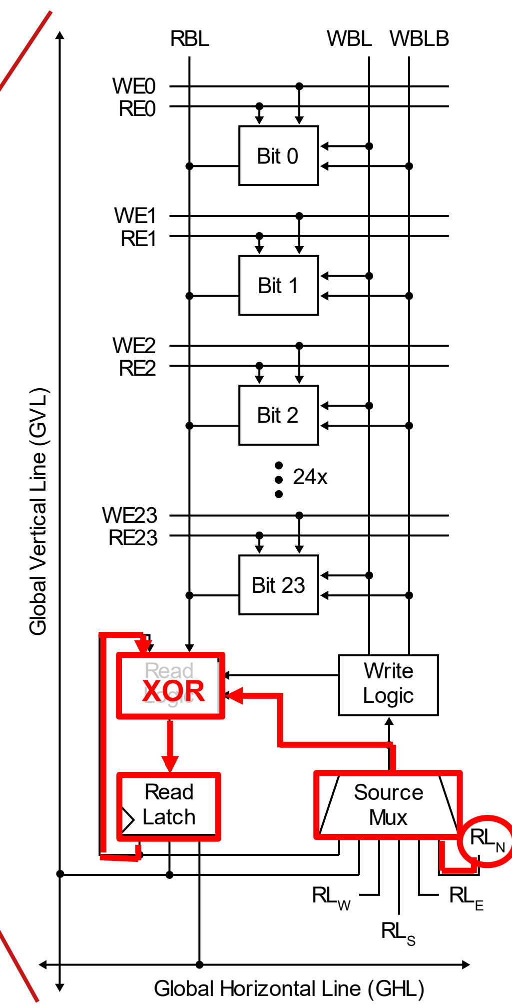


System Overview



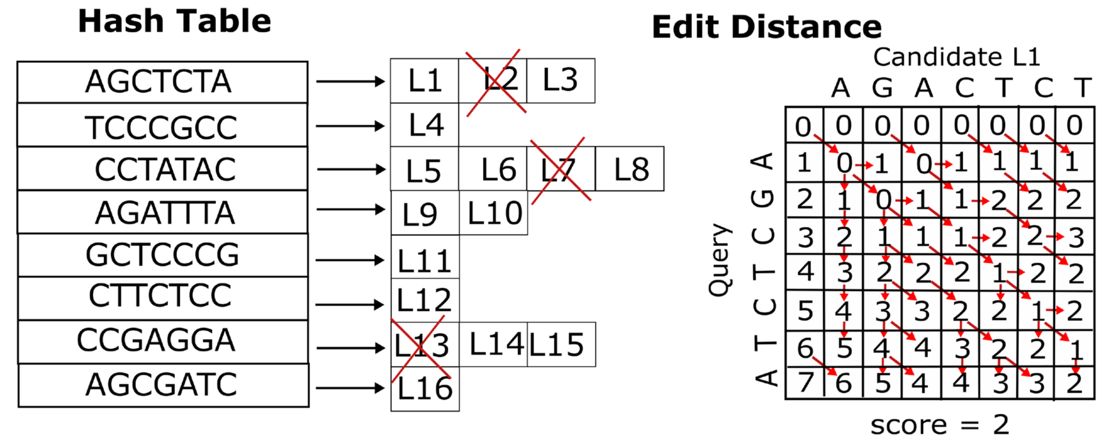
APU Core Logical View

SRAM Bank Physical View



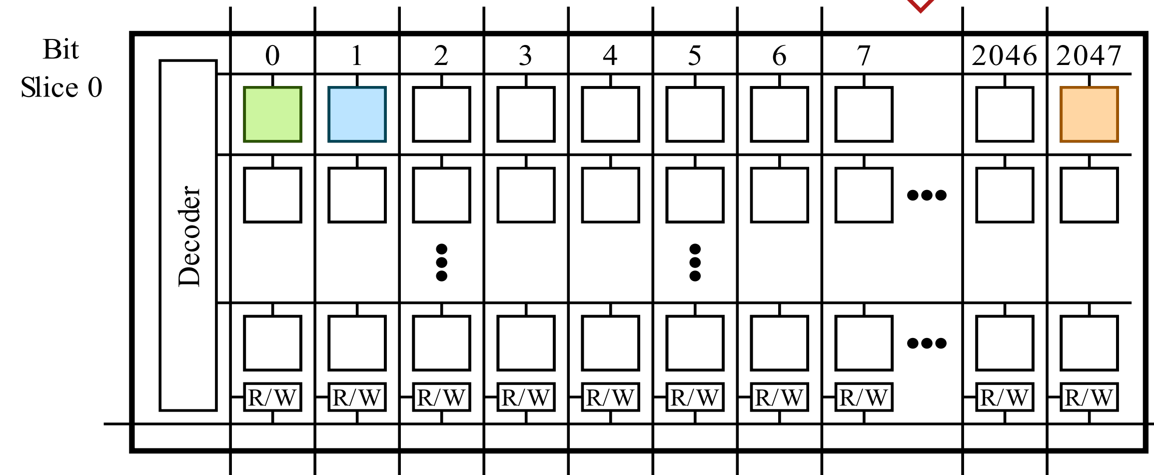
Bit Processor Circuitry

Accelerating Seed Location Filtering in DNA Read Mapping Using a Commercial Compute-in-SRAM Architecture




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

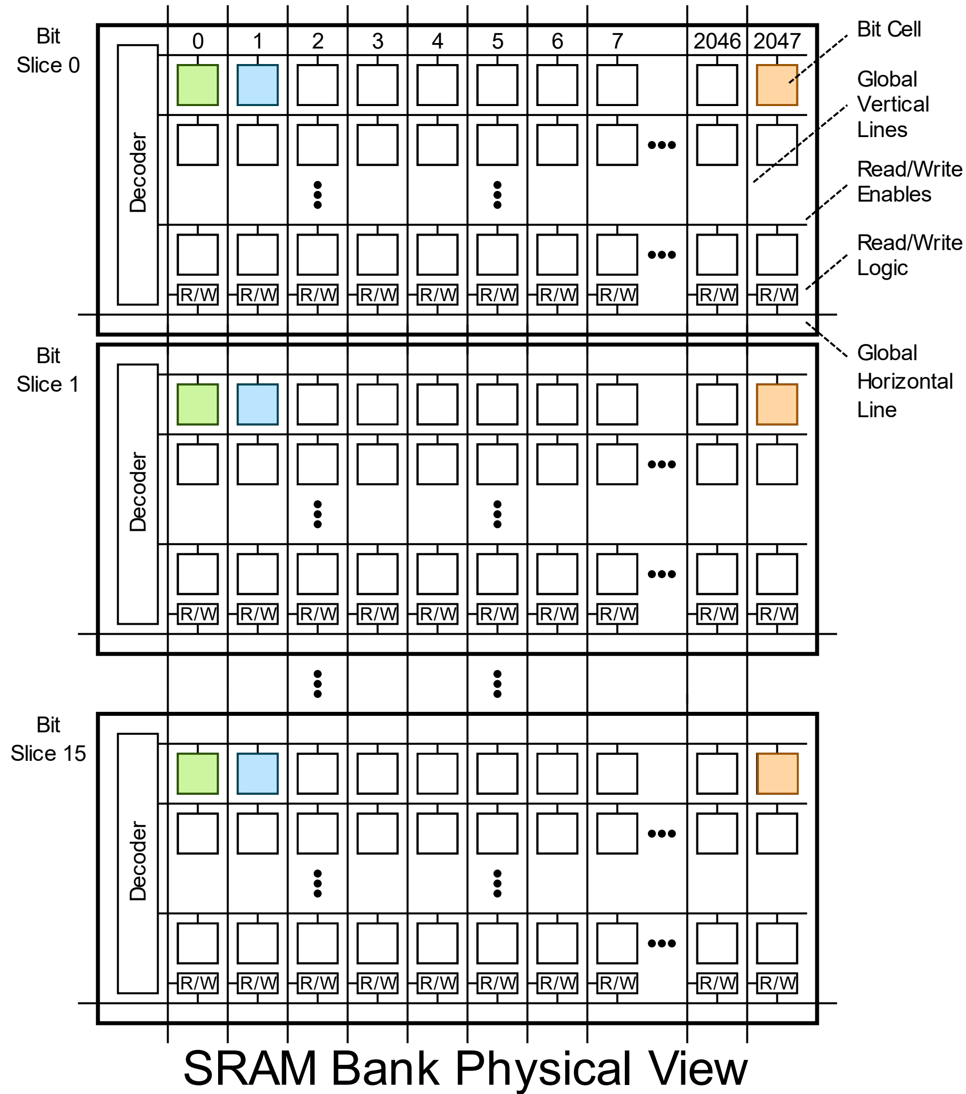
- Motivation
- APU Microarchitecture
- **APU Microcoding**
- APU for Myers' Acceleration



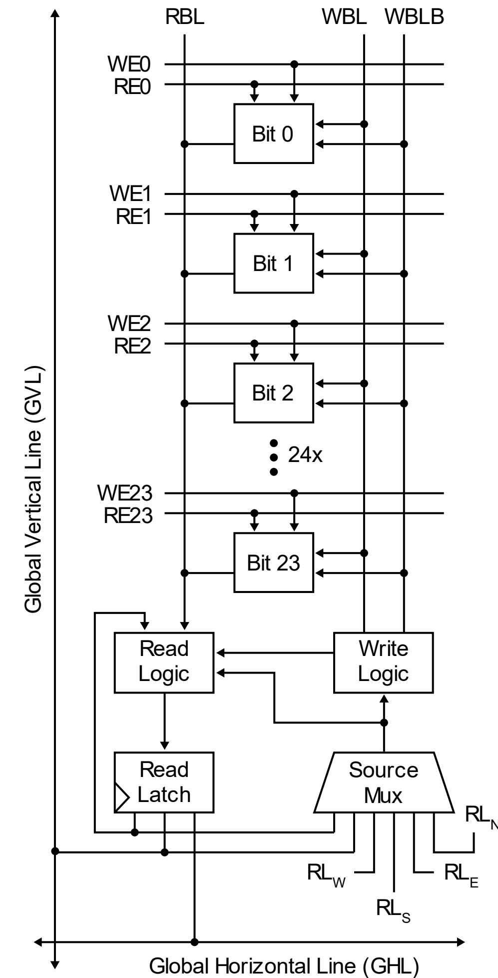
EXAMPLE #1: BITWISE OR

```
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;
}
```



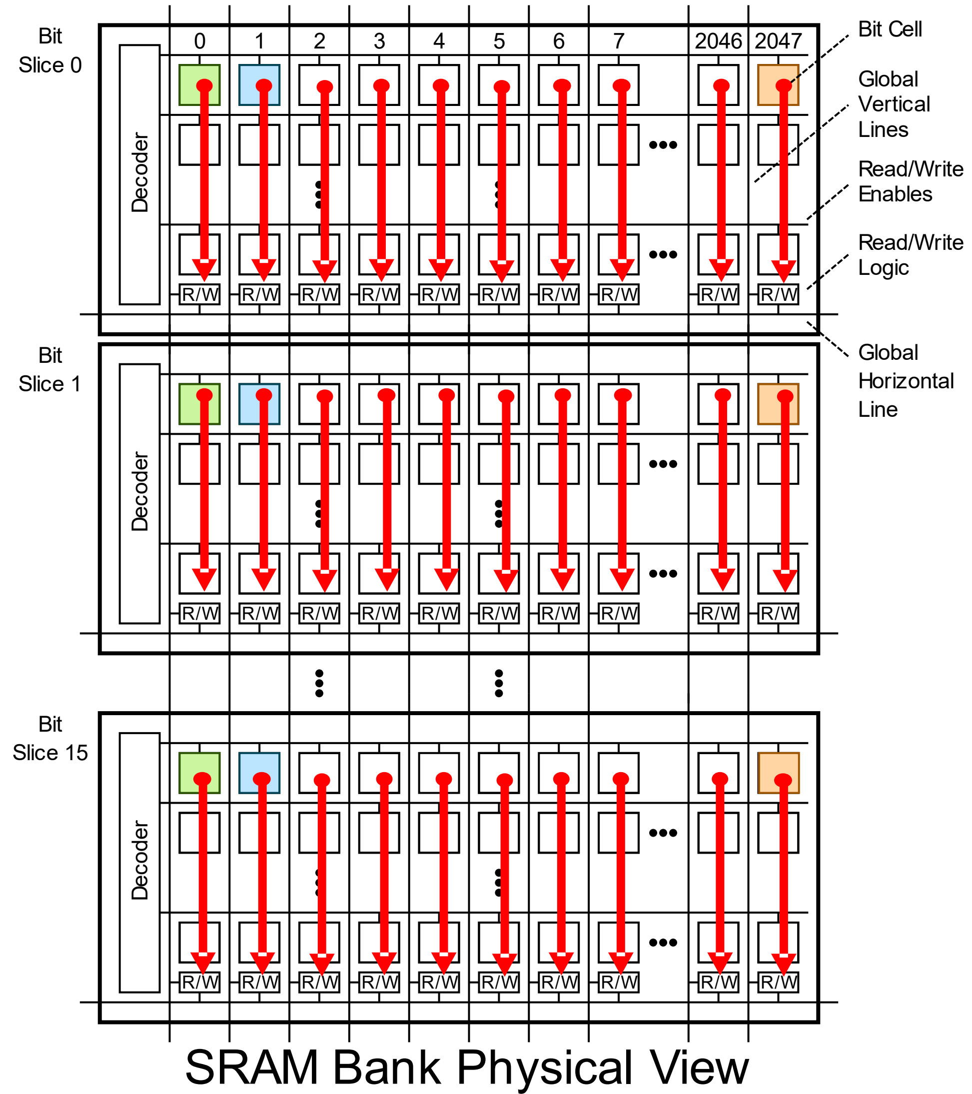
SRAM Bank Physical View



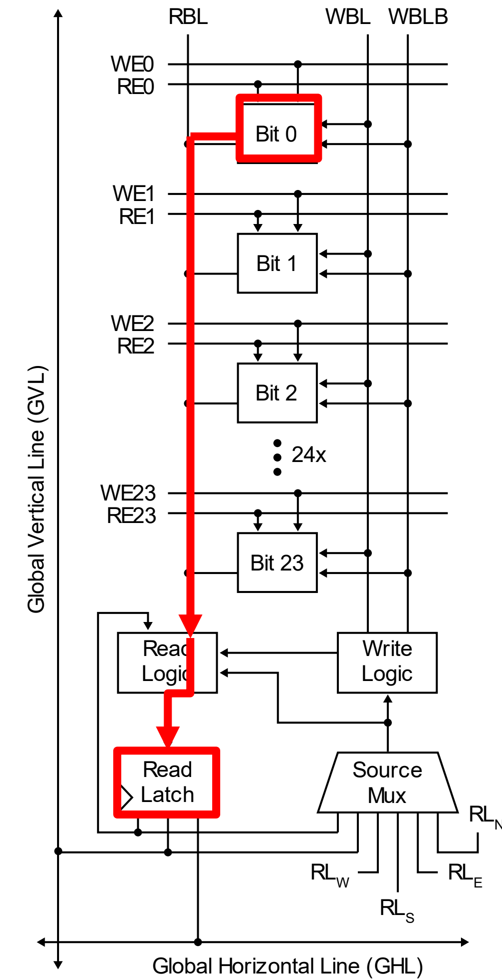
Bit Processor Circuitry

EXAMPLE #1: BITWISE OR

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



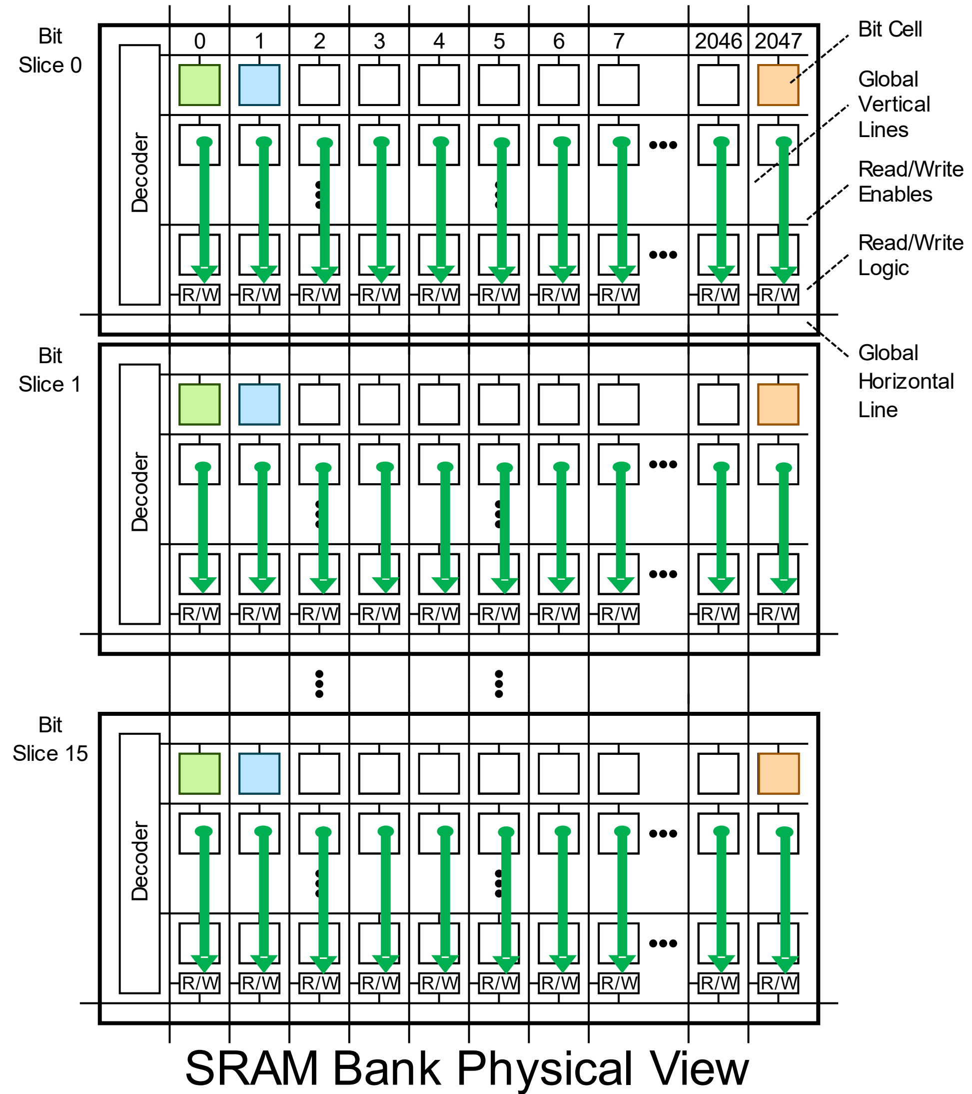
SRAM Bank Physical View



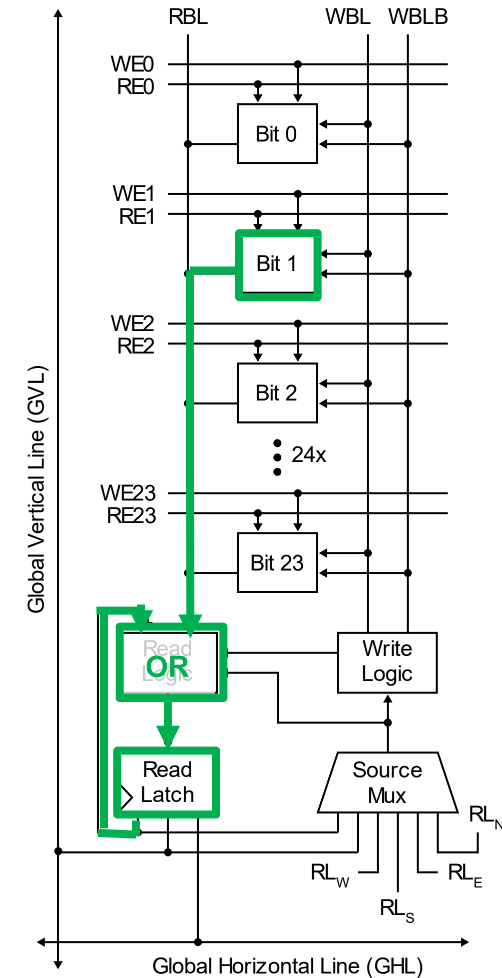
Bit Processor Circuitry

EXAMPLE #1: BITWISE OR

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



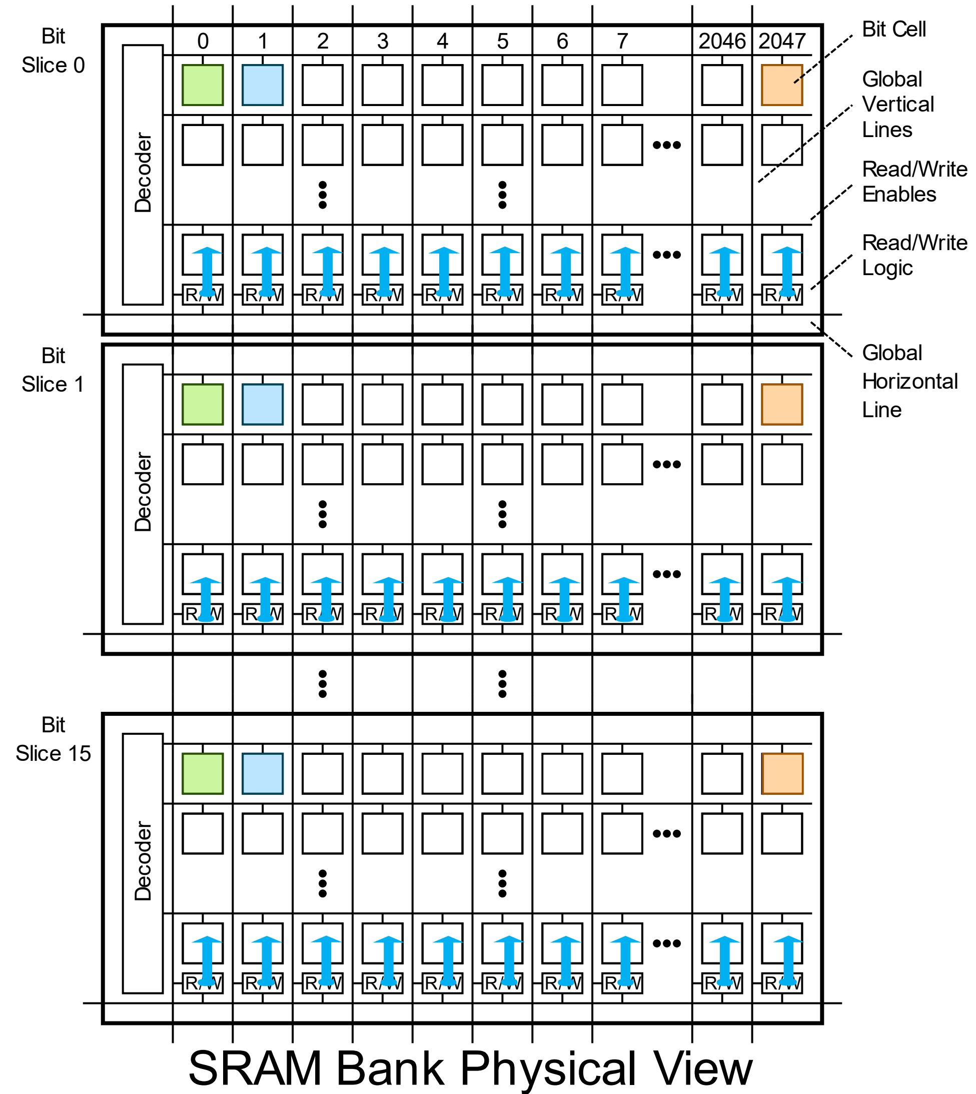
SRAM Bank Physical View



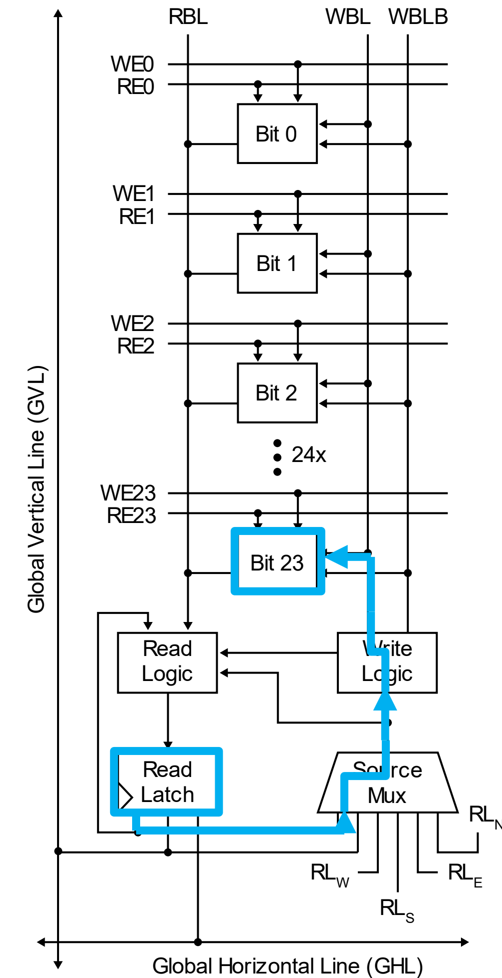
Bit Processor Circuitry

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



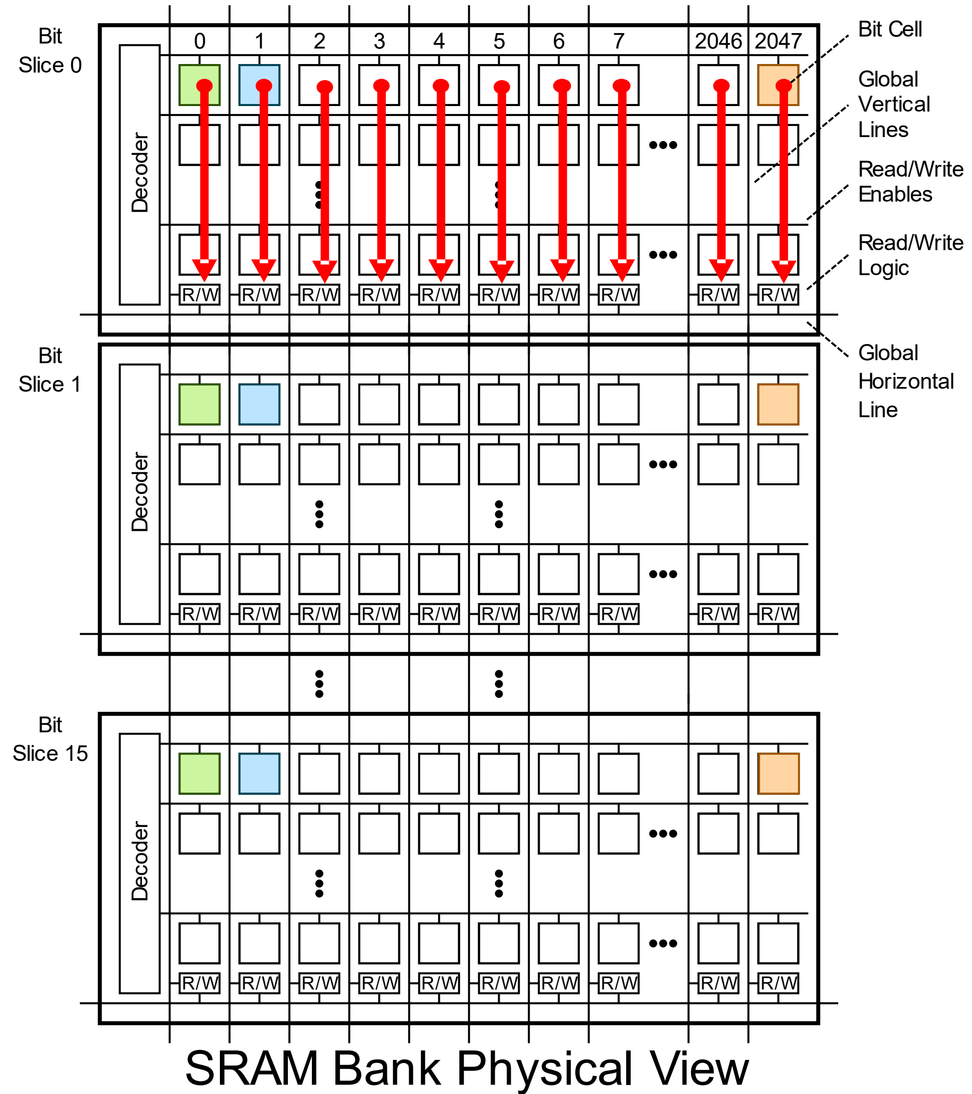
Bit Processor Circuitry

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

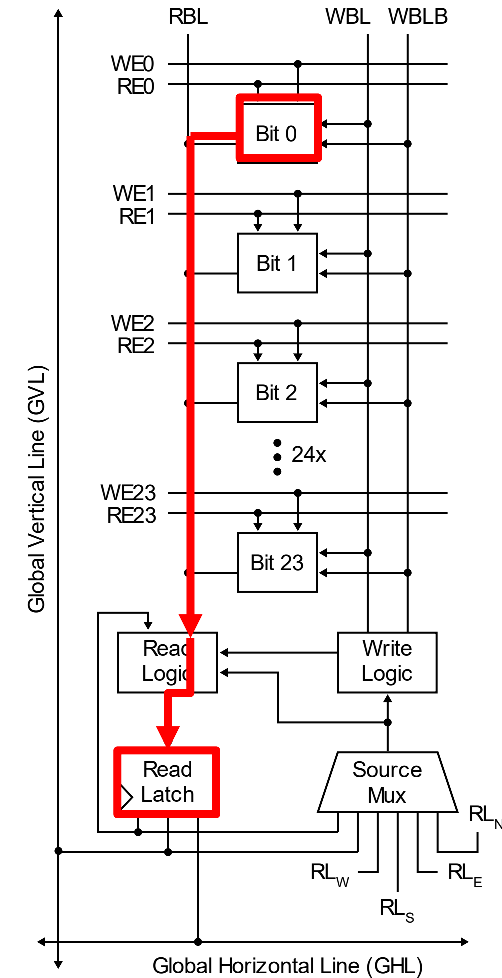
CSL

```
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



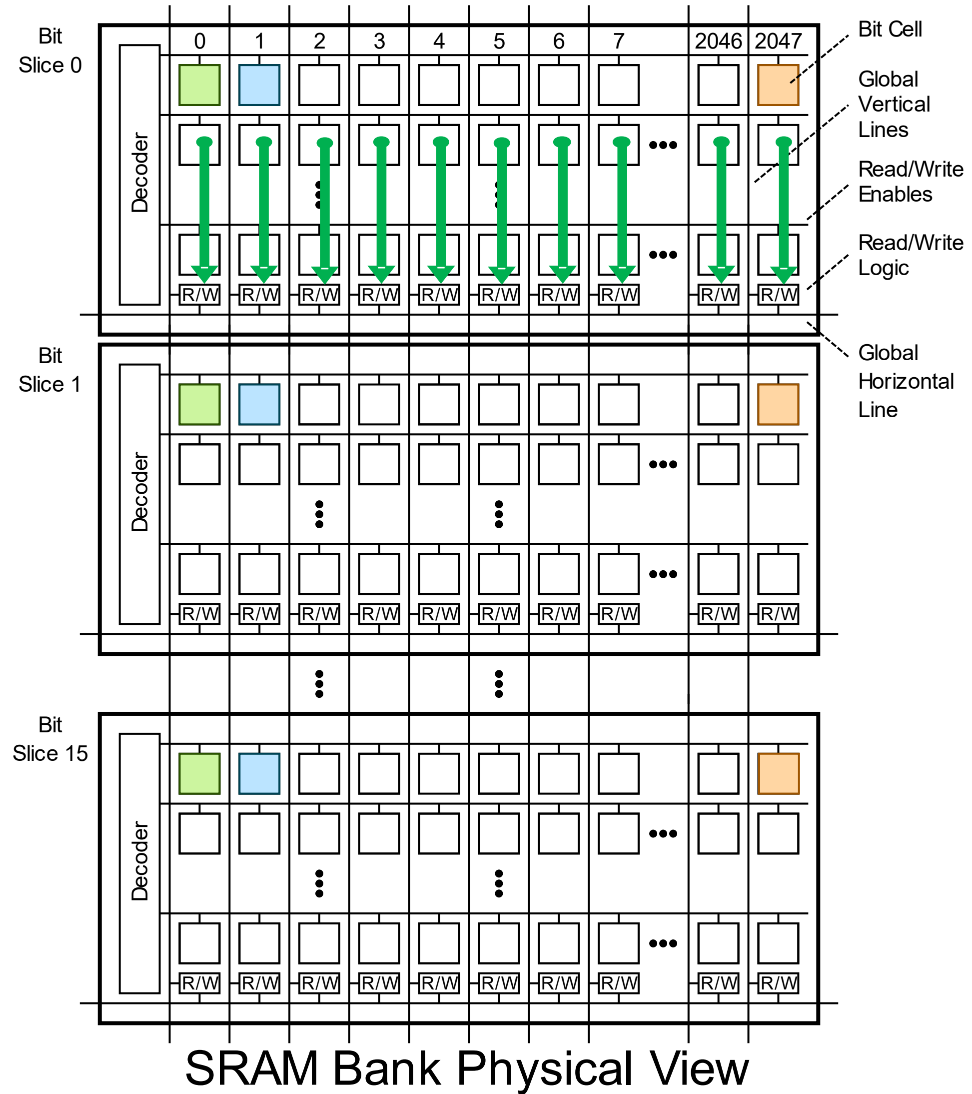
Bit Processor Circuitry

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

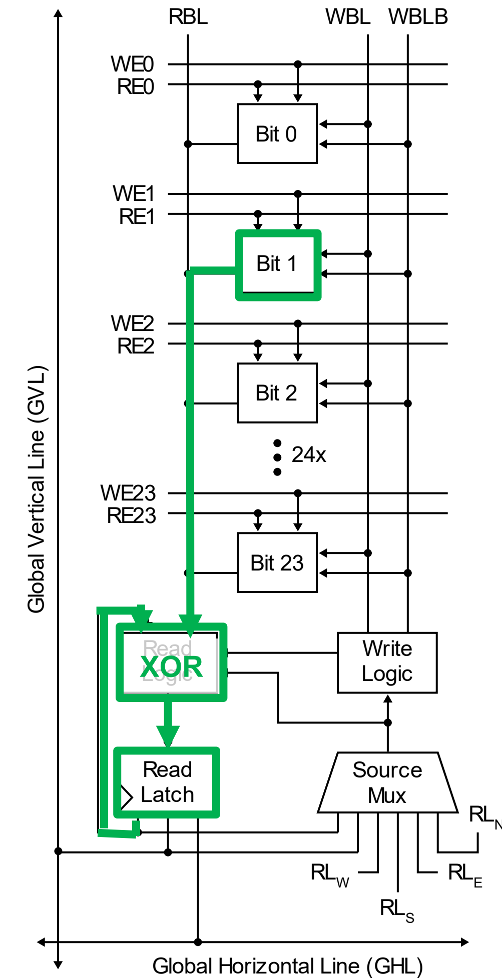
CSL

```
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

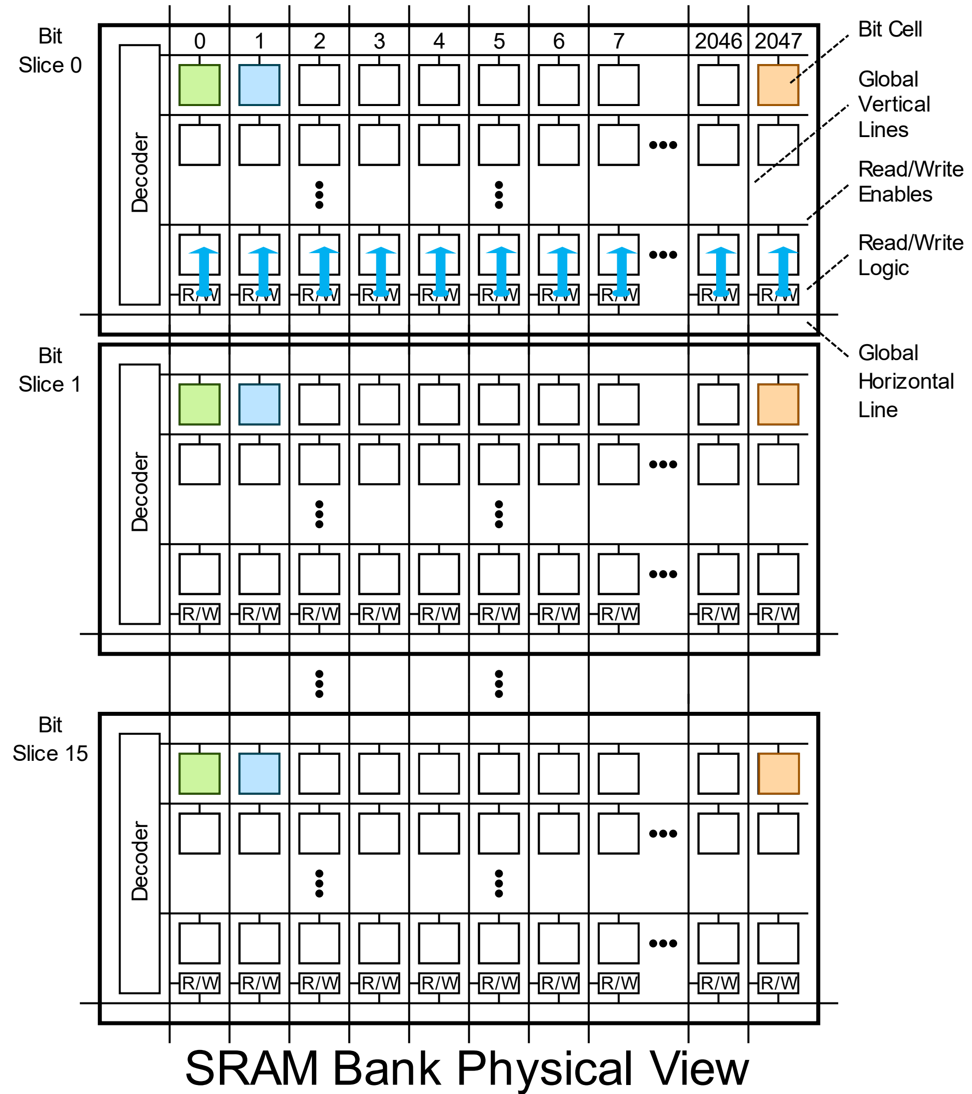


Bit Processor Circuitry

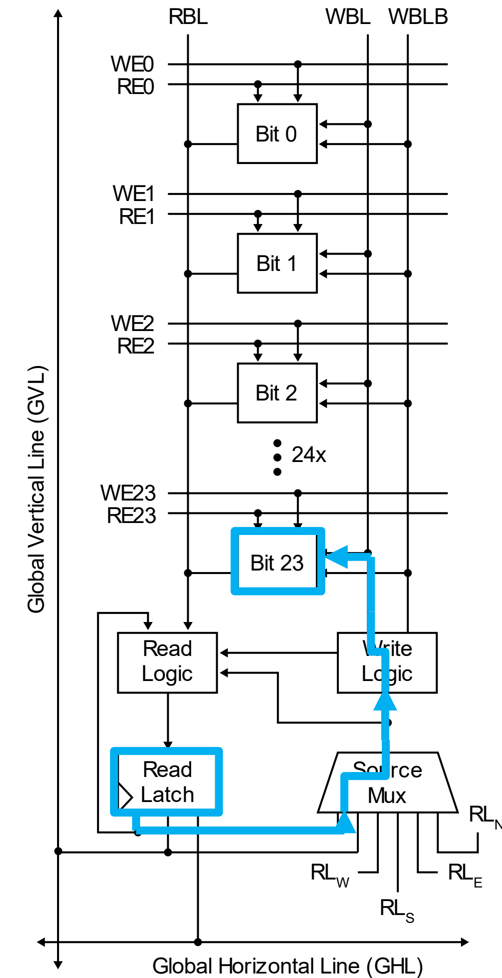
EXAMPLE #2: VECTOR-VECTOR ADD (BIT 0)


```
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



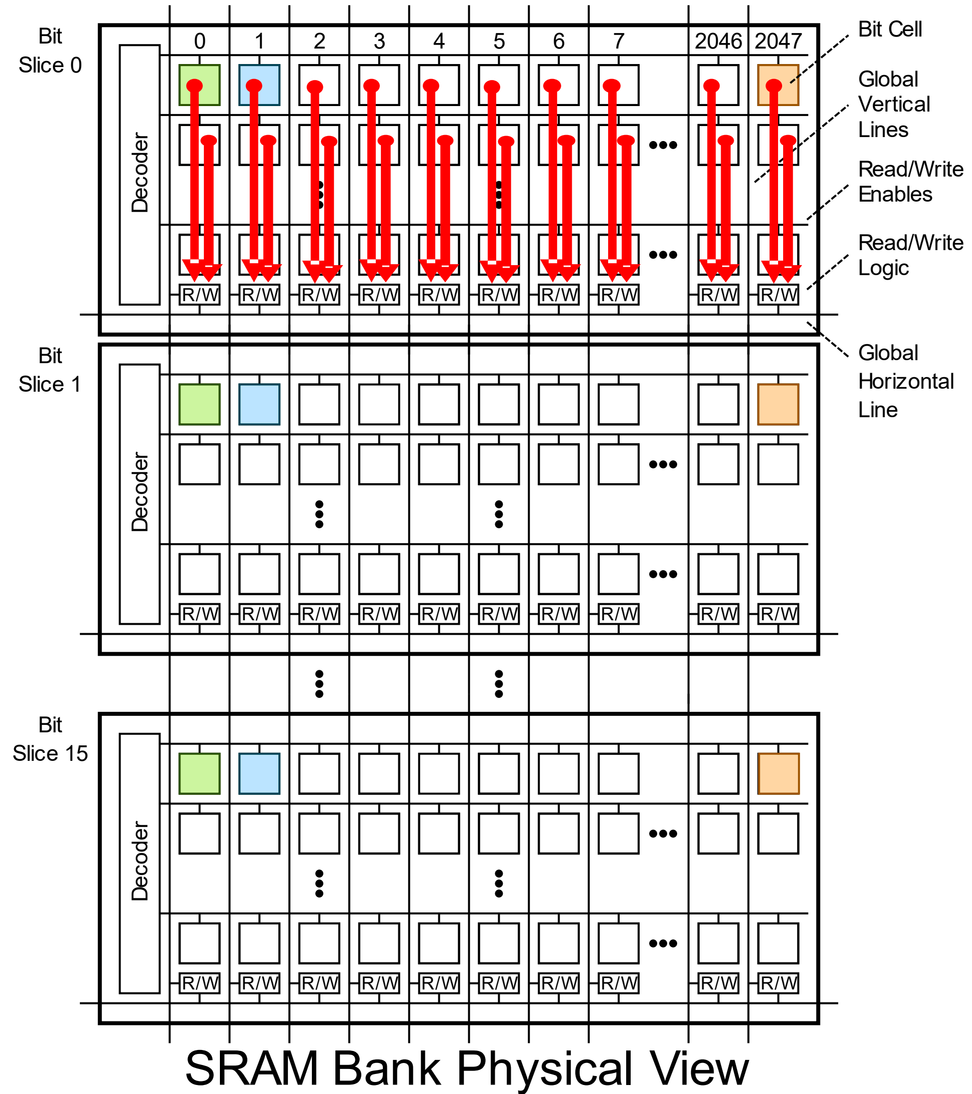
Bit Processor Circuitry

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

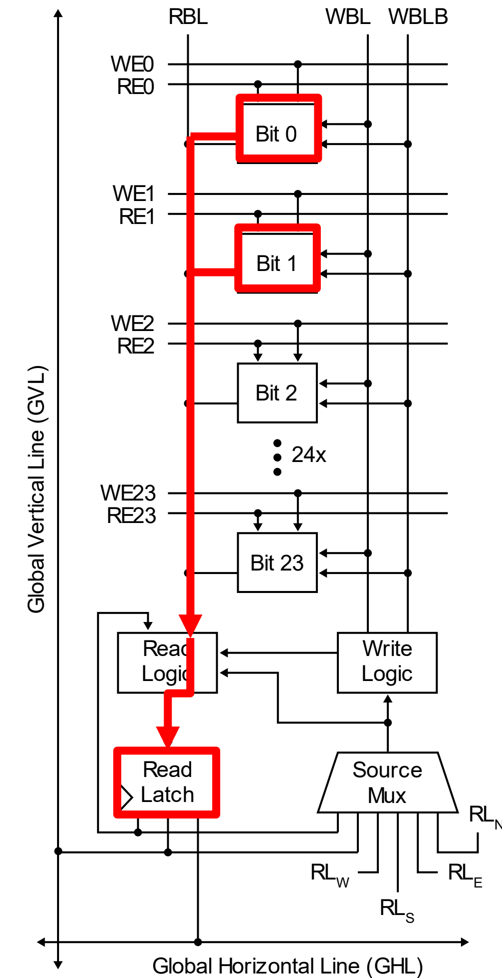
CSL

```
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



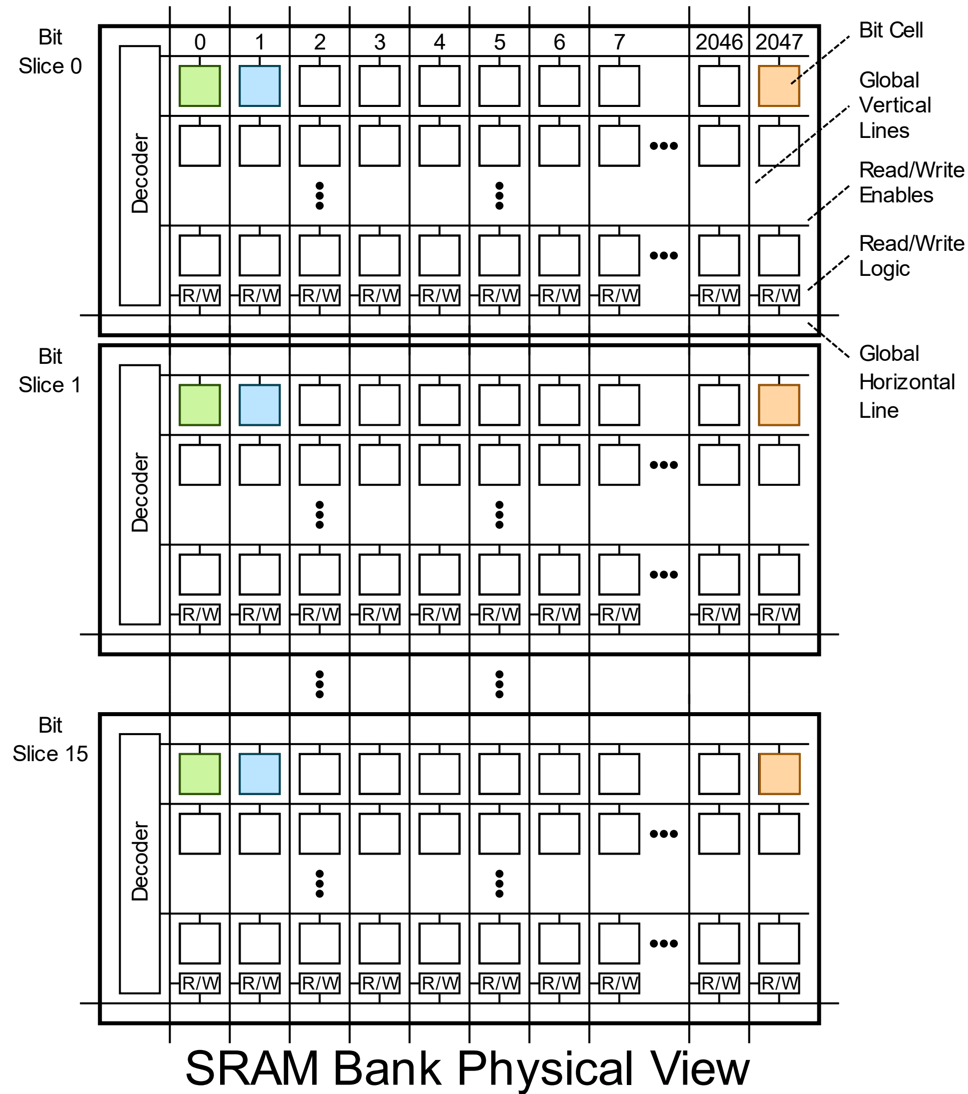
SRAM Bank Physical View



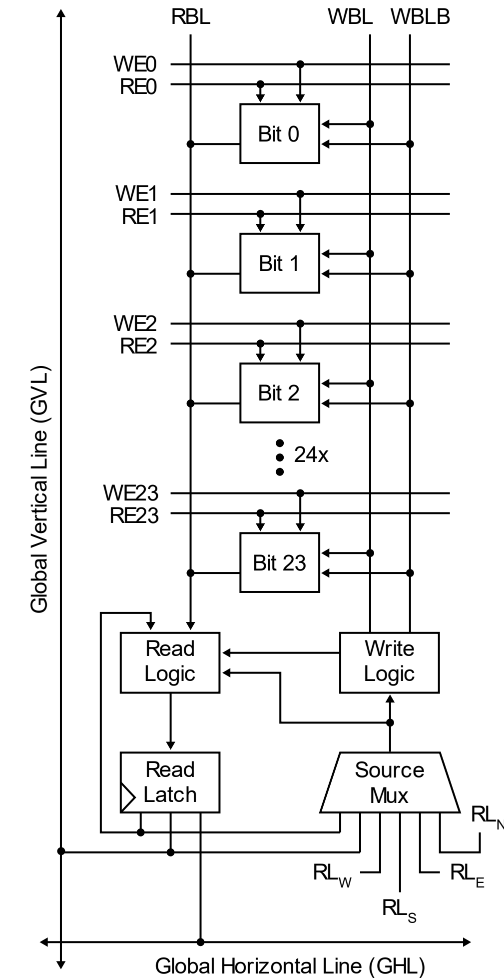
Bit Processor Circuitry

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];
  ...
```



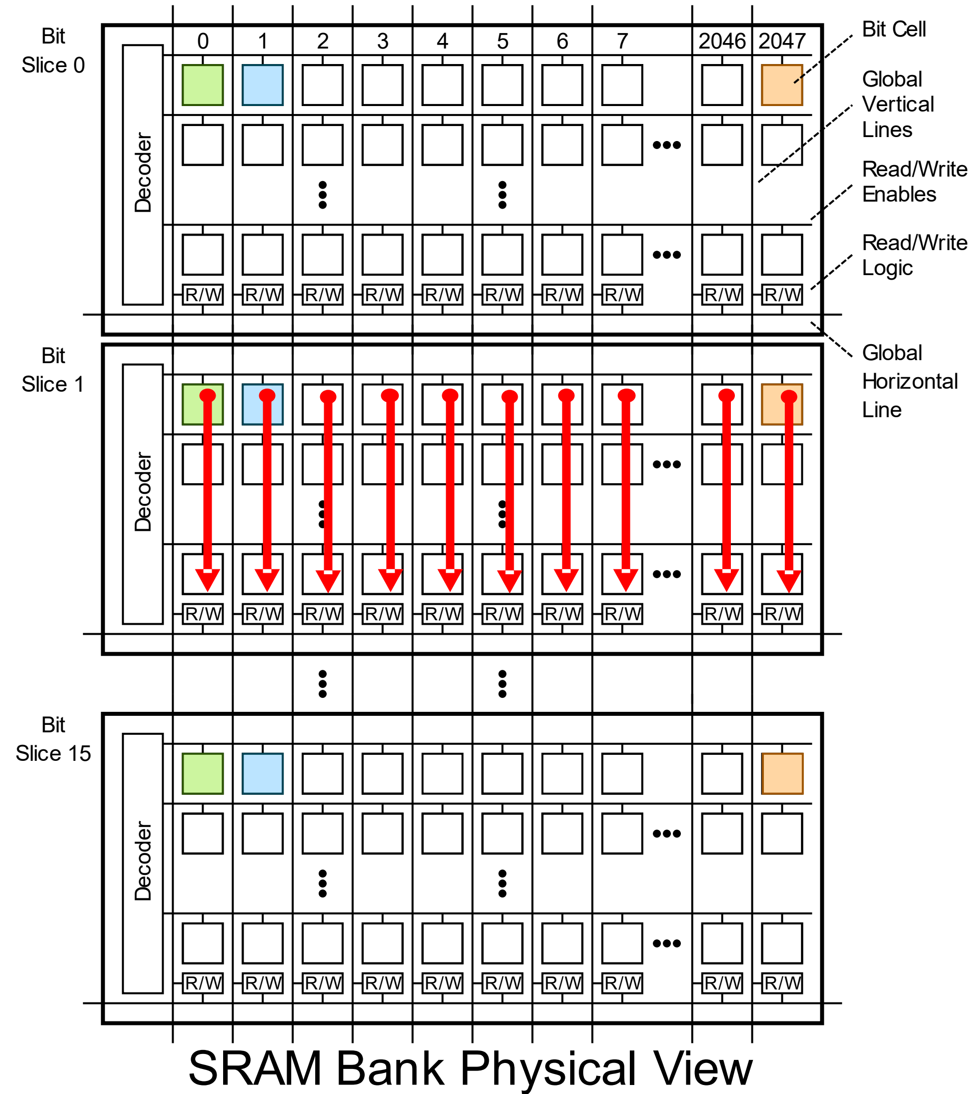
SRAM Bank Physical View



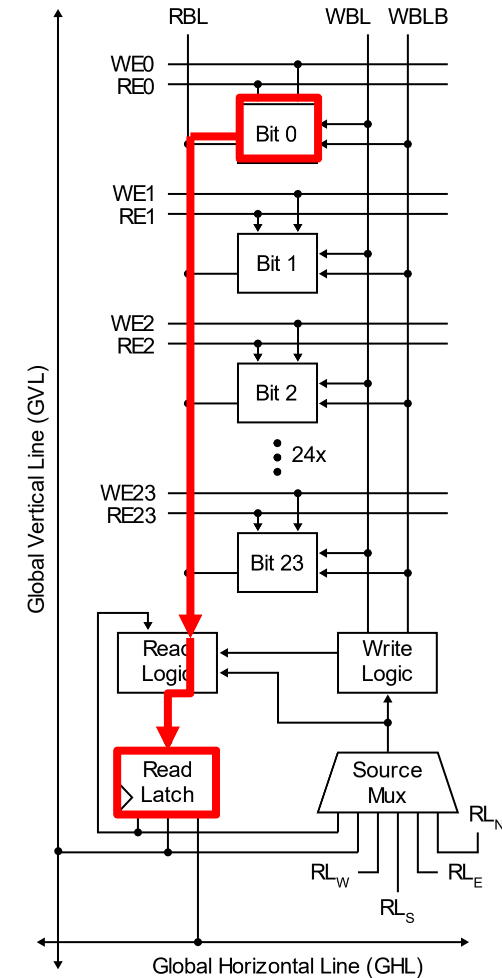
Bit Processor Circuitry

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

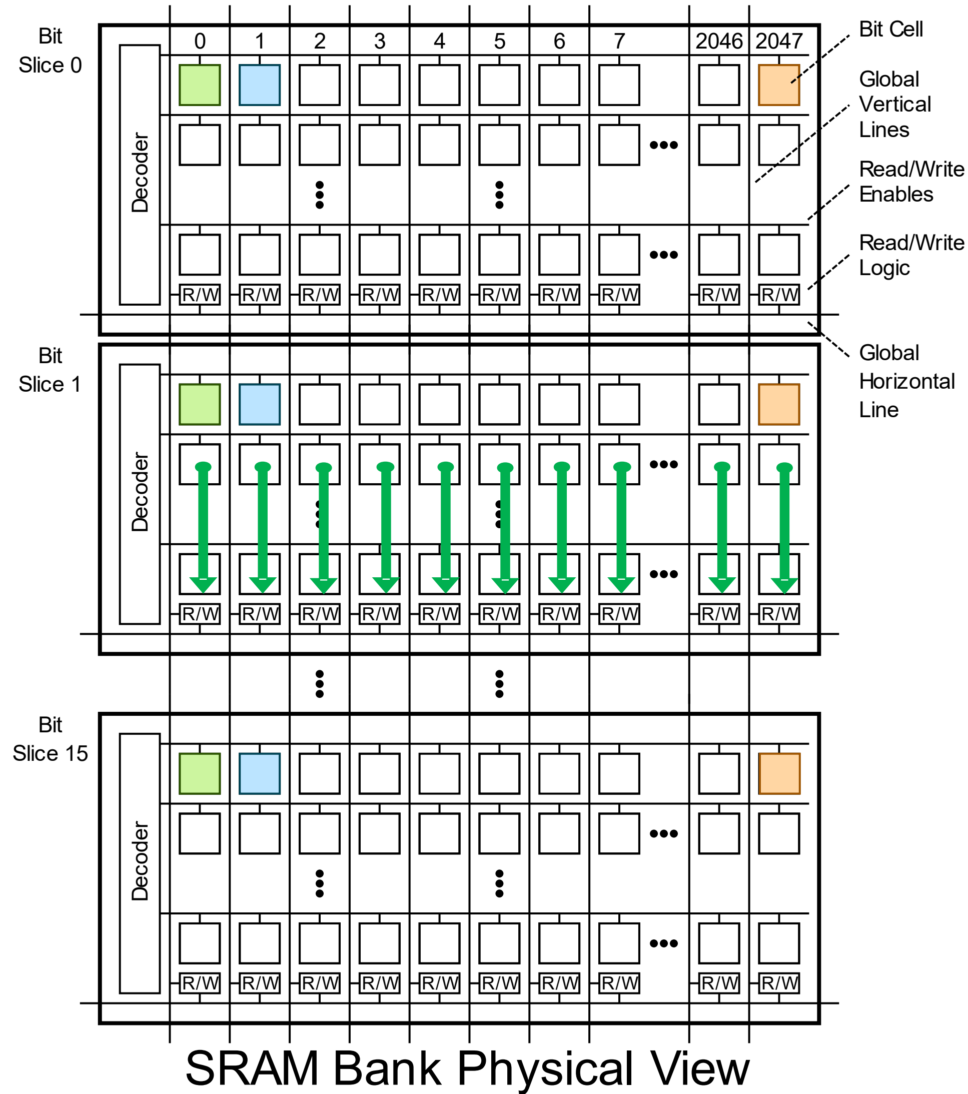


Bit Processor Circuitry

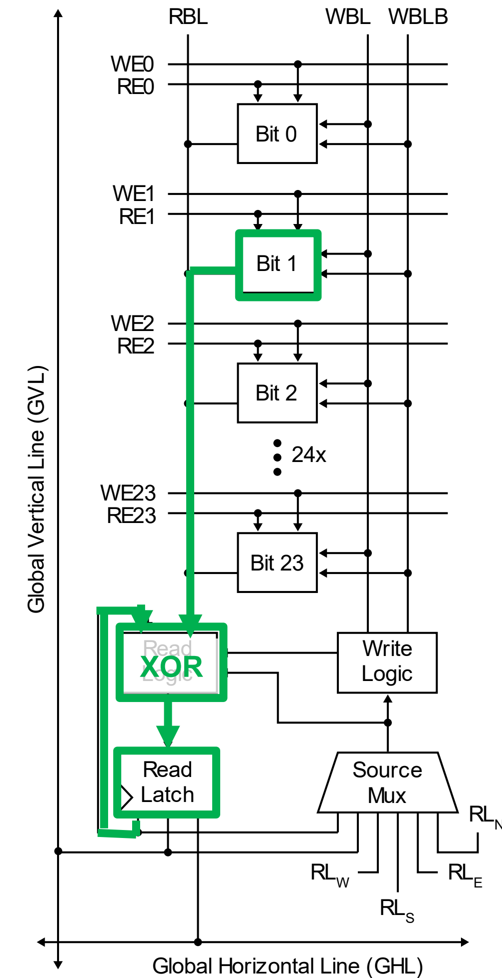
EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

CSL

```
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];
  ...
```



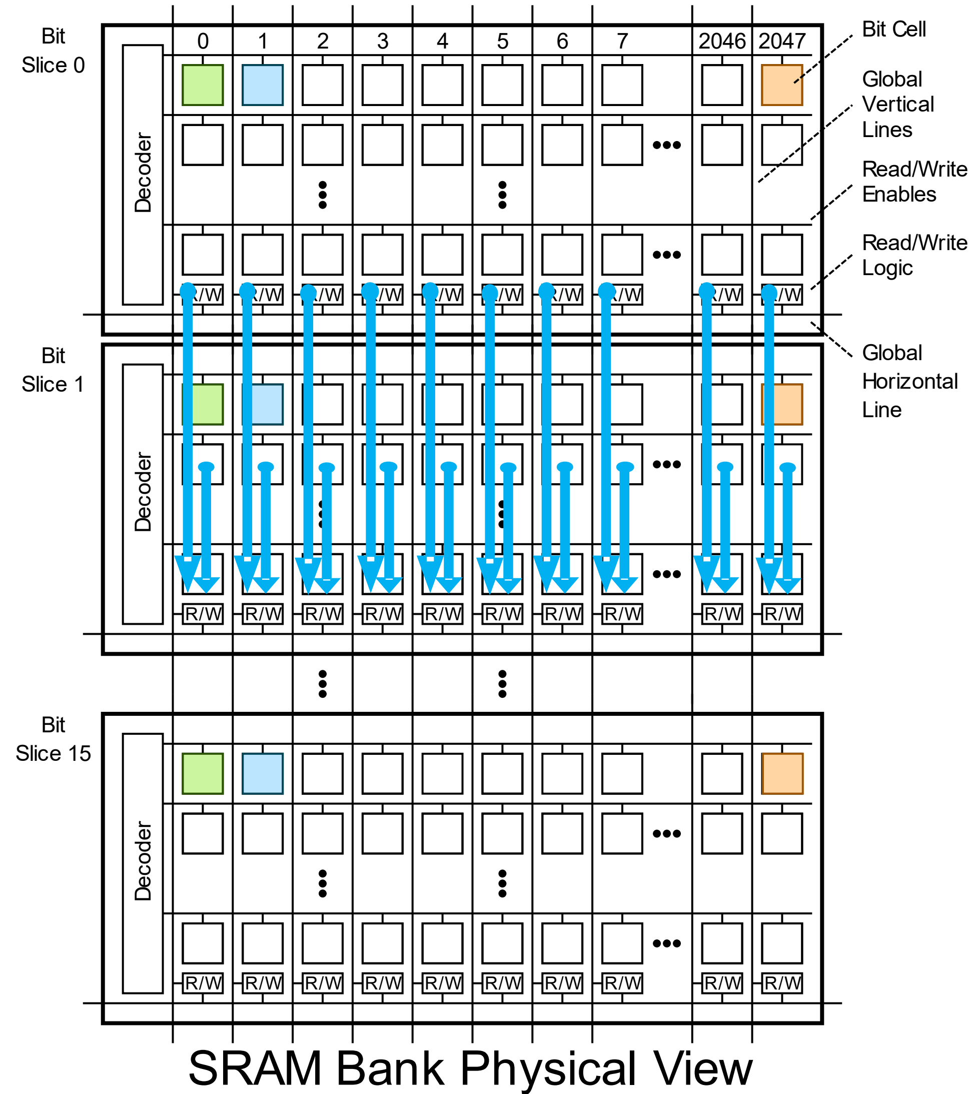
SRAM Bank Physical View



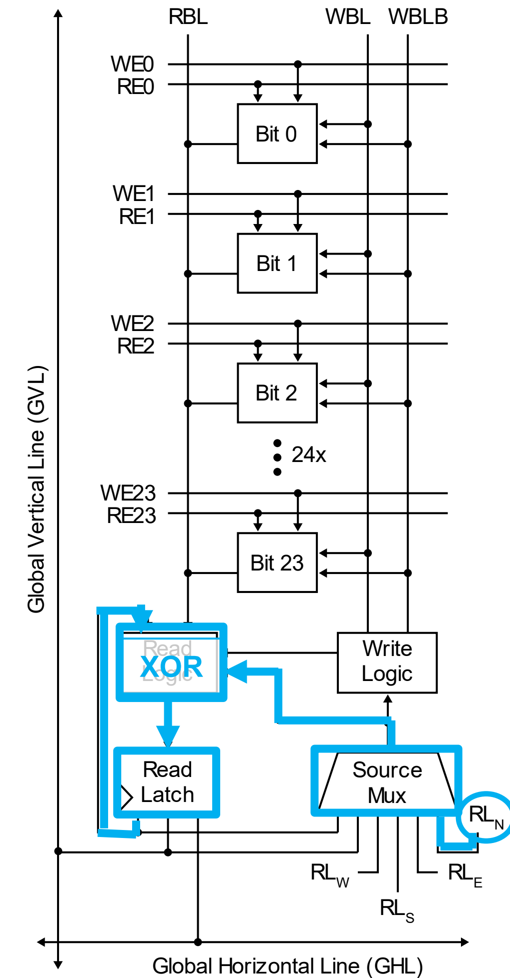
Bit Processor Circuitry

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

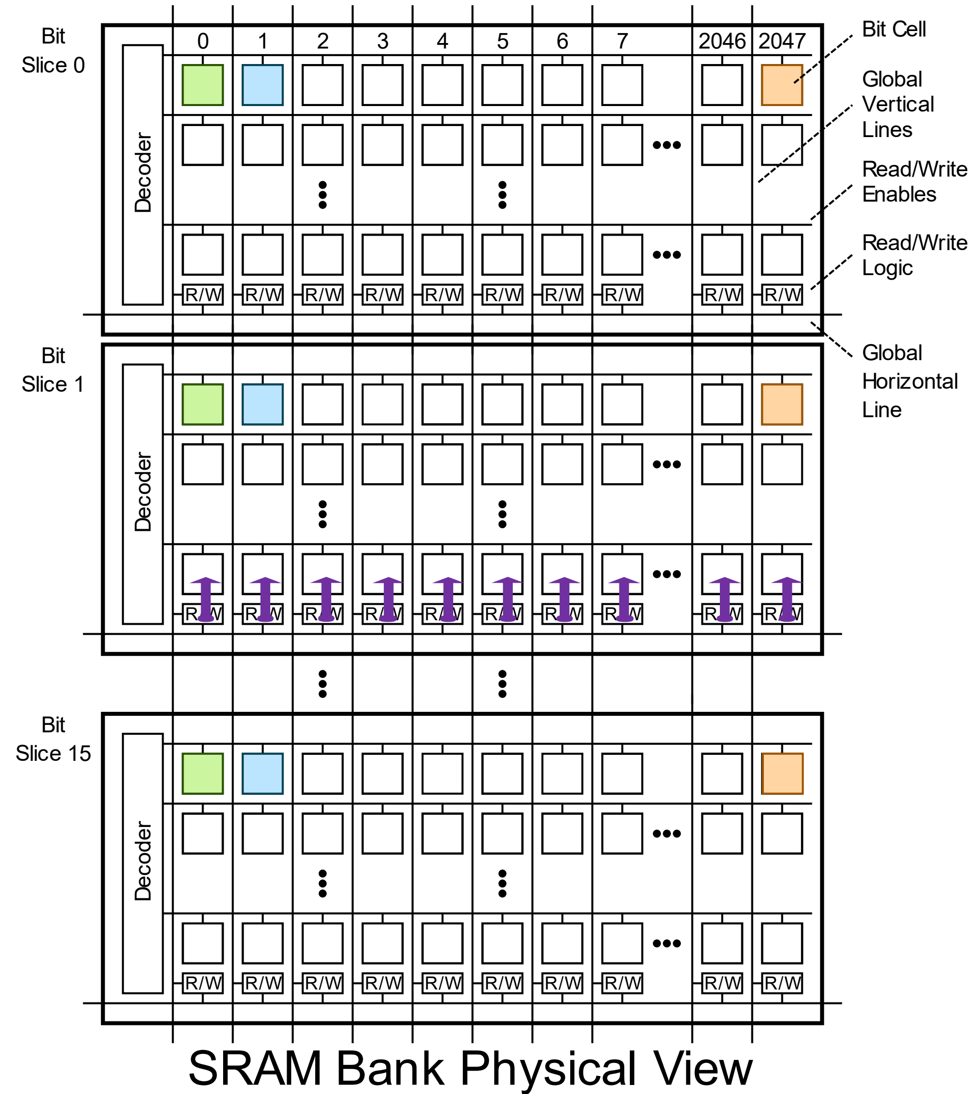


Bit Processor Circuitry

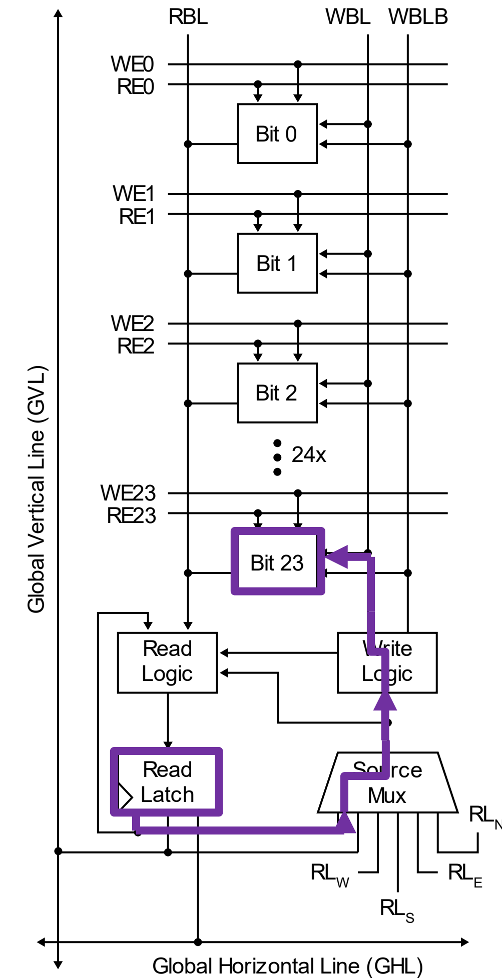
EXAMPLE #2: VECTOR-VECTOR ADD (BITS 1-15)

CSL

```
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];
  ...
```



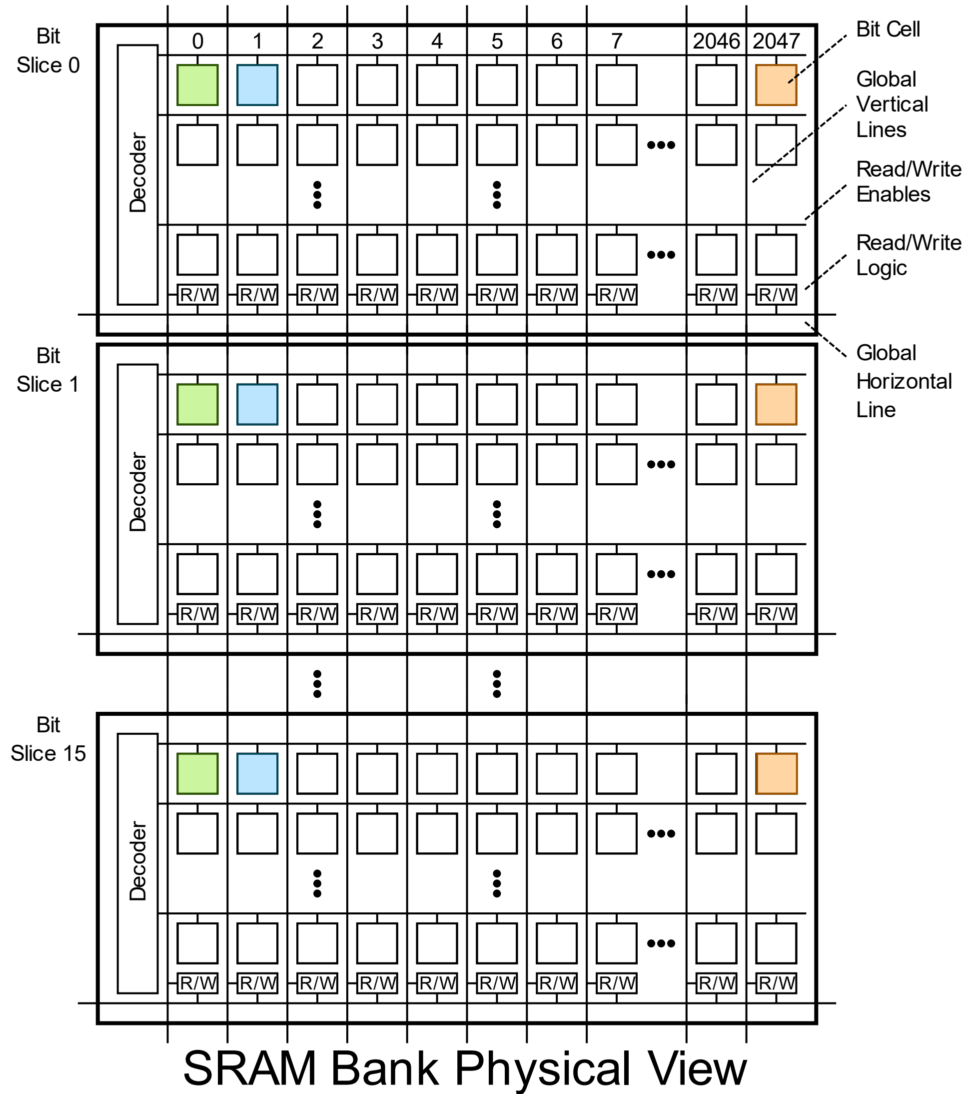
SRAM Bank Physical View



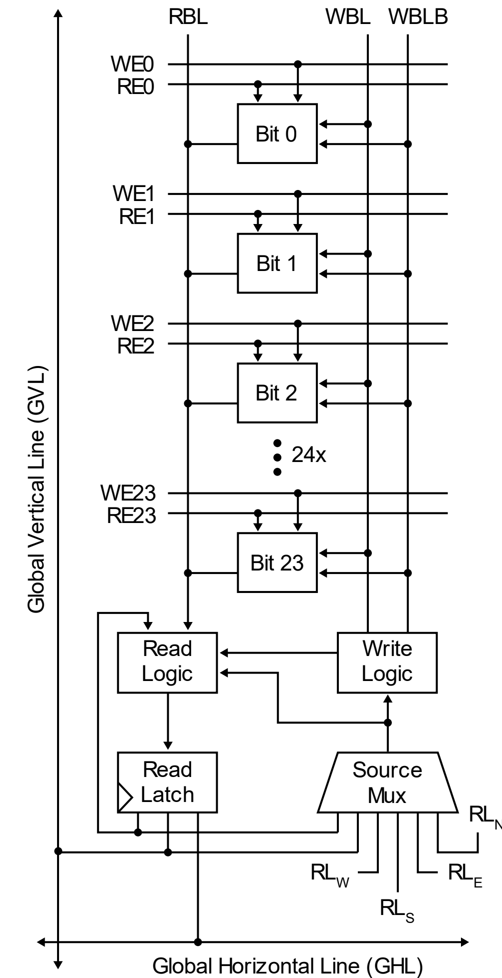
Bit Processor Circuitry

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



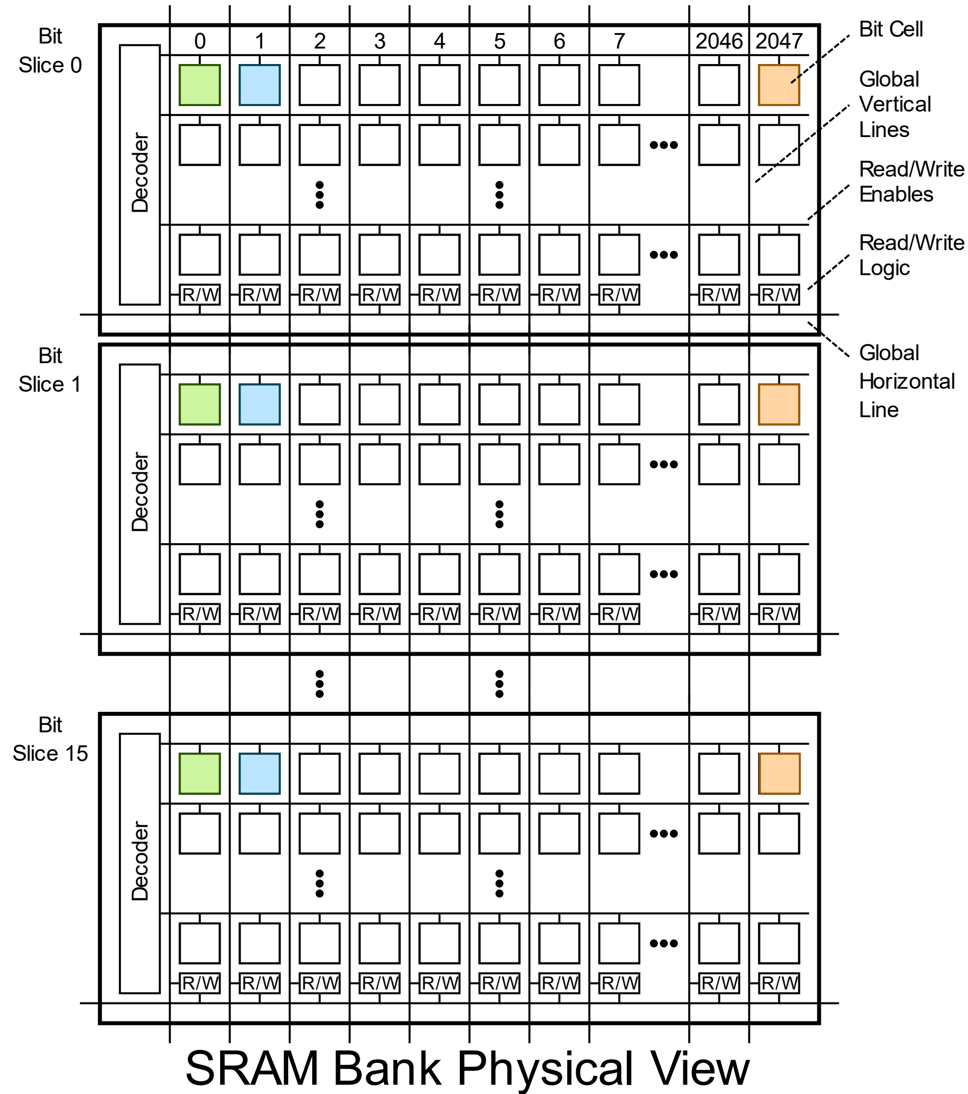
Bit Processor Circuitry

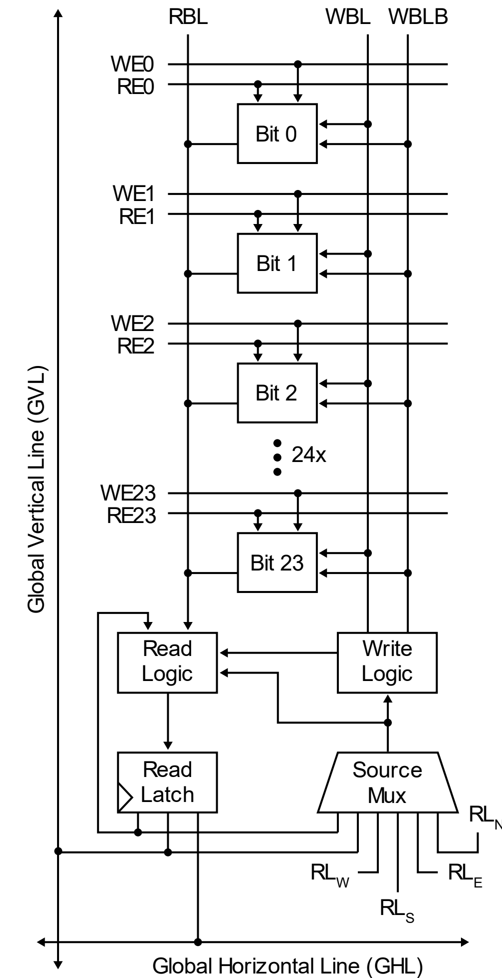
Example #2: Vector-Vector ADD (Bits 1-15)

```
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];
  ...
```

Actual implementation used is a more sophisticated carry-select approach

Ripple-carry: 215 cycles
Carry-select: 12 cycles



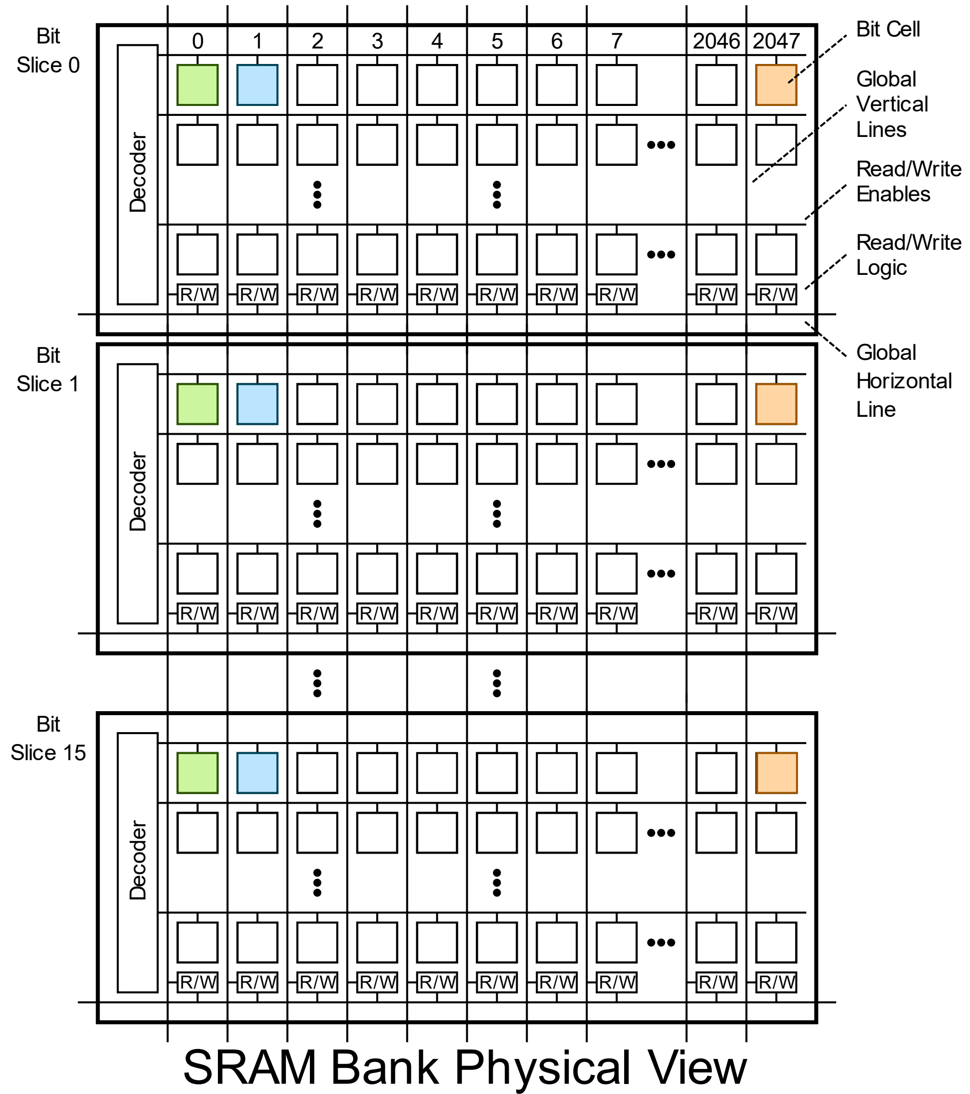


EXAMPLE #3: SEARCH

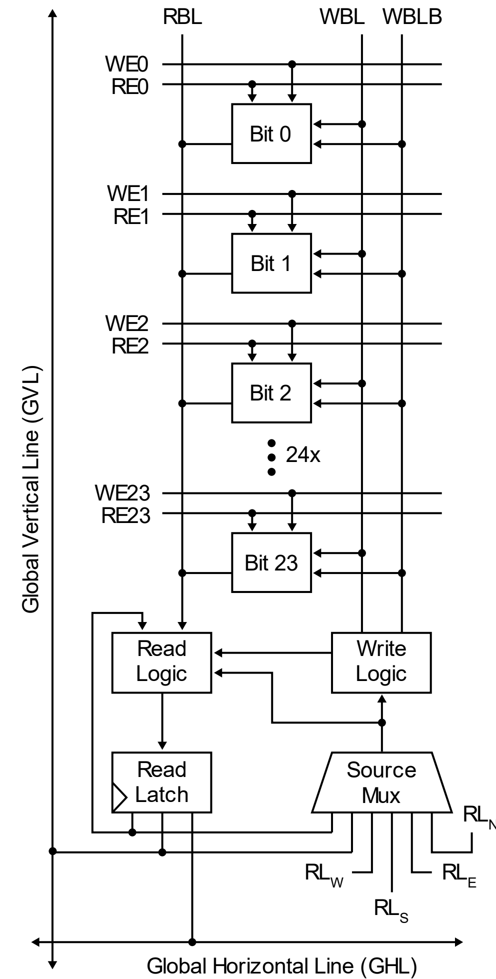
```
APL_FRAG search(vdst, vsrc, key):
  key:        RL = VRF[vsrc];
  ~key:       RL = ~VRF[vsrc];
  0xFFFF:     GVL = RL;
  0xFFFF:     VRF[vdst] = GVL;
```

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



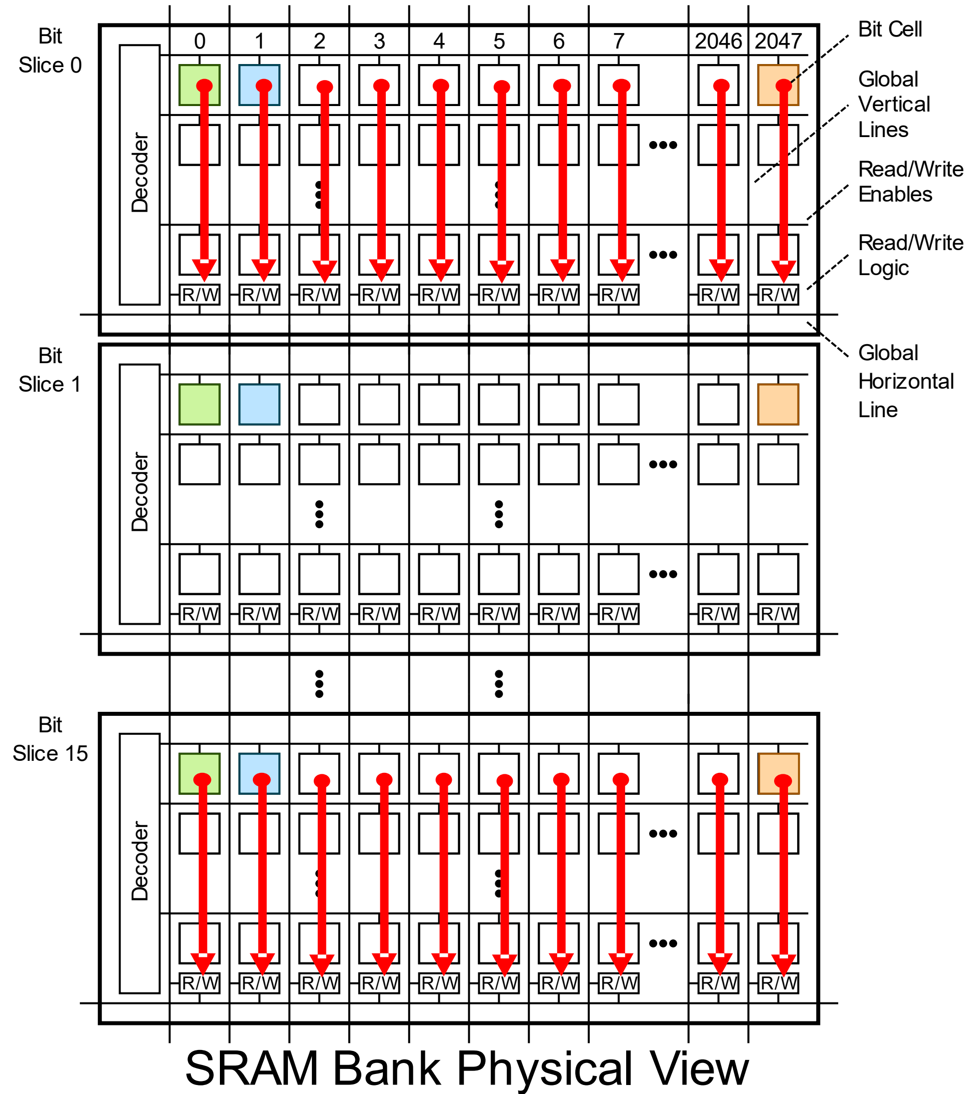
Bit Processor Circuitry

EXAMPLE #3: SEARCH

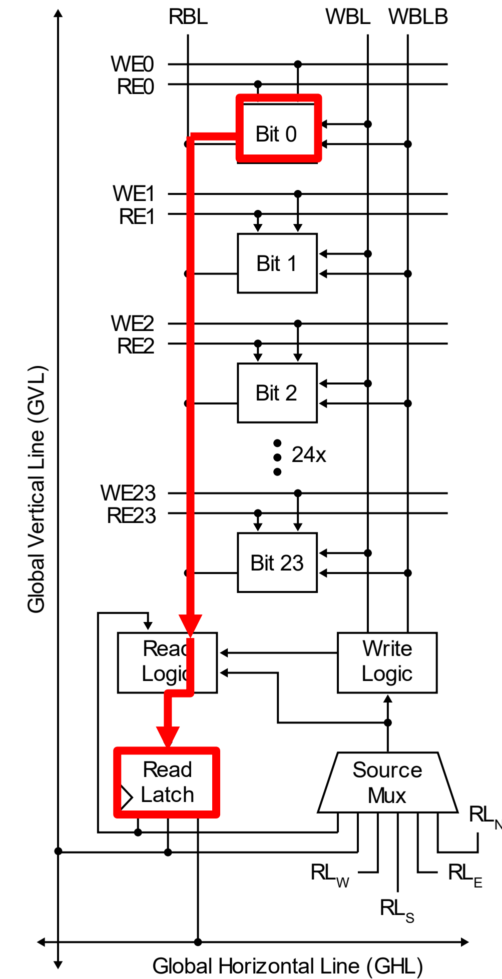
CSL

```
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

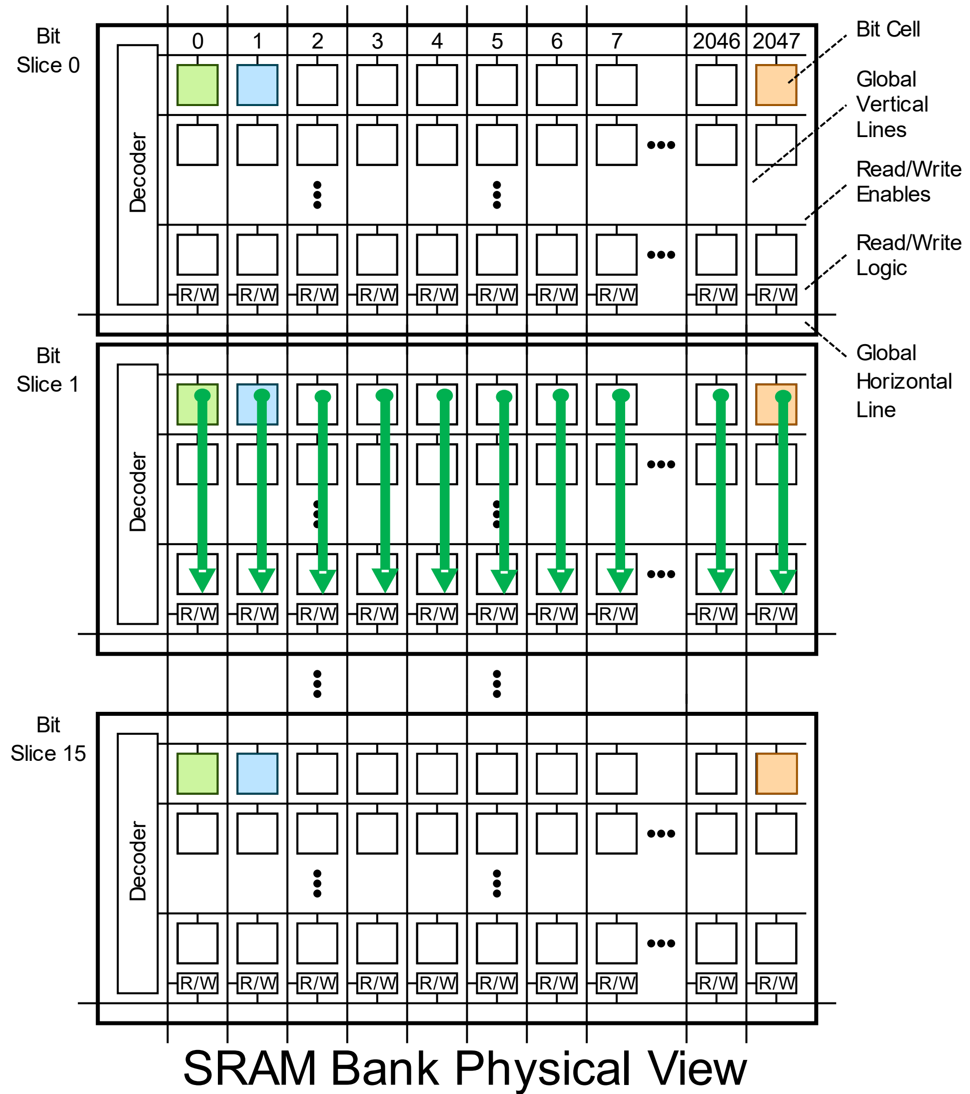
CSL

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

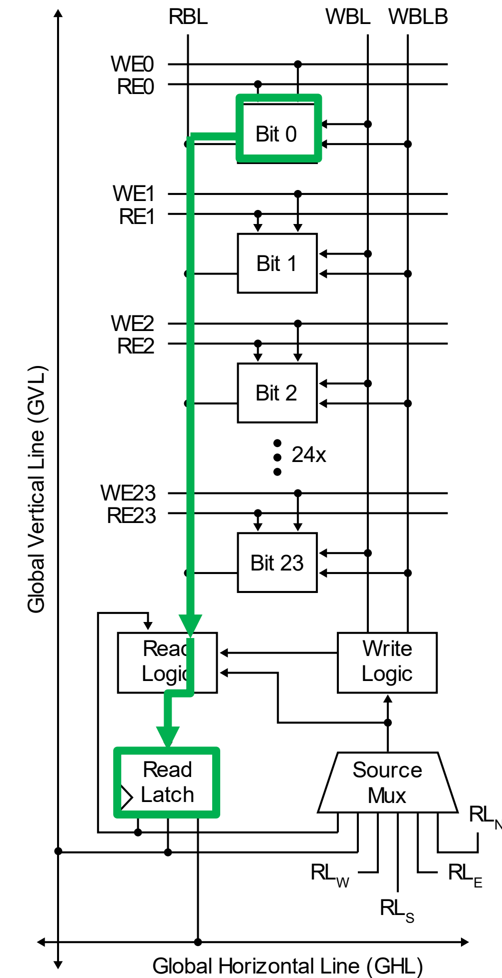
Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

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

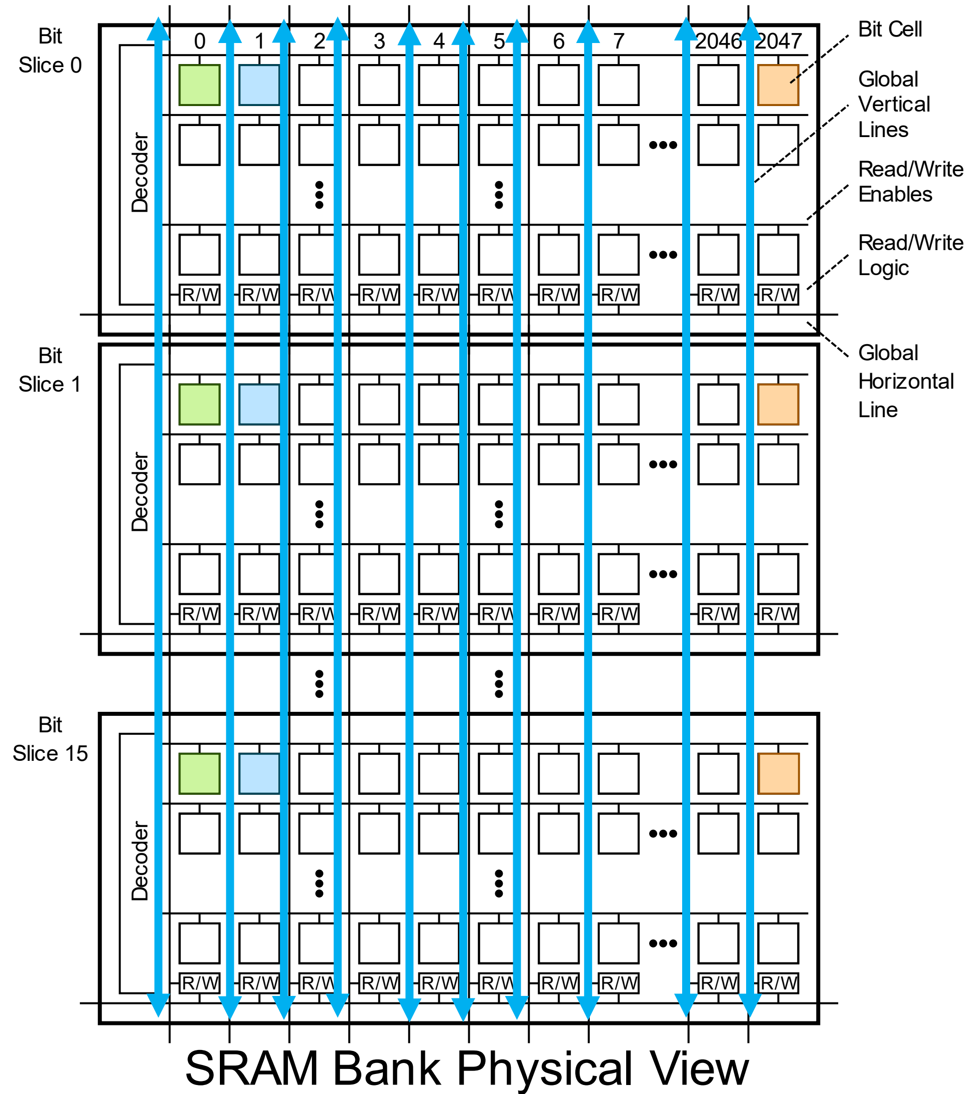
```
APL_FRAG search(vdst, vsrc, key):
  key:        RL = VRF[vsrc];
  ~key:       RL = ~VRF[vsrc];
  0xFFFF:     GVL = RL;
  0xFFFF:     VRF[vdst] = GVL;
```

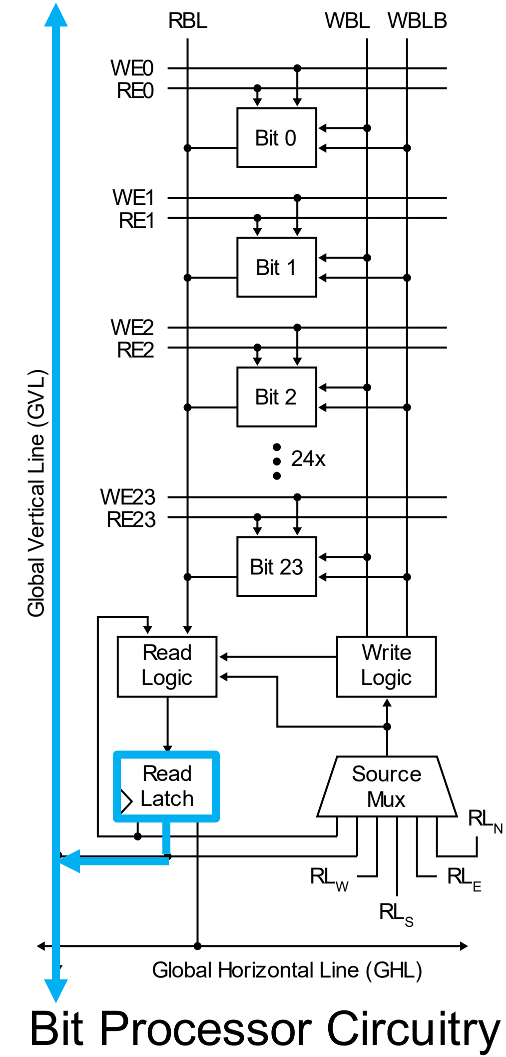
Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

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 1 1 0 0 1 0

AND





Example #3: Search

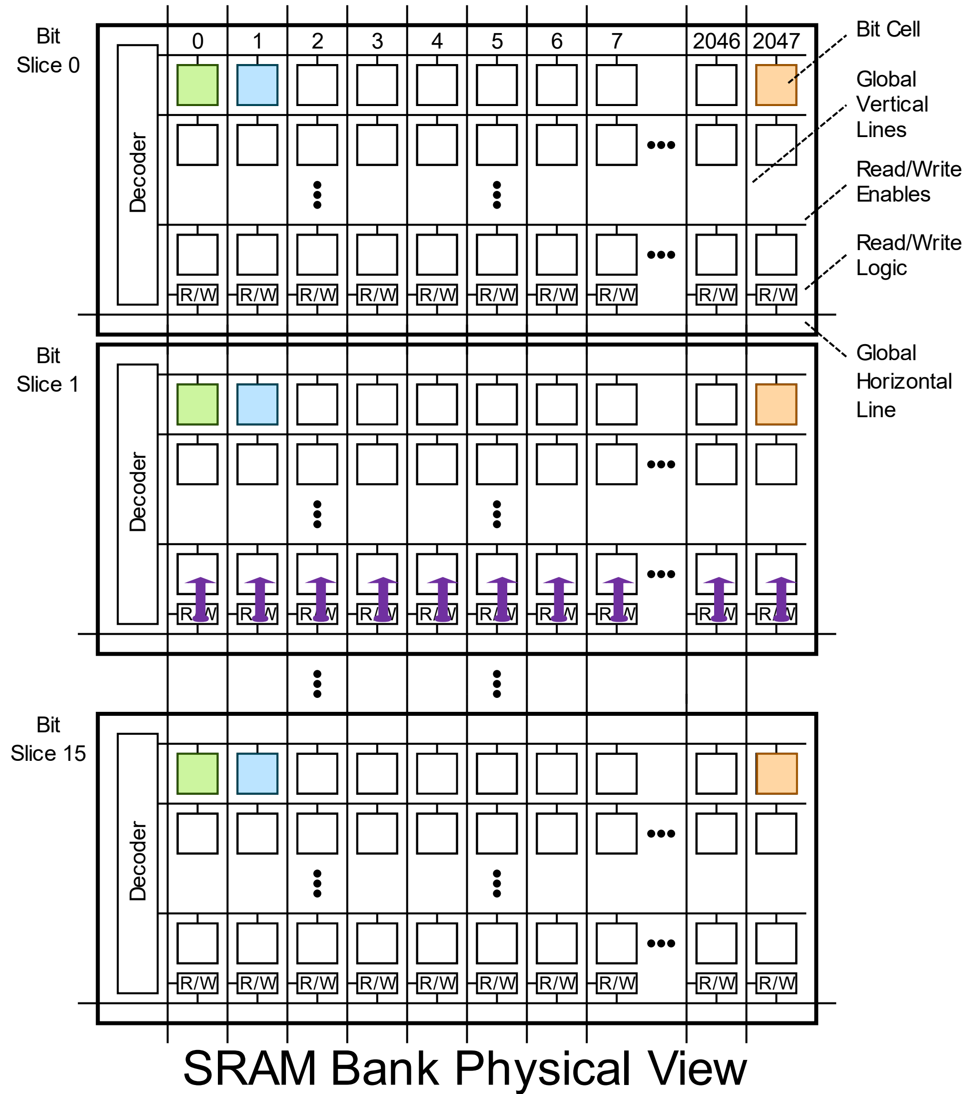
```
APL_FRAG search(vdst, vsrc, key):
  key:      RL = VRF[vsrc];
  ~key:     RL = ~VRF[vsrc];
  0xFFFF:   GVL = RL;
  0xFFFF:   VRF[vdst] = GVL;
```

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

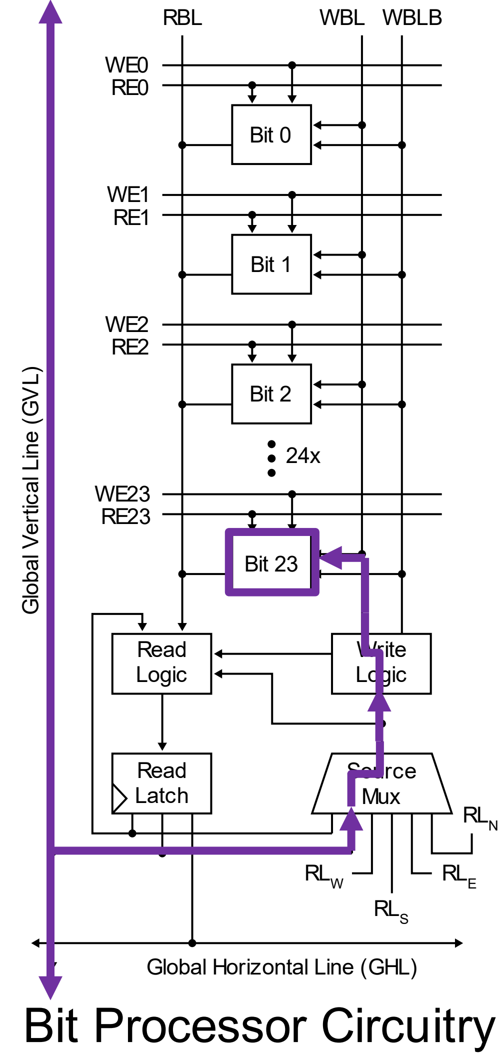
Element: 1 0 0 0 0 1 1 0 1 0 0 1 0 1 1 0

Result: 1 1 1 0 1 0 0 0 0 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;
```

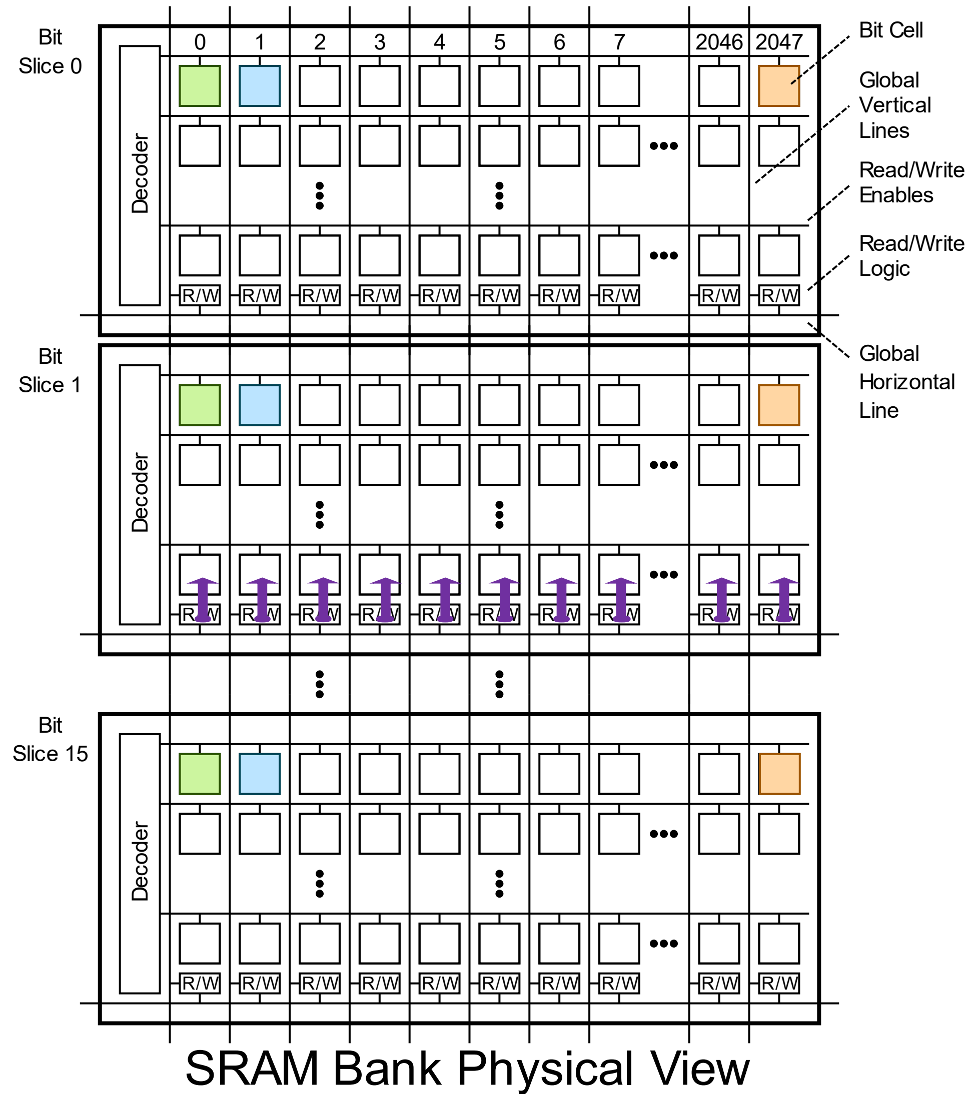
Key: 1 0 0 1 0 0 0 1 0 1 0 1 1 0 1 1

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

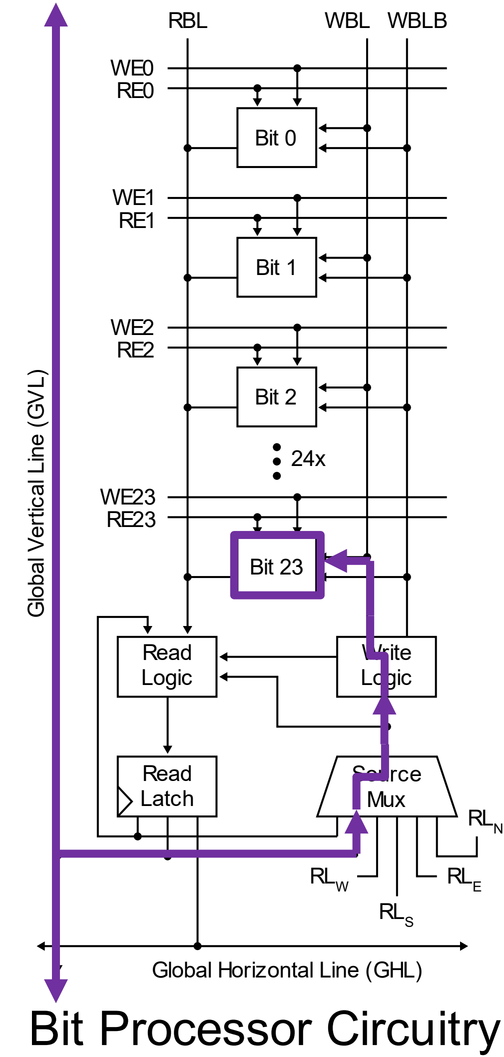
Result: 1 1 1 0 1 0 0 0 0 1 1 0 0 1 0

AND = 0

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



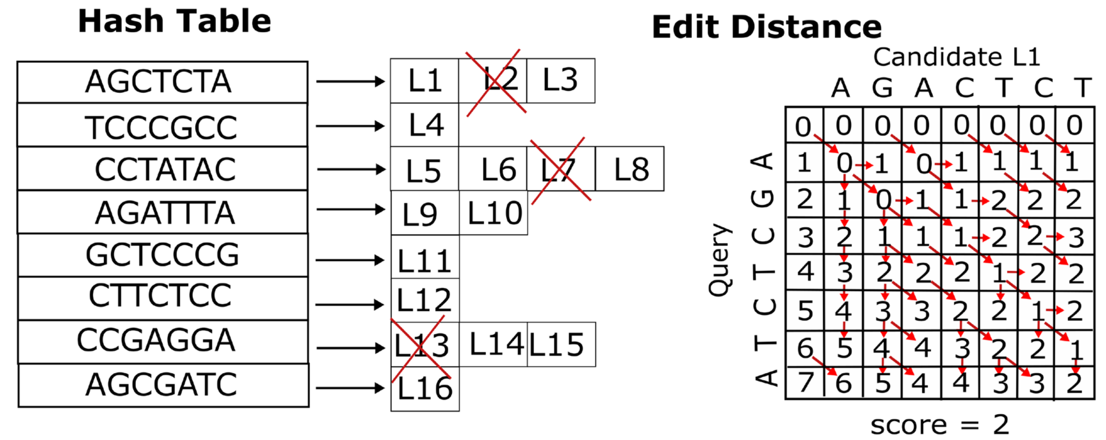
SRAM Bank Physical View



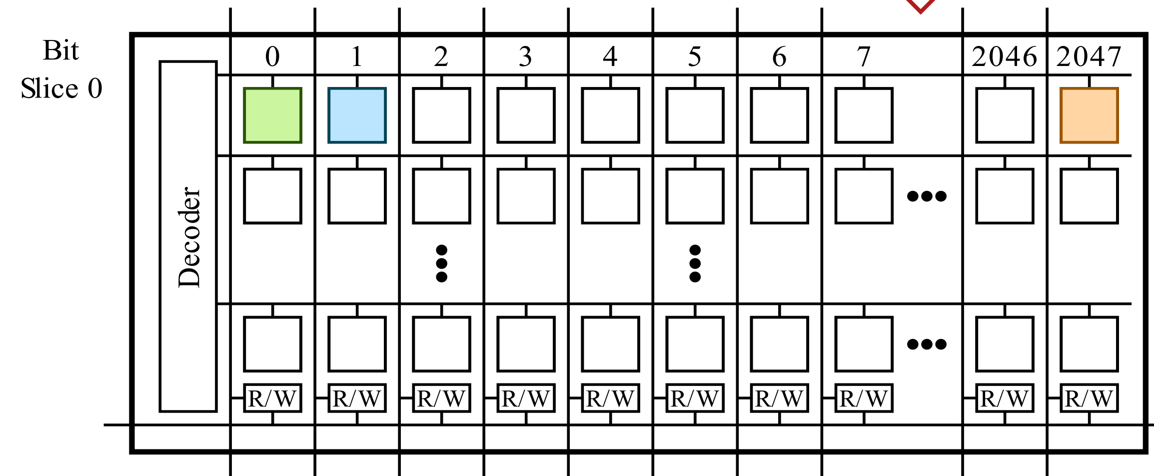
Bit Processor Circuitry

Accelerating Seed Location Filtering in DNA Read Mapping Using a Commercial Compute-in-SRAM Architecture

- Motivation
- APU Microarchitecture
- APU Microcoding
- **APU for Myers' Acceleration**




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



GENOMICS PIPELINE

Reference Genome

••• CTAGTTCCCTCGGAGAGTCTCCCCTTTTTGGGAAAATCTAGACTCTAGAGGAGACTCTCAAGAGATCTCTGATGTCCTGATAGCTATC •••

Read CAGCTCTAG

Kmers
CAGC
AGCT
GCTC
CTCT
•••

Hash Table

Kmer	Locations
CTCT	L1, ~~L2~~, L3
AGCT	L4
GCTC	L5, L6, ~~L7~~, L8
CAGC	L9, L10
ATCT	L11
GGAC	L12
CATG	~~L13~~, L14, L15
TGAC	~~L16~~

Seed and Extend

Query

CTCT ⟶ AGCTCTA

Candidate L1

CTCT ⟶ TGCTCTC

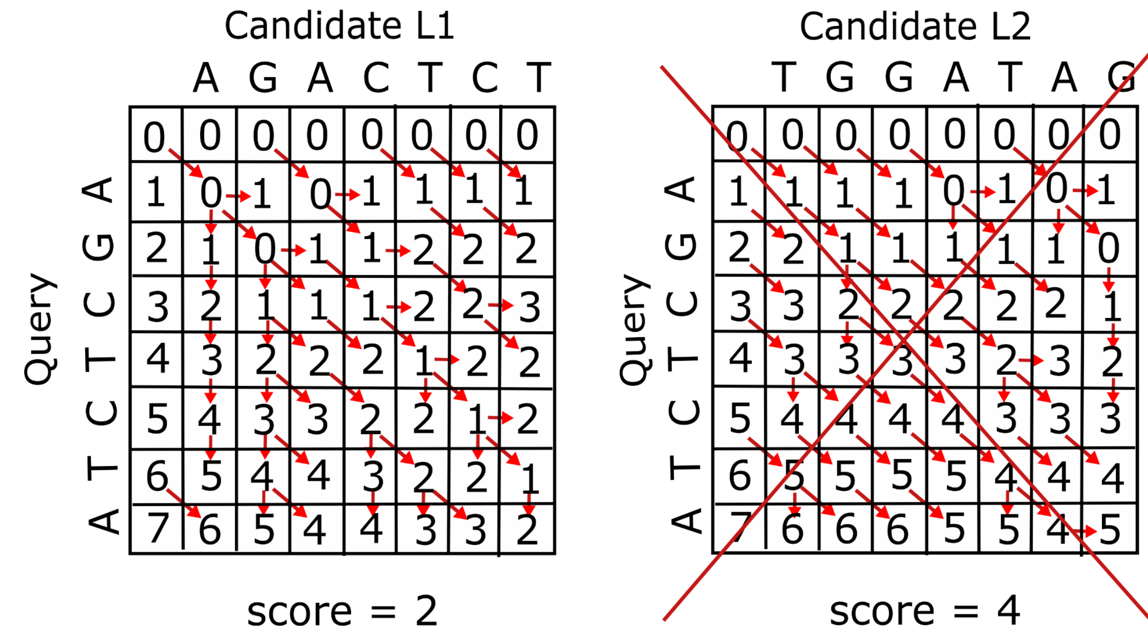
Candidate L2

CTCT ⟶ CCCTCTG

Candidate L3

CTCT ⟶ TGCTCTT

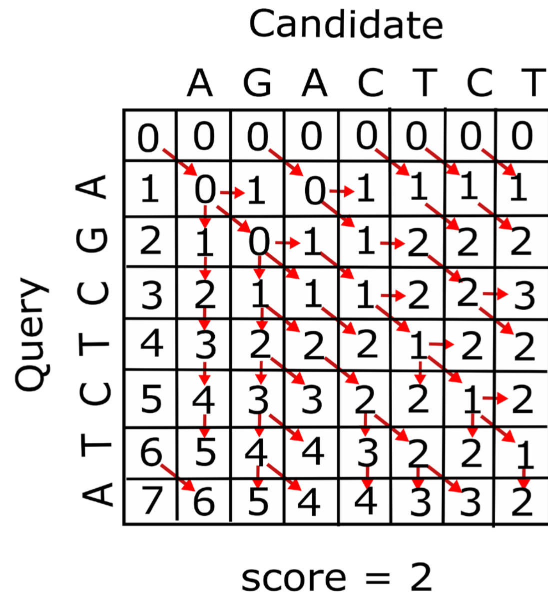
Filtering using Edit Distance



Final accurate alignment uses gap-affine distance

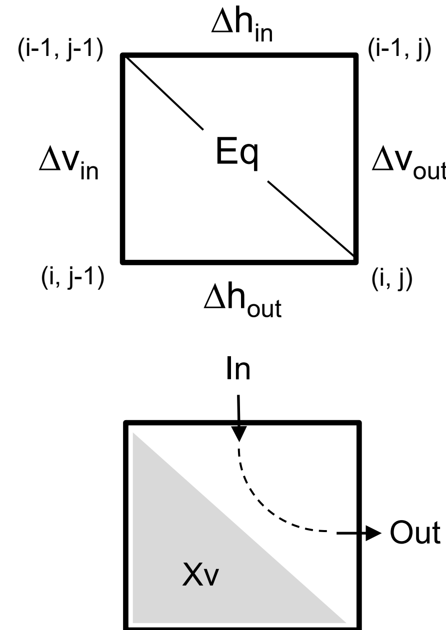
MYERS' BIT-PARALLEL ALGORITHM

Traditional Edit Distance:




```
let match = (query[i] == candidate[i])

S[i,j] = min { S[i-1, j-1] + !match,
               S[i-1,j]    + 1,
               S[i,j-1]    + 1      }
```

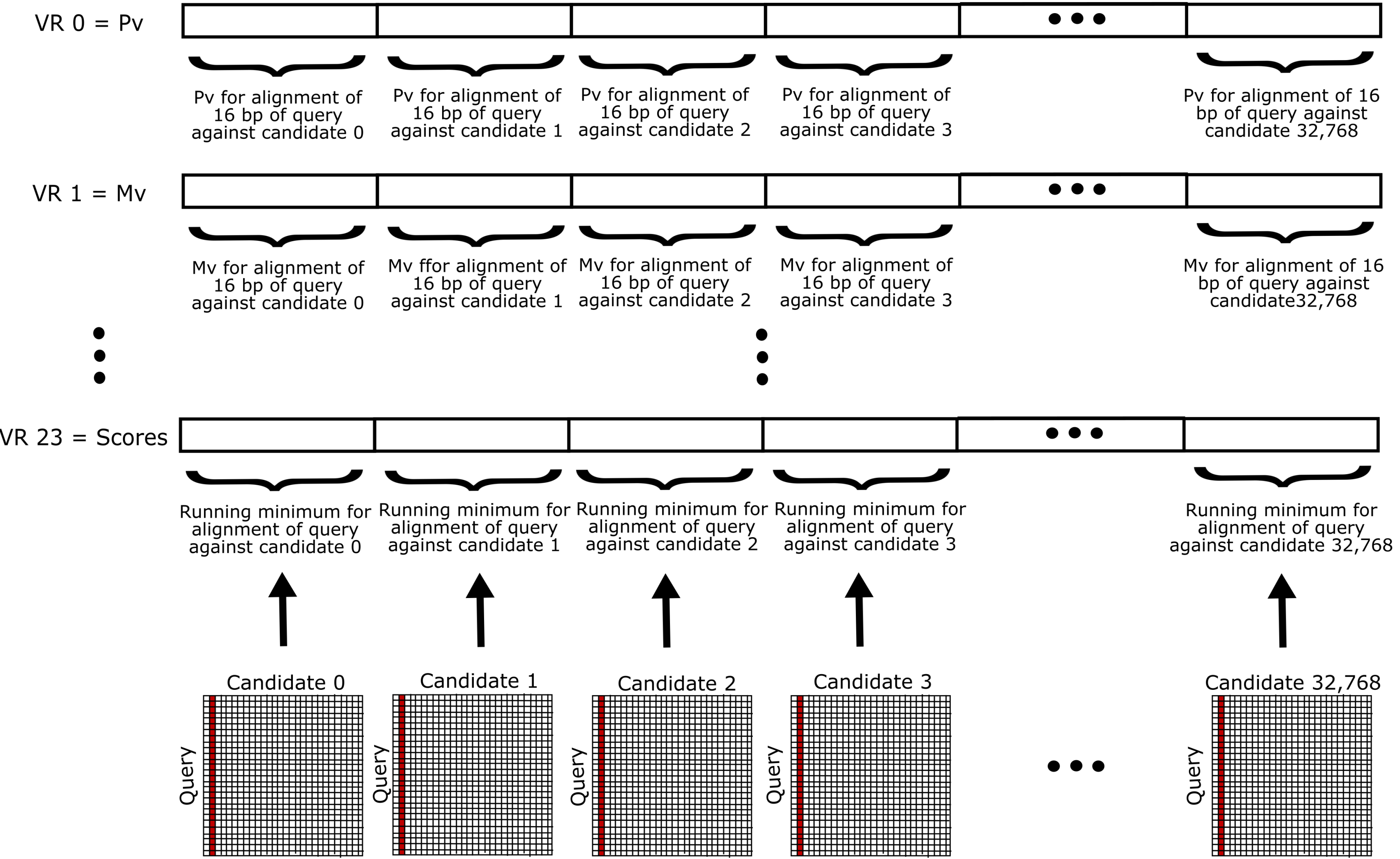
Myers' Algorithm:



Δv_j	Pv	Mv
-1	0	1
0	0	0
+1	1	0

```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11      for j in [0...n-1]:
12          eq = peq[seed(j), i]
13          Xv = eq | Mv
14          Xh = ((eq & Pv) + Pv ^ Pv) | eq
15          Ph = Mv | ~(Xh | Pv)
16          Mh = Pv & Xh
17
18          if last bit of Ph is 1, score += 1
19          if last bit of Mh is 1, score -= 1
20          if score < min_score, min_score = score
21
22          shift Ph
23          save old MSB of Ph
24
25          shift Mh
26          save old MSB of Mh
27
28          Pv = Mh | ~(Xv | Ph)
29          Mv = Ph & Xv
```

DATA LAYOUT ON APU

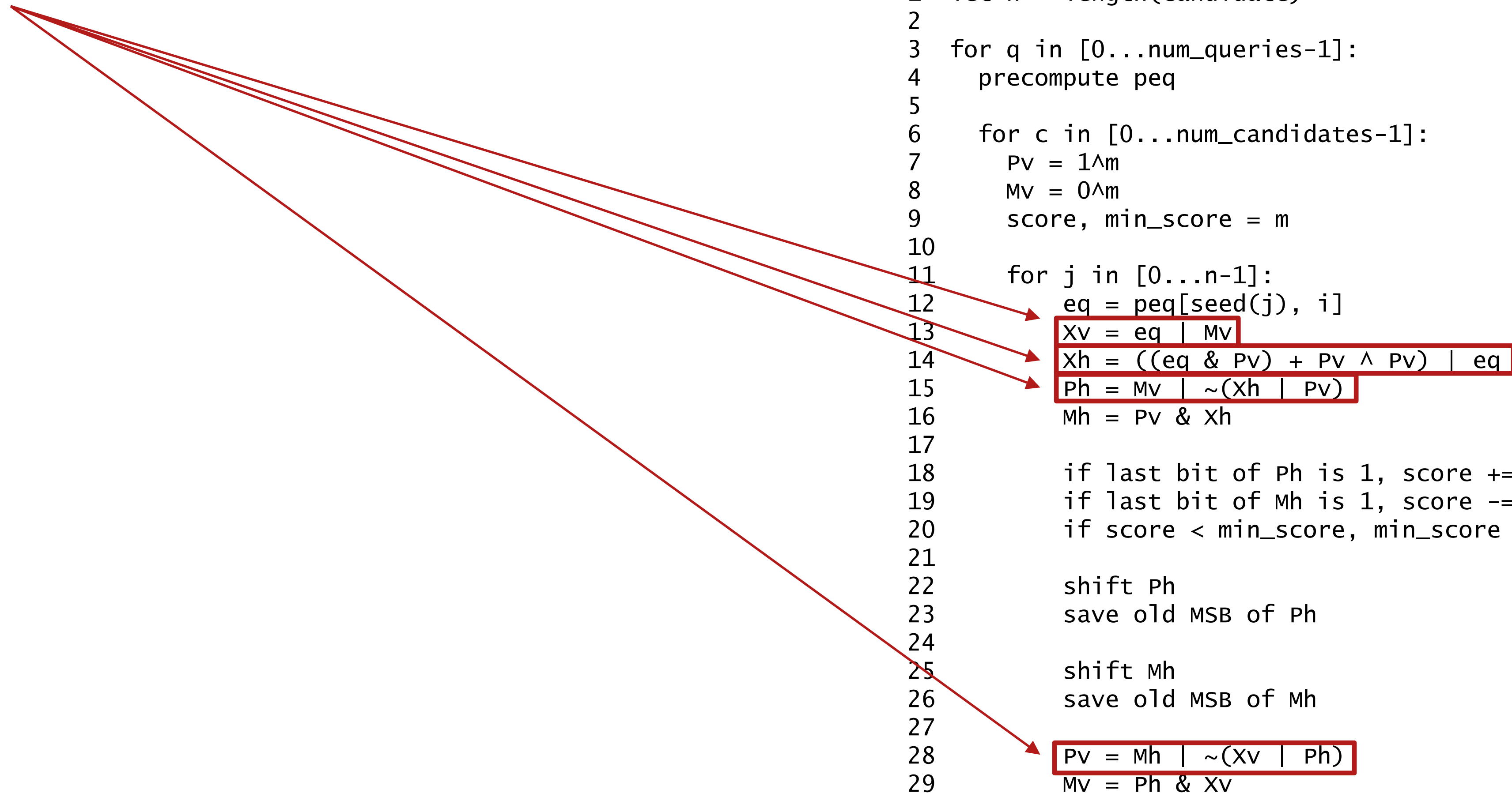



```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11      for j in [0...n-1]:
12          eq = peq[seed(j), i]
13          Xv = eq | Mv
14          Xh = ((eq & Pv) + Pv ^ Pv) | eq
15          Ph = Mv | ~(Xh | Pv)
16          Mh = Pv & Xh
17
18          if last bit of Ph is 1, score += 1
19          if last bit of Mh is 1, score -= 1
20          if score < min_score, min_score = score
21
22          shift Ph
23          save old MSB of Ph
24
25          shift Mh
26          save old MSB of Mh
27
28          Pv = Mh | ~(Xv | Ph)
29          Mv = Ph & Xv
```

MICROCODE OPERATIONS IN MYERS'

CSL

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



MICROCODE OPERATIONS IN MYERS'

CSL

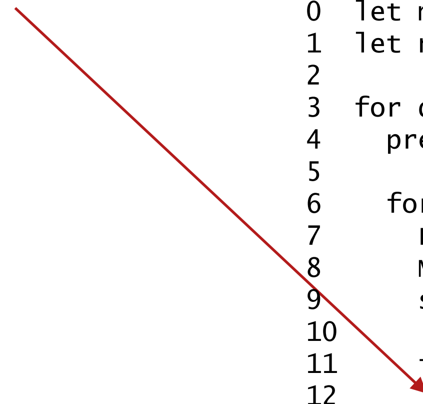
```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
   0xFFFF: RL = VRF[vsrc0];
   0xFFFF: RL |= VRF[vsrc1];
   0xFFFF: VRF[vdst] = RL;

APL_FRAG vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 XOR vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;
  // cout = vsrc0 AND vsrc1
  0x0001:  RL = VRF[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];
  ...
```

```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11      for j in [0...n-1]:
12          eq = peq[seed(j), i]
13          Xv = eq | Mv
14          Xh = ((eq & Pv) + Pv ^ Pv) | eq
15          Ph = Mv | ~(Xh | Pv)
16          Mh = Pv & Xh
17
18          if last bit of Ph is 1, score += 1
19          if last bit of Mh is 1, score -= 1
20          if score < min_score, min_score = score
21
22          shift Ph
23          save old MSB of Ph
24
25          shift Mh
26          save old MSB of Mh
27
28          Pv = Mh | ~(Xv | Ph)
29          Mv = Ph & Xv
```

MICROCODE OPERATIONS IN MYERS'

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

APL_FRAG vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 XOR vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;
  // cout = vsrc0 AND vsrc1
  0x0001:  RL = VRF[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];
  ...
```

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

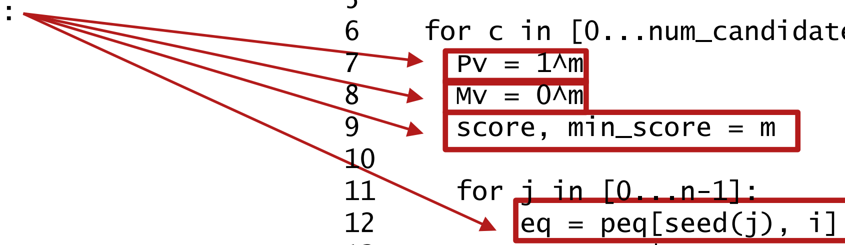


```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11      for j in [0...n-1]:
12        eq = peq[seed(j), i]
13        Xv = eq | Mv
14        Xh = ((eq & Pv) + Pv ^ Pv) | eq
15        Ph = Mv | ~(Xh | Pv)
16        Mh = Pv & Xh
17
18        if last bit of Ph is 1, score += 1
19        if last bit of Mh is 1, score -= 1
20        if score < min_score, min_score = score
21
22        shift Ph
23        save old MSB of Ph
24
25        shift Mh
26        save old MSB of Mh
27
28        Pv = Mh | ~(Xv | Ph)
29        Mv = Ph & Xv
```

MICROCODE OPERATIONS IN MYERS'

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

APL_FRAG vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 XOR vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;
  // cout = vsrc0 AND vsrc1
  0x0001:  RL = VRF[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];
  ...
```

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

APL_FRAG set_scalar(vdst, key):
  0xFFFF: RL = 0;
  key:    RL = 1;
  0xFFFF: VRF[vdst] = RL;
```



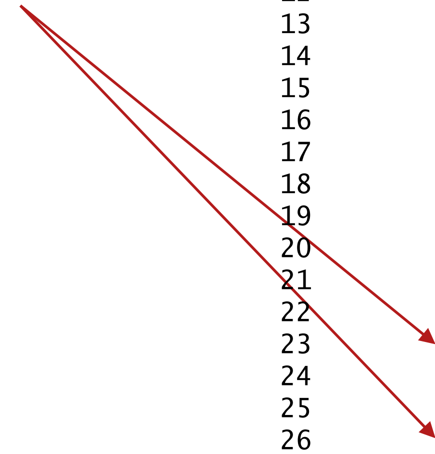
```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11        for j in [0...n-1]:
12          eq = peq[seed(j), i]
13          Xv = eq | Mv
14          Xh = ((eq & Pv) + Pv ^ Pv) | eq
15          Ph = Mv | ~(Xh | Pv)
16          Mh = Pv & Xh
17
18          if last bit of Ph is 1, score += 1
19          if last bit of Mh is 1, score -= 1
20          if score < min_score, min_score = score
21
22          shift Ph
23          save old MSB of Ph
24
25          shift Mh
26          save old MSB of Mh
27
28          Pv = Mh | ~(Xv | Ph)
29          Mv = Ph & Xv
```

MICROCODE OPERATIONS IN MYERS'

CSL

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

APL_FRAG vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 XOR vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;
  // cout = vsrc0 AND vsrc1
  0x0001:  RL = VRF[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];
  ...
```

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

APL_FRAG set_scalar(vdst, key):
  0xFFFF: RL = 0;
  key:    RL = 1;
  0xFFFF: VRF[vdst] = RL;

APL_FRAG save_last_bit(vdst, msk_b16, msk_bit):
  msk_b16: GVL = RL;
  msk_bit: VRF[vdst] = GVL;
```



```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11      for j in [0...n-1]:
12          eq = peq[seed(j), i]
13          Xv = eq | Mv
14          Xh = ((eq & Pv) + Pv ^ Pv) | eq
15          Ph = Mv | ~(Xh | Pv)
16          Mh = Pv & Xh
17
18          if last bit of Ph is 1, score += 1
19          if last bit of Mh is 1, score -= 1
20          if score < min_score, min_score = score
21
22          shift Ph
23          save old MSB of Ph
24
25          shift Mh
26          save old MSB of Mh
27
28          Pv = Mh | ~(Xv | Ph)
29          Mv = Ph & Xv
```

Microcode Operations in Myers'

CSL

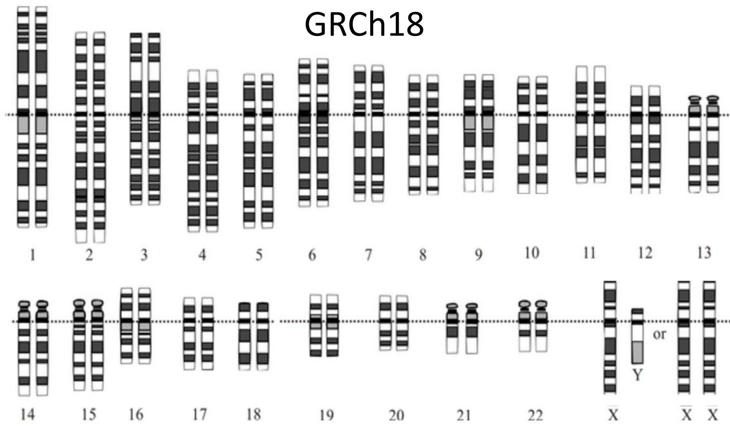
```
APL_FRAG _frag_bitwise_or(vdst, vsrc0, vsrc1):
    0xFFFF: RL = VRF[vsrc0];
    0xFFFF: RL |= VRF[vsrc1];
    0xFFFF: VRF[vdst] = RL;

APL_FRAG vadd(vdst, vsrc0, vsrc1):
  // ---- bit 0 ----
  // vdst = vsrc0 XOR vsrc1
  0x0001:  RL = VRF[vsrc0];
  0x0001:  RL ^= VRF[vsrc1];
  0x0001:  VRF[vdst] = RL;
  // cout = vsrc0 AND vsrc1
  0x0001:  RL = VRF[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];
  ...
```

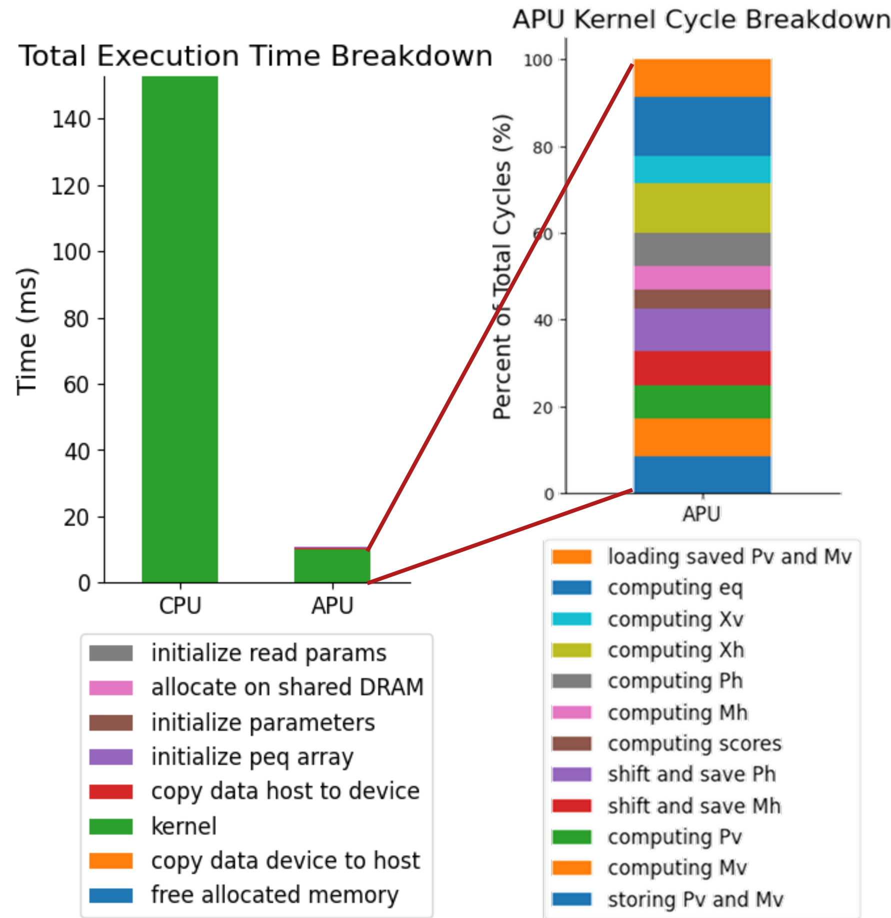
```
APL_FRAG search(vdst, vsrc, key):
  key:        RL = VRF[vsrc];
  ~key:       RL = ~VRF[vsrc];
  0xFFFF:     GVL = RL;
  0xFFFF:     VRF[vdst] = GVL;

APL_FRAG set_scalar(vdst, key):
  0xFFFF: RL = 0;
  key:    RL = 1;
  0xFFFF: VRF[vdst] = RL;

APL_FRAG save_last_bit(vdst, msk_b16, msk_bit):
  msk_b16: GVL = RL;
  msk_bit: VRF[vdst] = GVL;

APL_FRAG lsl_with_cin(vsrc, cin):
  0xFFFF: RL = VRF[vsrc];
  0xFFFF: VRF[vsrc] = RL_N;
  0x0001: RL = VRF[cin];
  0x0001: VRF[vsrc] = RL;

.
.
.
```

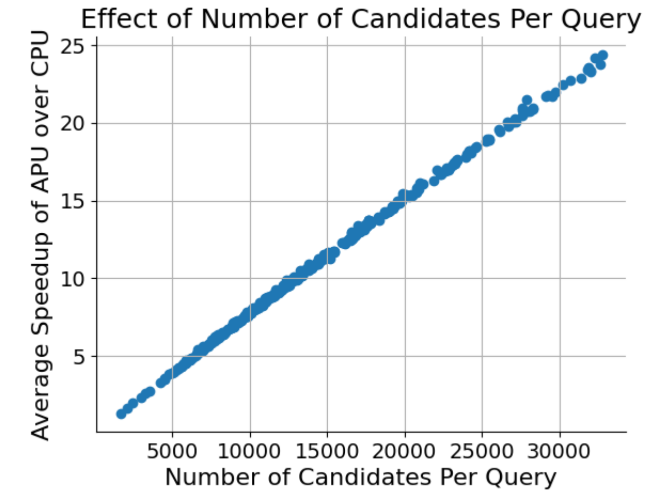
```
0   let m = length(query)
1   let n = length(candidate)
2
3   for q in [0...num_queries-1]:
4     precompute peq
5
6     for c in [0...num_candidates-1]:
7       Pv = 1^m
8       Mv = 0^m
9       score, min_score = m
10
11      for j in [0...n-1]:
12          eq = peq[seed(j), i]
13          Xv = eq | Mv
14          Xh = ((eq & Pv) + Pv ^ Pv) | eq
15          Ph = Mv | ~(Xh | Pv)
16          Mh = Pv & Xh
17
18          if last bit of Ph is 1, score += 1
19          if last bit of Mh is 1, score -= 1
20          if score < min_score, min_score = score
21
22          shift Ph
23          save old MSB of Ph
24
25          shift Mh
26          save old MSB of Mh
27
28          Pv = Mh | ~(Xv | Ph)
29          Mv = Ph & Xv
```

RESULTS

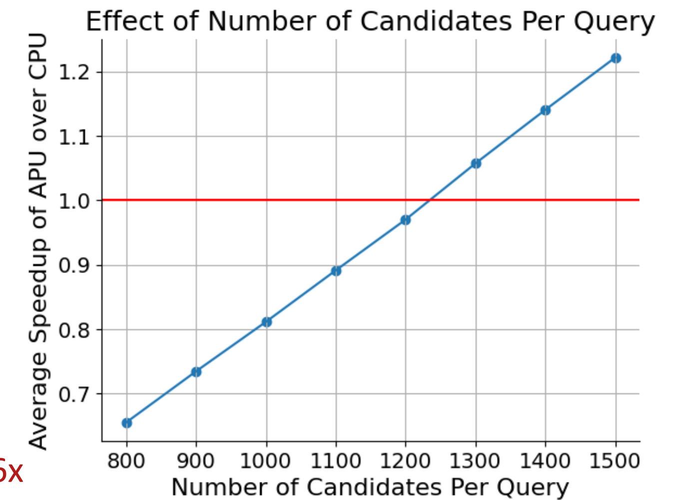
CSL







Best-case Speedup: 24.1x



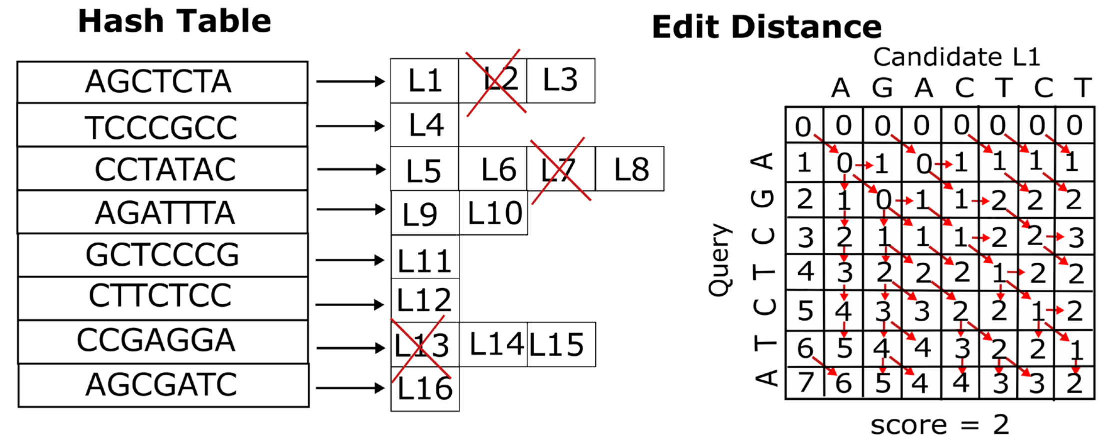
Average Speedup (1-core CPU, 1-core APU): 14.01x

Average Speedup (16-core CPU, 1-core APU): 1.49x

Estimated Average Speedup (16-core CPU, 4-core APU): 5-6x

Accelerating Seed Location Filtering in DNA Read Mapping Using a Commercial Compute-in-SRAM Architecture

1. APU is an interesting platform for applications with massive parallelism, bitwise operations, narrow bitwidths, and high data reuse
2. APU is a promising way to accelerate filtering and other genomics workloads
3. APU's massive parallelism might prompt rethinking of traditional genomics algorithms




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