

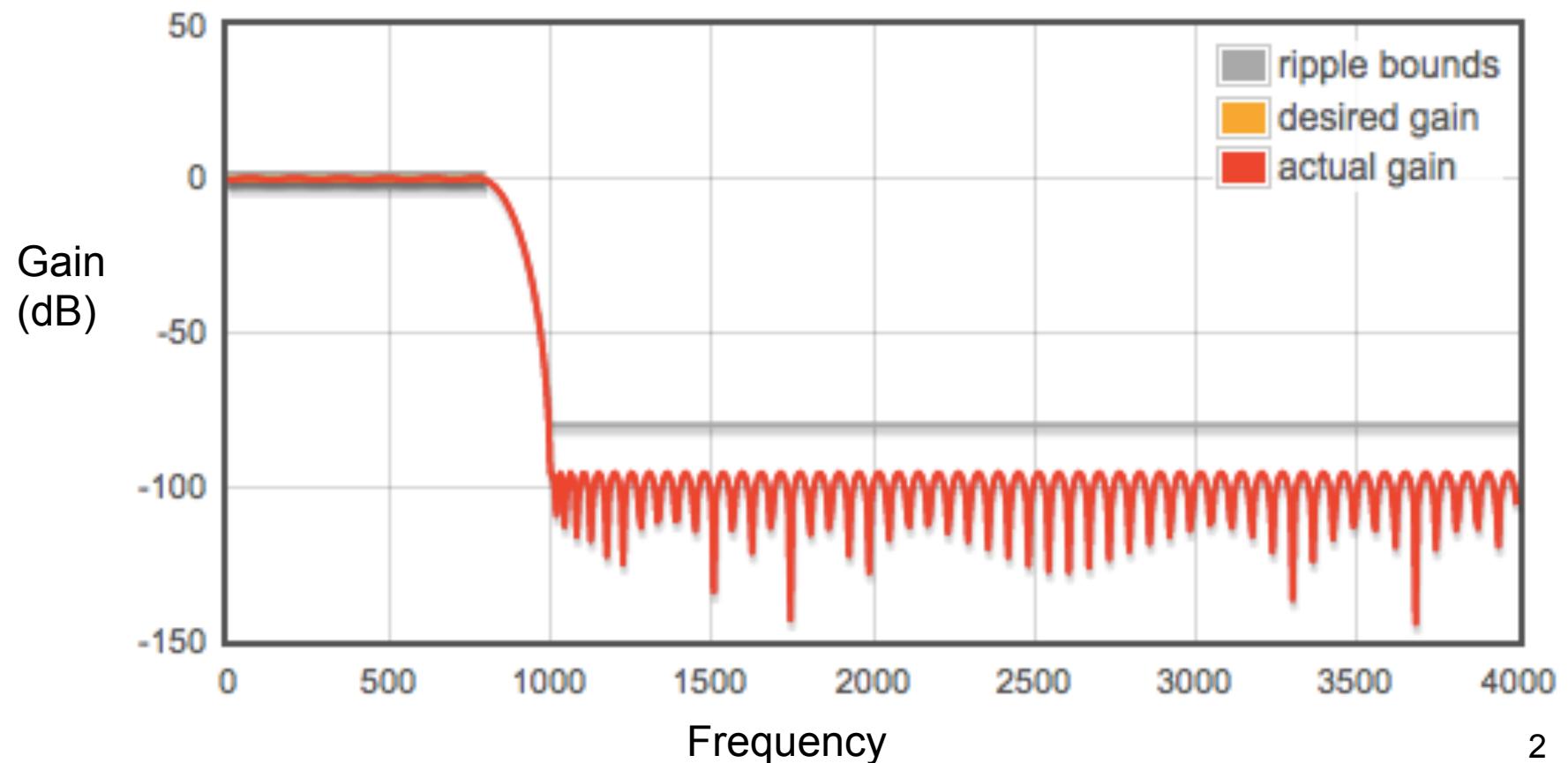
**VectorBlox**  
embedded supercomputing

## Introduction to Programming

ORCA RISC V with the  
VectorBlox MXP™ Matrix Processor

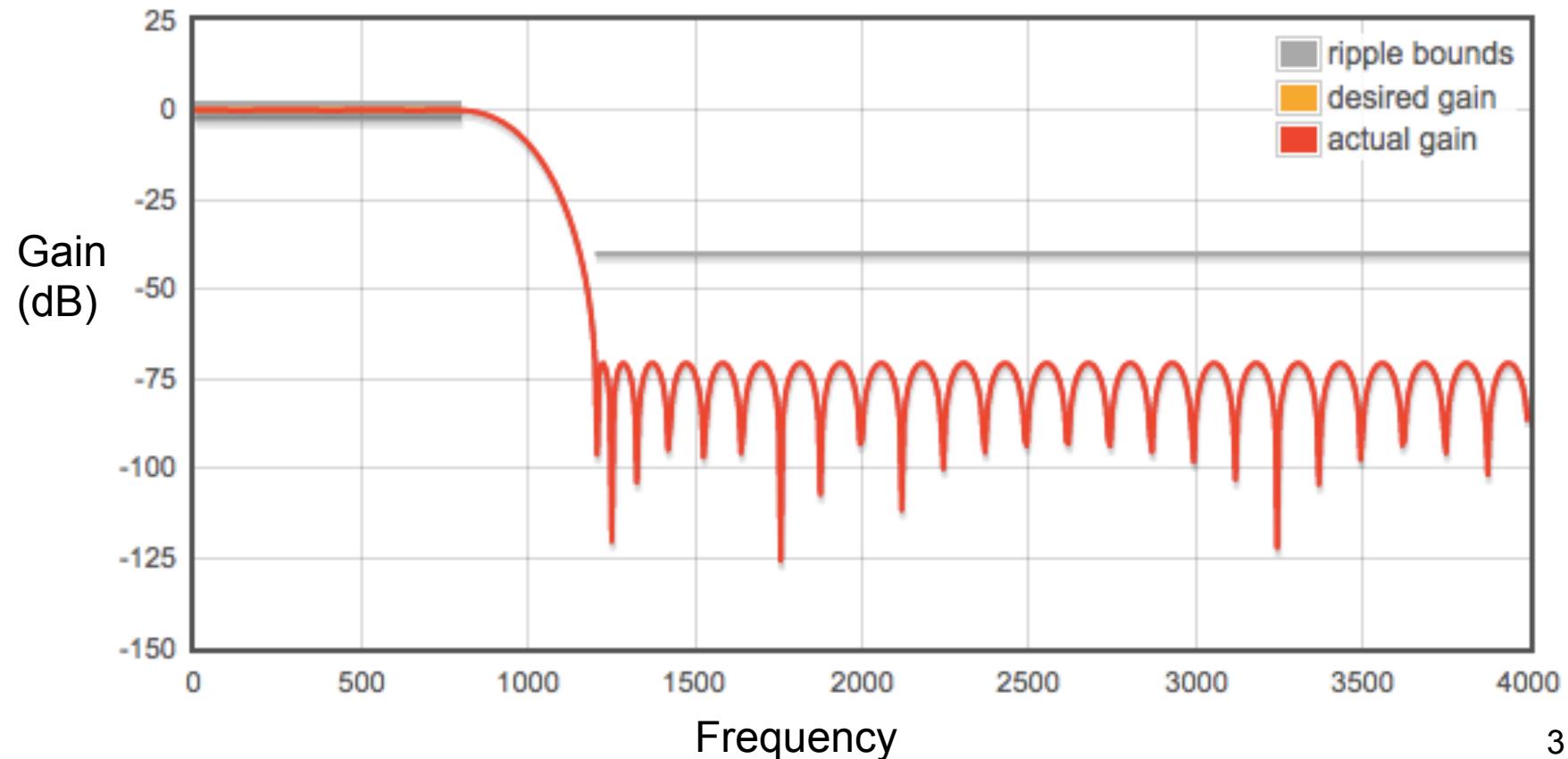
# Demo Application

- Audio FIR filter
  - Low-pass 0 to 800Hz
  - **128 taps**, 16b coefficients



# Demo Application

- Audio FIR filter
  - Low-pass 0 to 800Hz
  - **64 taps**, 16b coefficients



# Audio Filter

- Audio characteristics
  - Sample rate 8000 Hz
  - 16b signed audio data
  - Using mono sound (L+R)
- Approach (infinite loop)
  - RECORD: Read N samples (stalls if FIFO empty)
  - Compute filtered output (128 multiply-accumulates)
  - PLAYBACK: Write N samples (stalls if FIFO full)

# Performance Expectation

- Rates
  - RV32IM running at 20MHz
  - Audio sampled at 8kHz
- Deadline
  - $20M / 8k = 2500$  cycles per sample
- Compute
  - 128 multiply-accumulates
  - RV32IM: ~10 cycles per multiply-accumulate
  - Estimated at 1280 cycles / sample
  - MXP: 8 multiply-accumulates per cycle (0.125 cycles per MAC)
  - Estimated at 16 cycles / sample (speedup ~80x)

# Actual Code

```
static vbx_half_t *v_taps;
static vbx_half_t *v_inp;
static vbx_half_t *v_out;

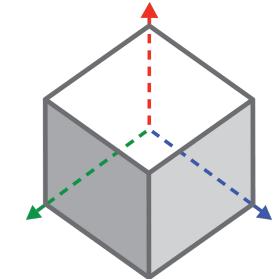
void scalar_filter( ) {
    int i, j, acc;
    for( i=0; i < SND_LEN-NTAPS; i++ ) {
        acc=0;
        for( j=0; j < NTAPS; j++ ) {
            const int sample = (int) v_inp[i+j];
            const int tap     = (int) v_taps[j];
            acc += (sample*tap);
        }
        v_out[i] = (vbx_half_t)(acc >> 16);
    }
}
```

# Scalar Assembly Code

```
// scalar code

708:      00079703      lh      a4 , 0(a5)
70c:      00069583      lh      a1 , 0(a3)
710:      00278793      addi    a5 , a5 , 2
714:      00268693      addi    a3 , a3 , 2
718:      02b70733      mul     a4 , a4 , a1
71c:      00e60633      add     a2 , a2 , a4
720:      fef514e3      bne    a0 , a5 , 708 <scalar_filter+0x28>
```

# Accelerator Solution

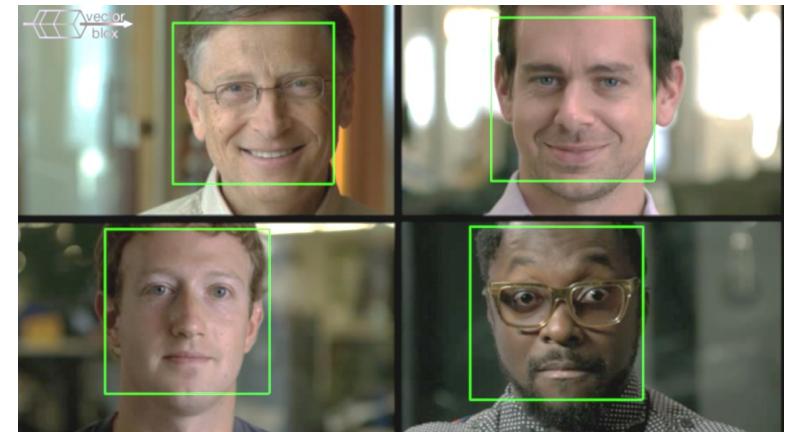


**Host  
Processor**

**C/C++  
Programmable  
Accelerator**

**Algorithm  
Solutions**

VectorBlox      VectorBlox  
**ORCA**      +      **MXP**      =  
 **RISC-V**      Matrix Processor



- Instant compile & run
- Massively improved performance

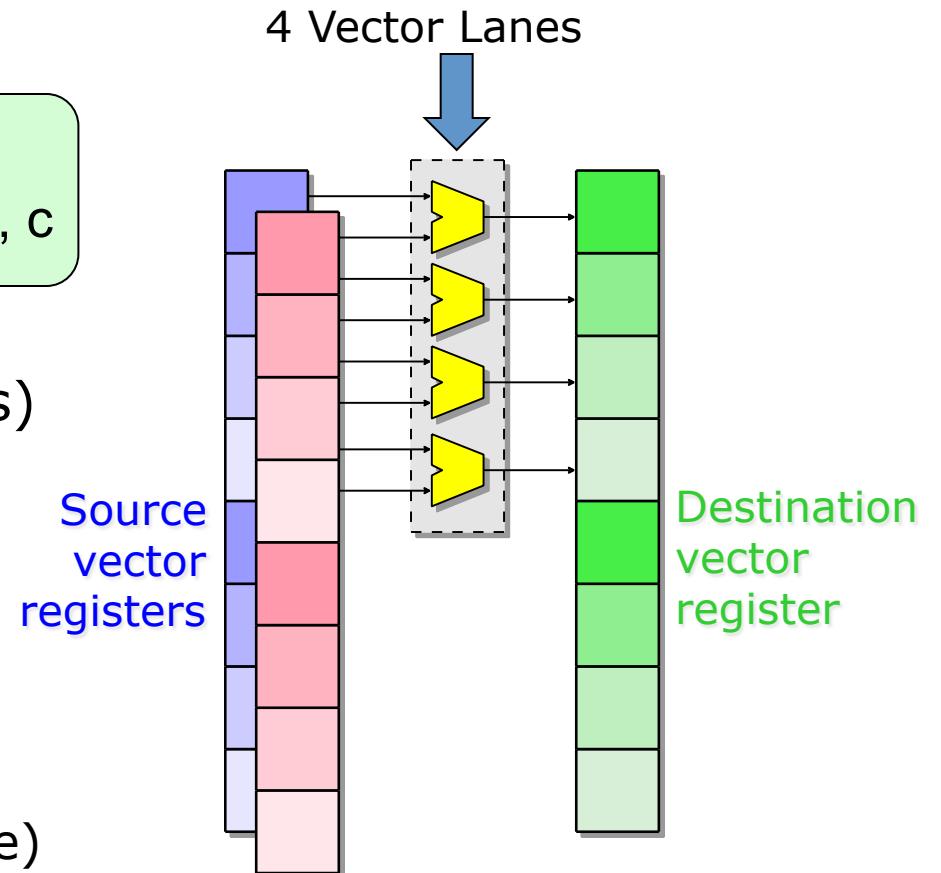
# MXP Core Technology



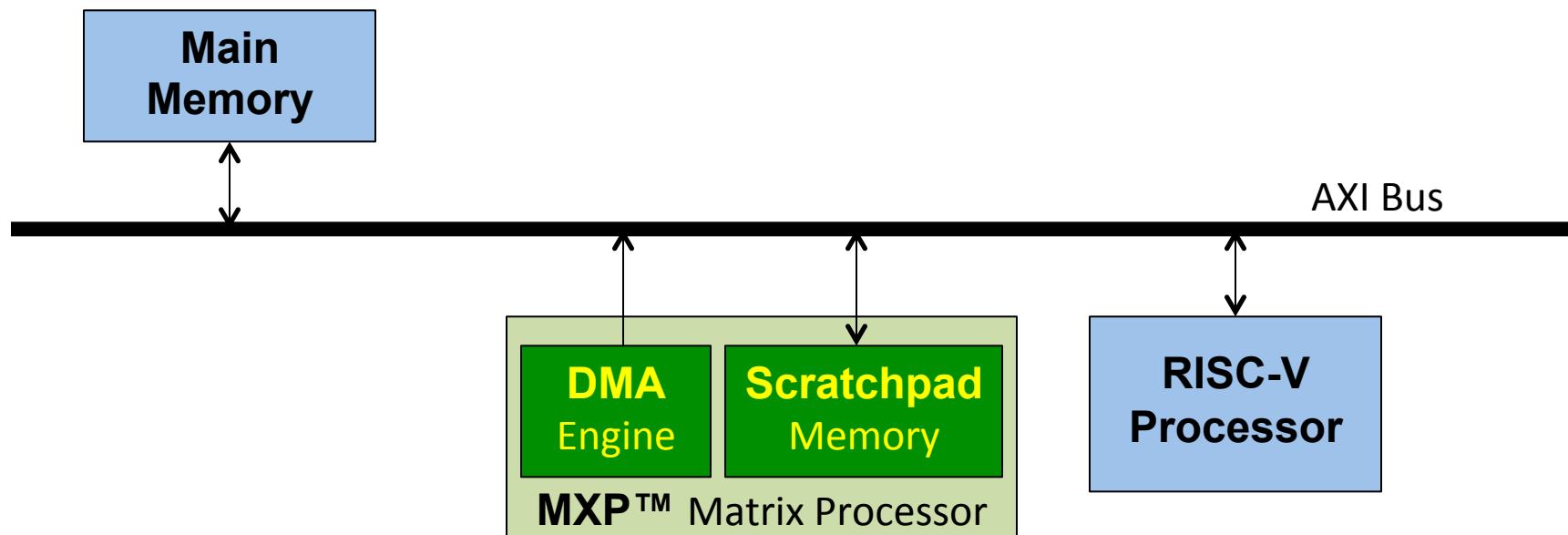
## Data-level parallelism

```
for ( i=0; i<8; i++ )      set_vl, 8  
    a[i] = b[i] * c[i];       vmult a, b, c
```

- 1D, 2D, 3D vectors (matrices)
- Default data types
  - Byte, halfword, word
  - Integer, fixed-point
- Optional data types
  - Floating-point (single, double)

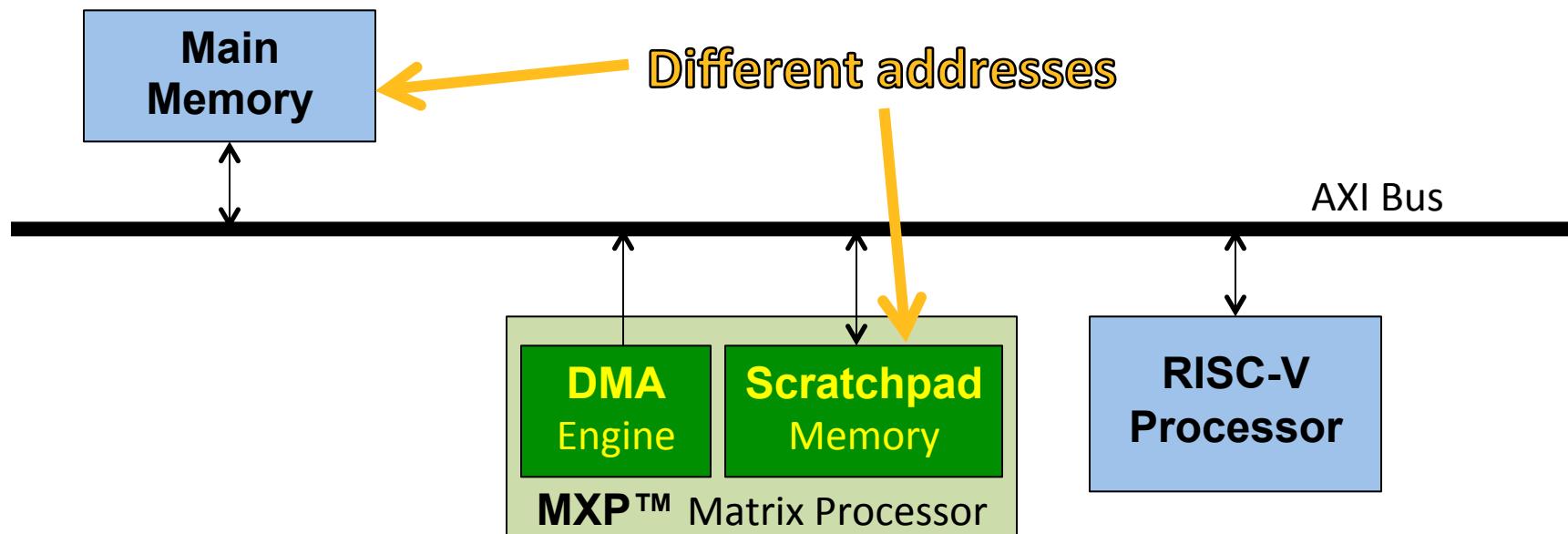


# System Model



- Host processor runs C/C++ code
- Lightweight/inline function calls to VectorBlox library

# System Model



- Vector instructions operate only on scratchpad
- Allocate a vector of 8 words

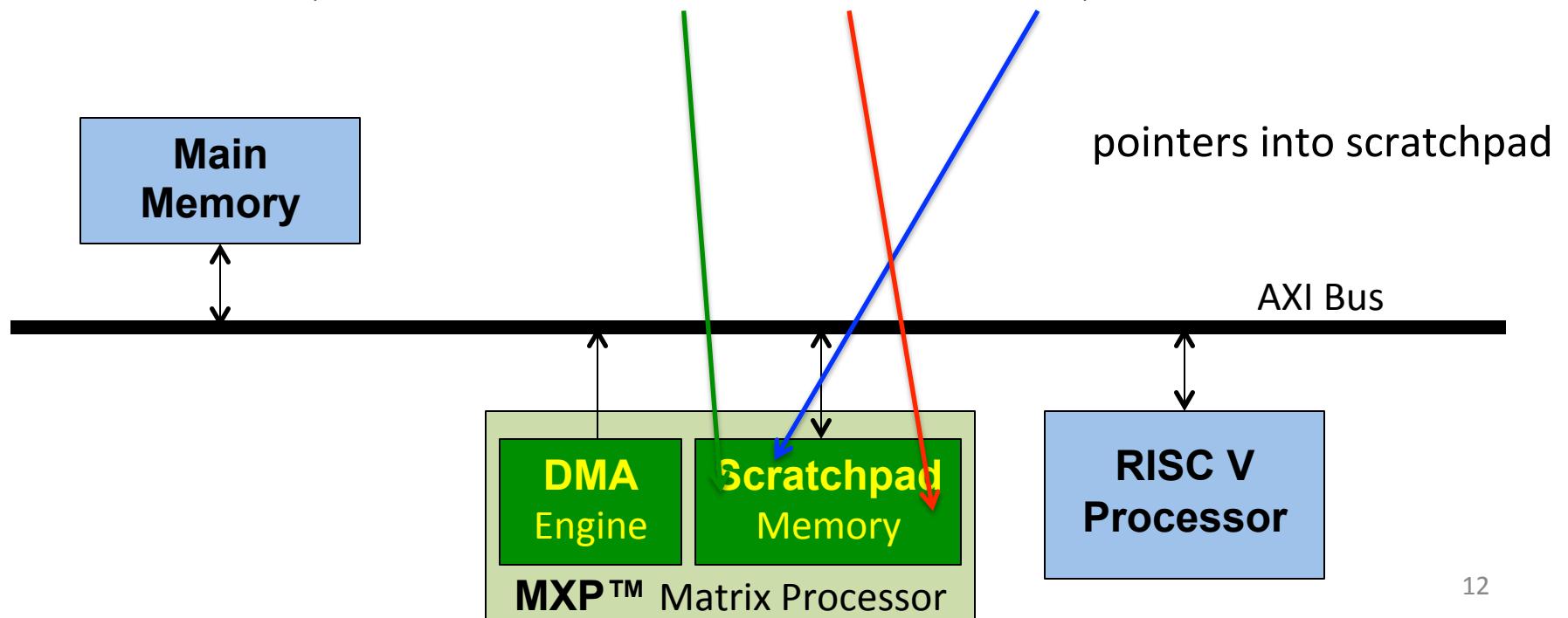
```
vbx_word_t *vsrc1 = vbx_sp_malloc( 32 );
```

# API



## Intrinsics

```
vbx_word_t *vsrc1 = vbx_sp_malloc( 32 );  
vbx_set_vl( 8 );  
vbx( VVW, VADD, vdst, vsrc1, vsrc2 ); // C, or  
vbxx( VADD, vdst, vsrc1, vsrc2 ); // C++
```

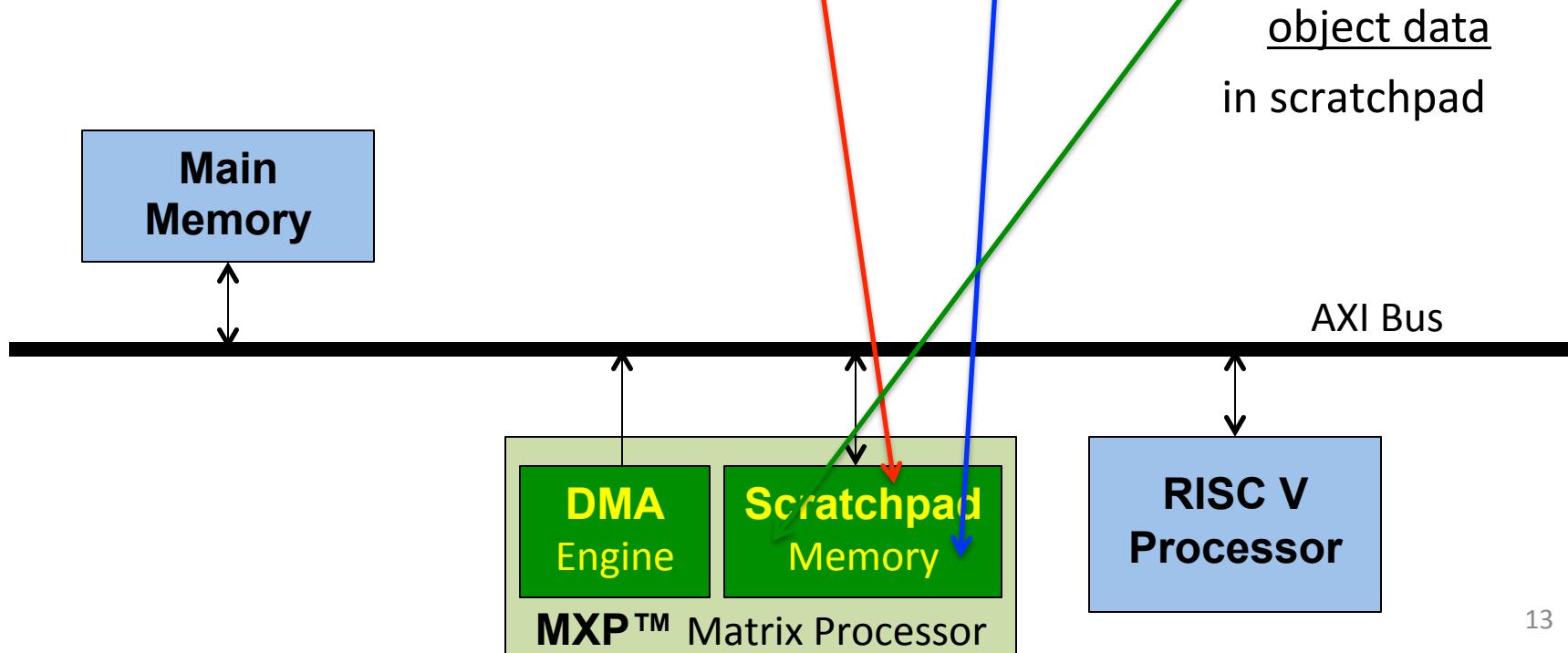


# Execution and Memory Model



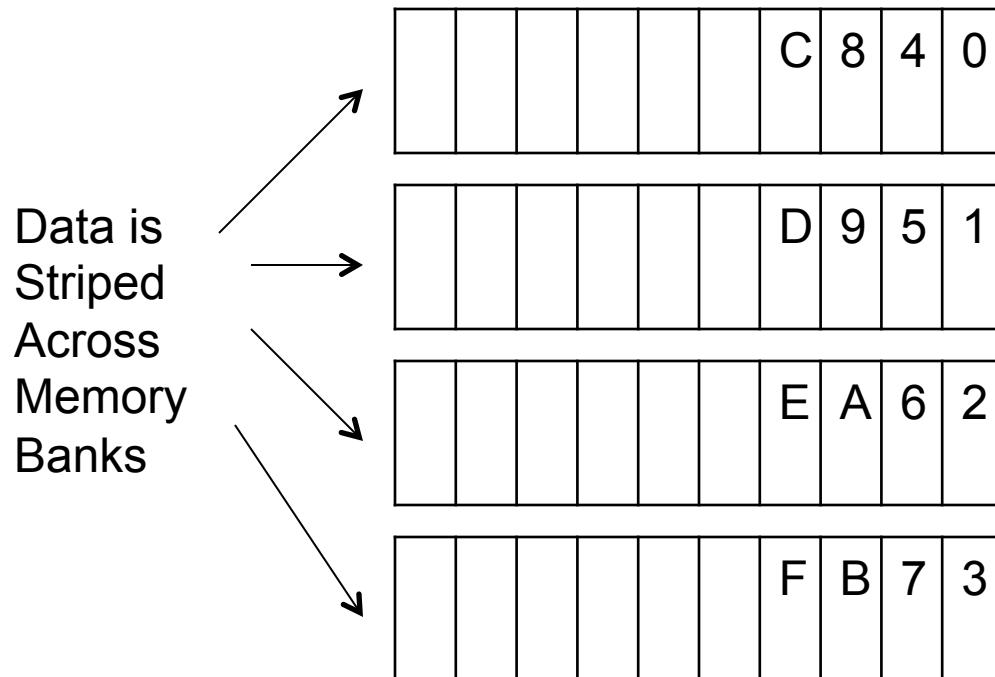
## C++ Objects

```
Vector<vbx_word_t> vsrc1(8), vsrc2(8), vdst(8);  
vdst = vsrc1 + vsrc2;
```



# Scratchpad Memory

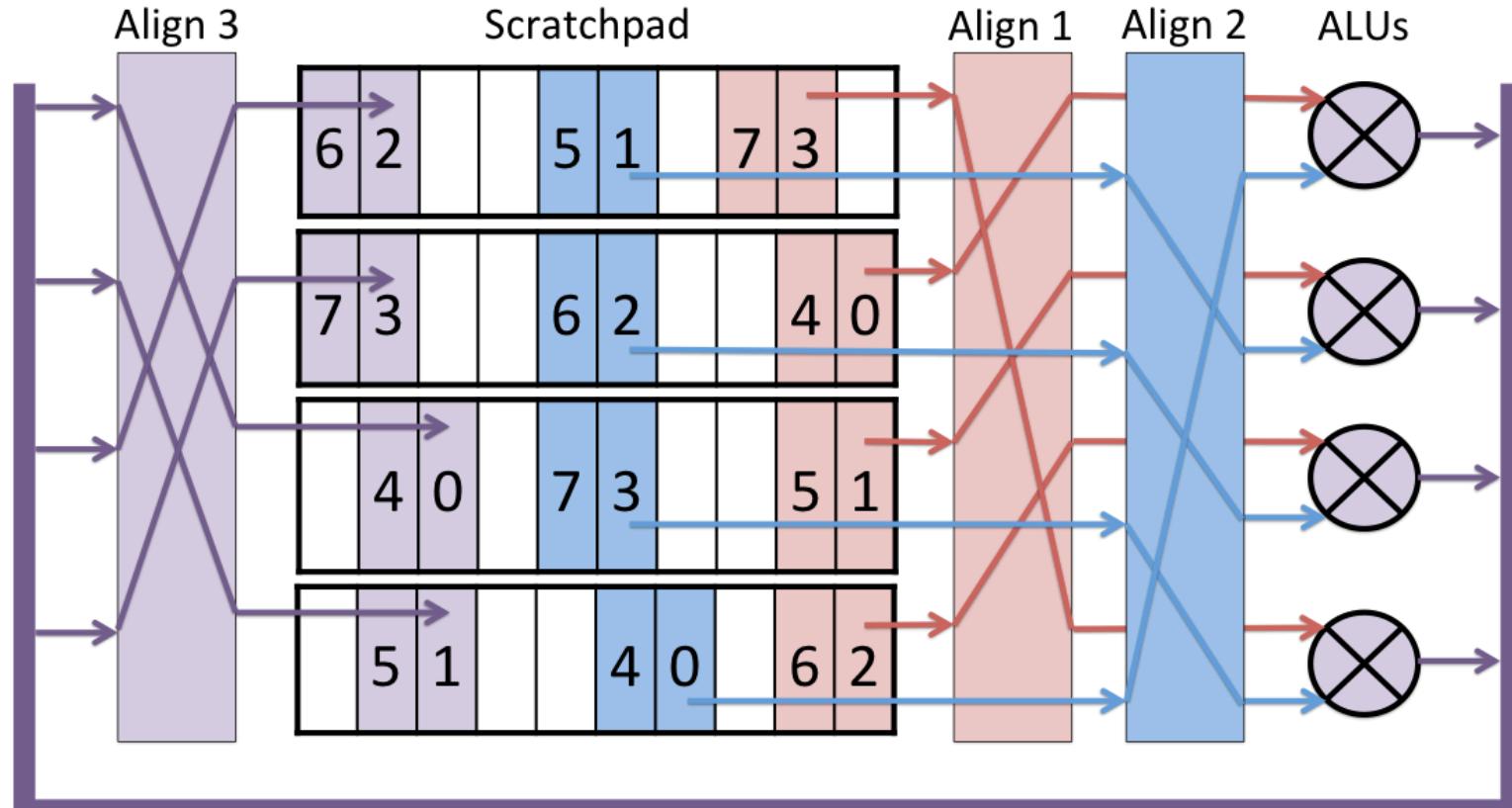
- Multi-banked, parallel access
  - Addresses striped across banks, like RAID disks
  - `vbx_sp_malloc(12)` allocates 3 words in Scratchpad



# Scratchpad-based Computing

```
vbx_word_t *vdst, *vsrc1, *vsrc2;
```

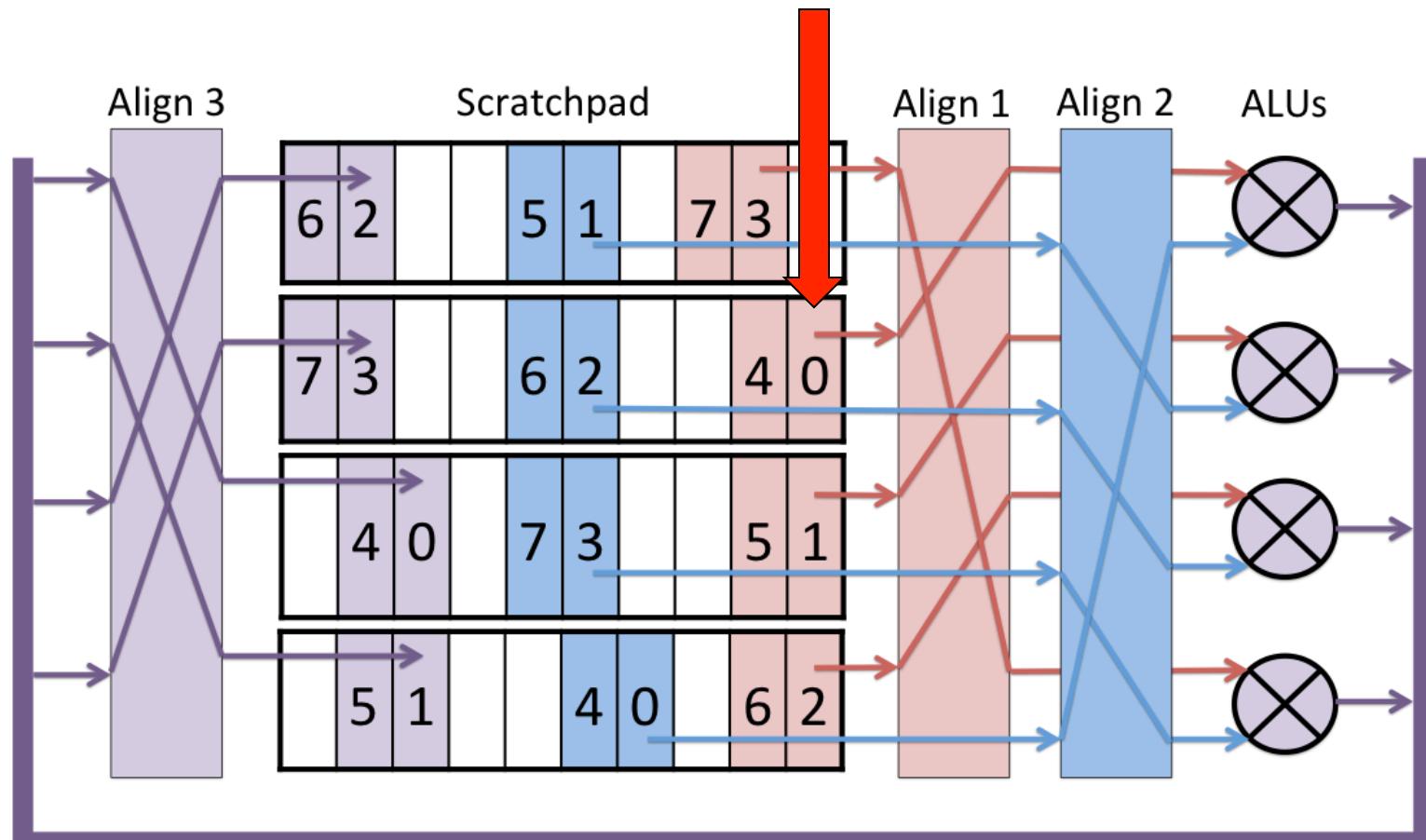
```
vbx( VVW, VADD, vdst, vsrc1, vsrc2 );
```



# Scratchpad-based Computing

```
vbx_word_t *vdst, *vsrc1, *vsrc2;
```

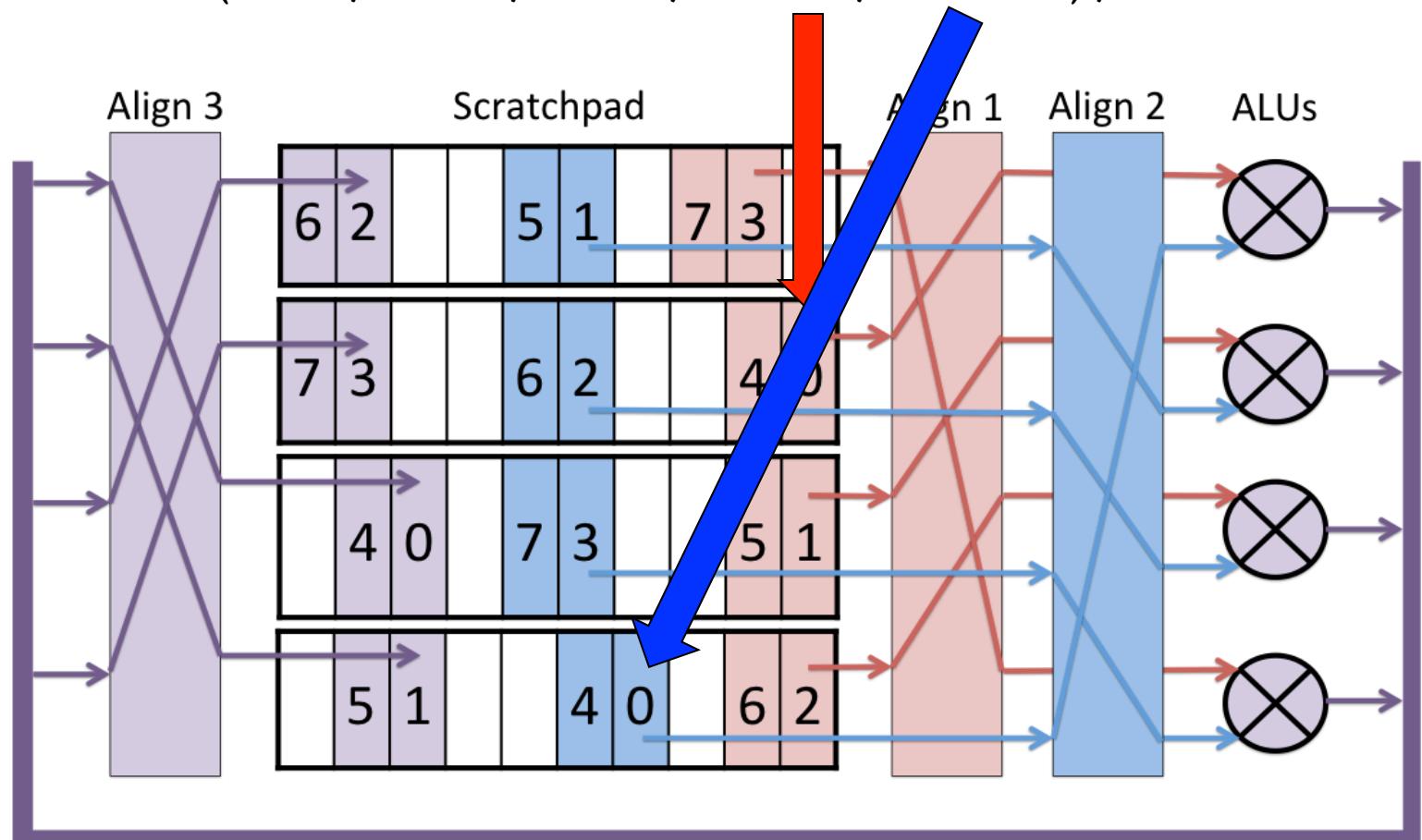
```
vbx( VVW, VADD, vdst, vsrc1, vsrc2 );
```



# Scratchpad-based Computing

```
vbx_word_t *vdst, *vsrc1, *vsrc2;
```

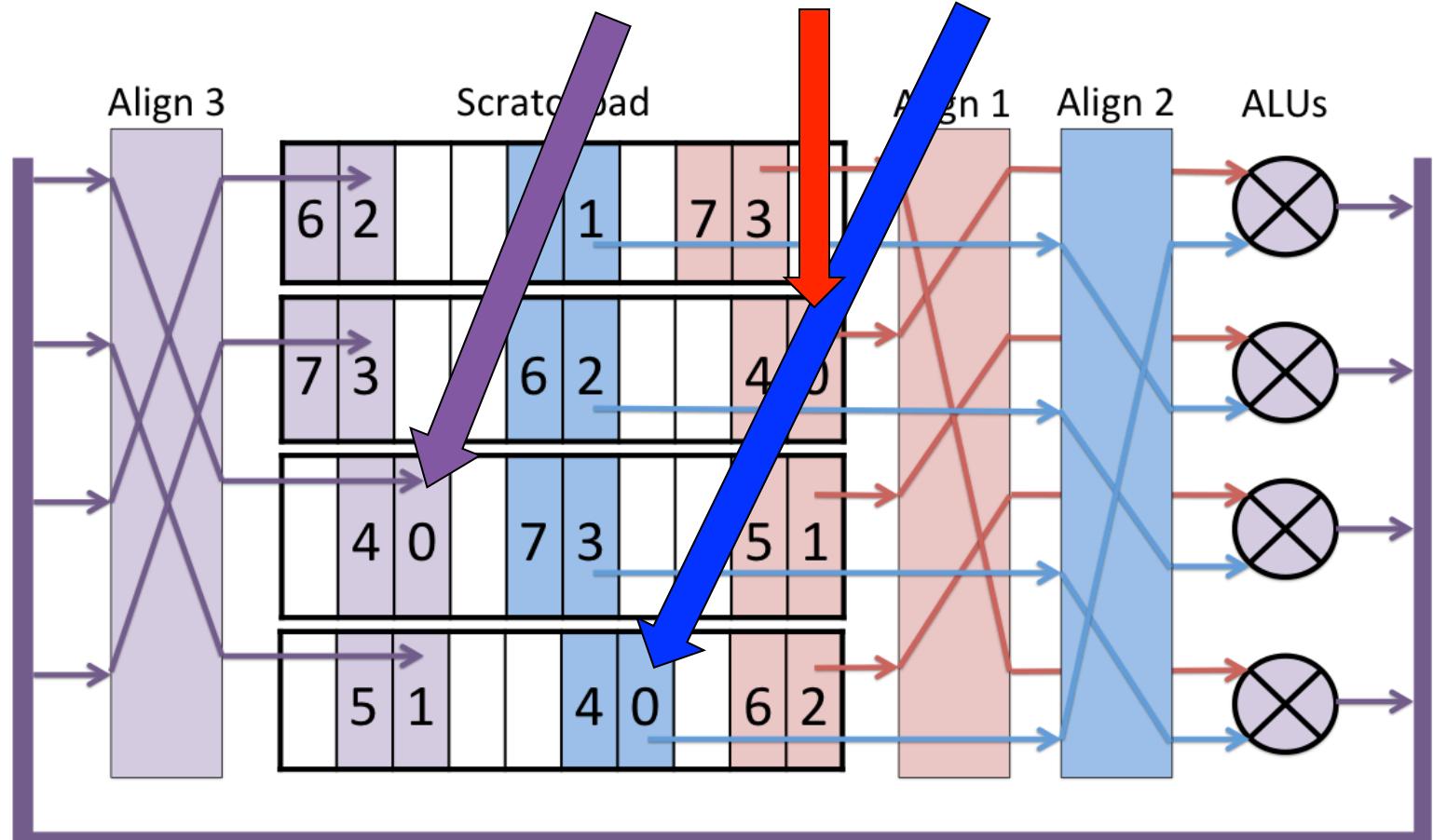
```
vbx( VVW, VADD, vdst, vsrc1, vsrc2 );
```



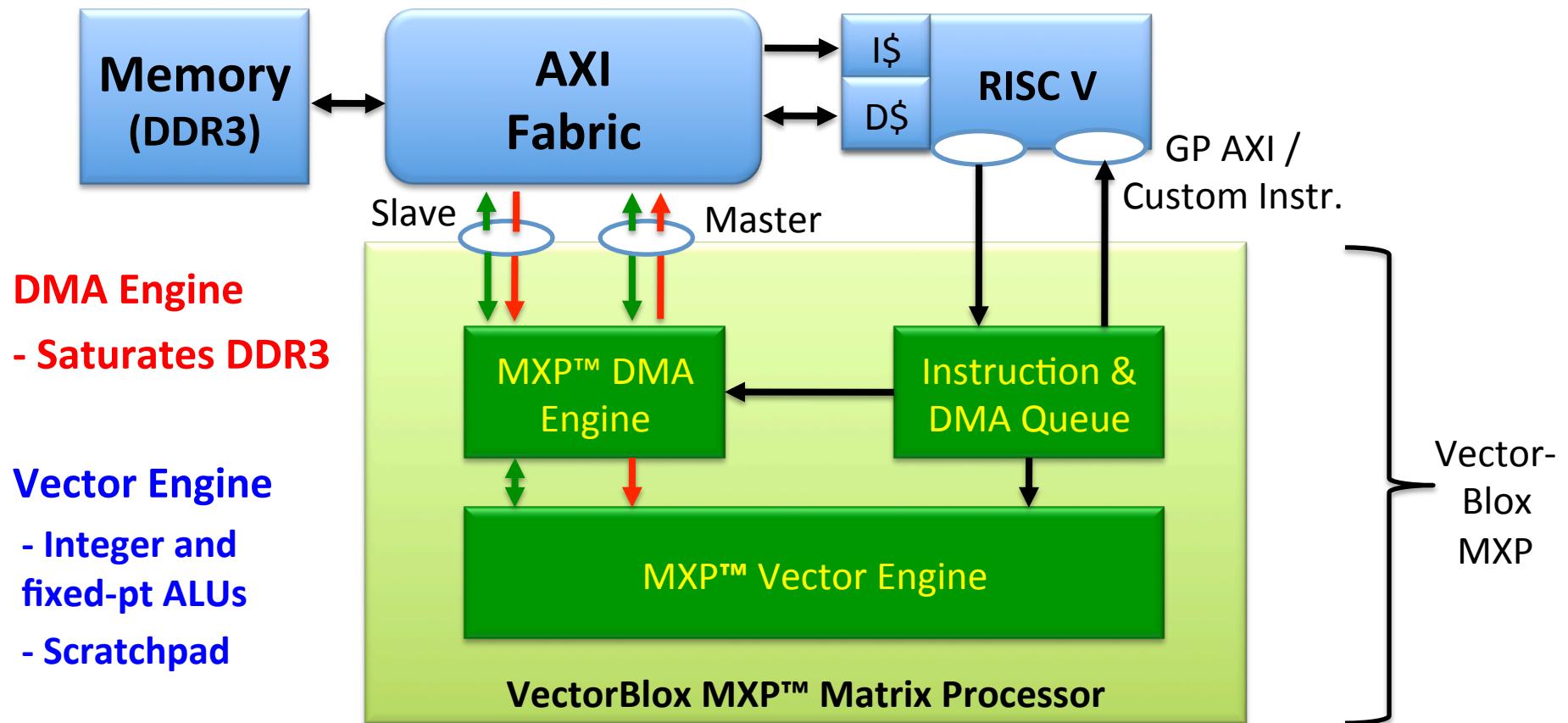
# Scratchpad-based Computing

```
vbx_word_t *vdst, *vsrcl, *vsrc2;
```

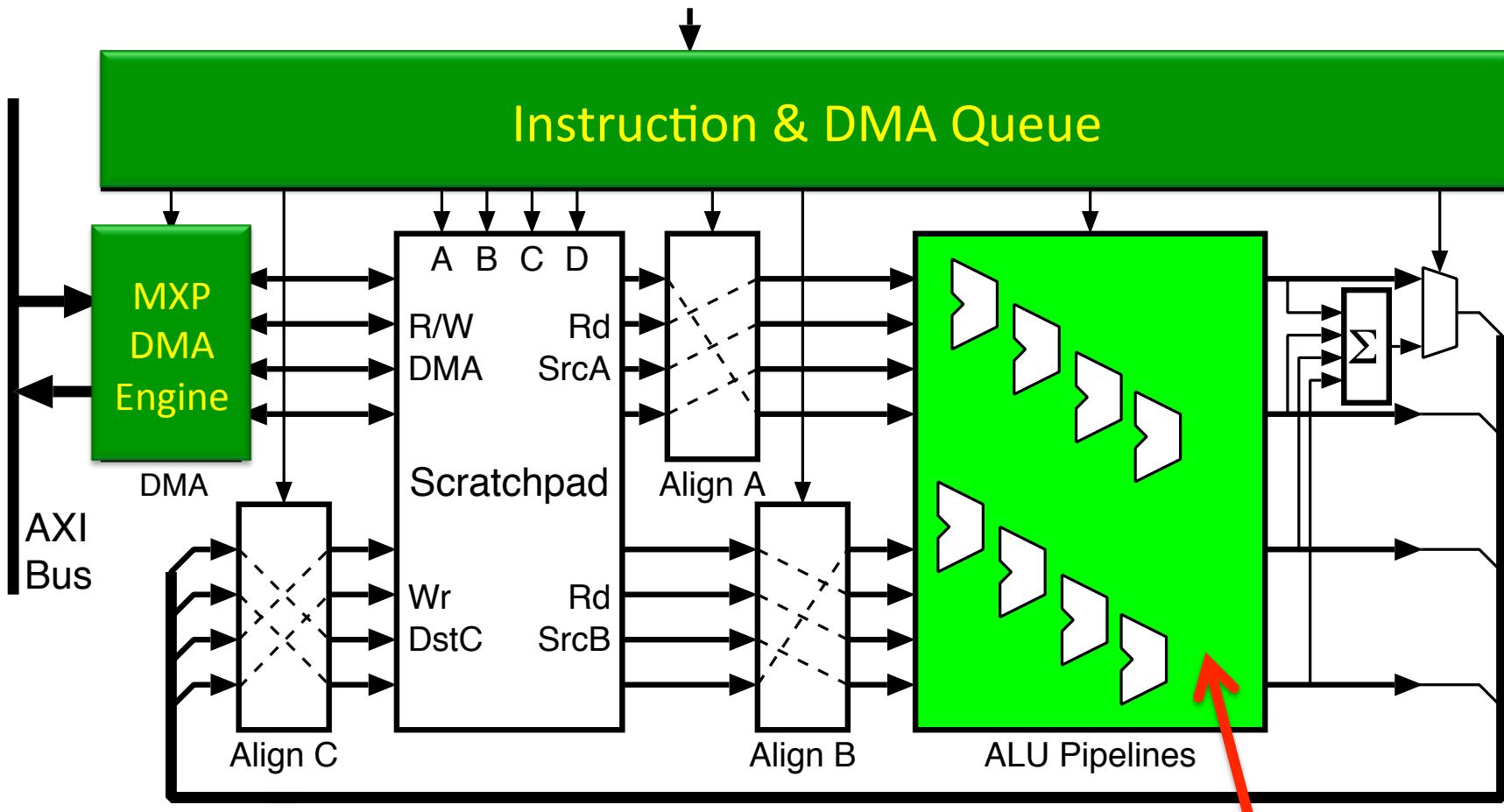
```
vbx( VVW, VADD, vdst, vsrcl, vsrc2 );
```



# Inside the MXP (1)

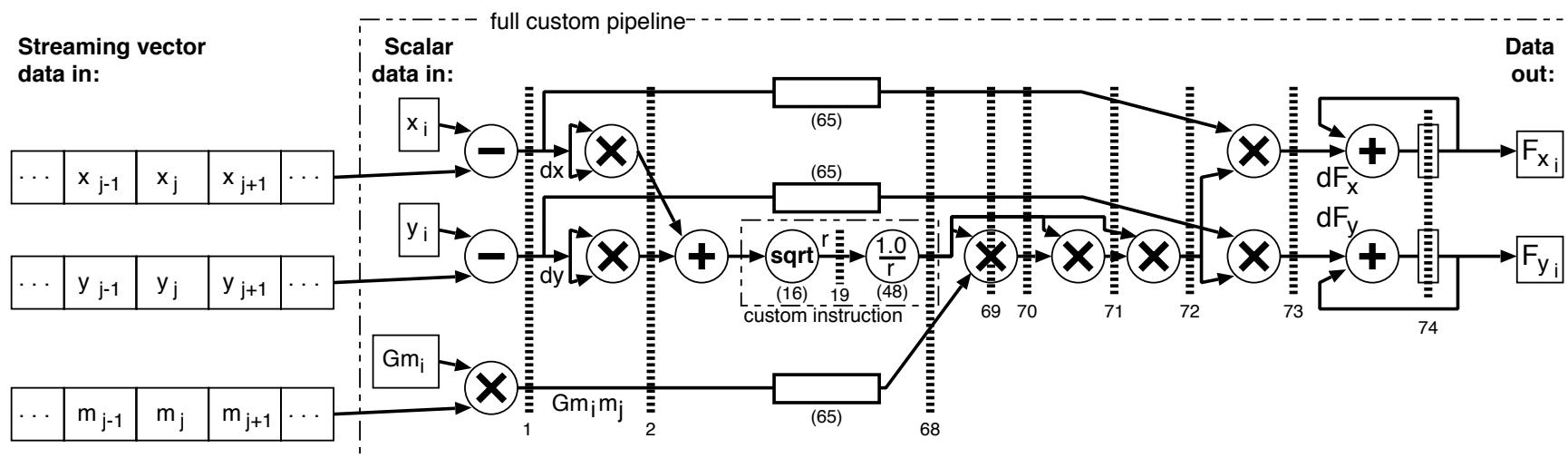


# Inside the MXP (2)



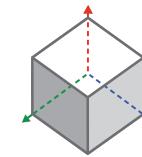
User Adds (Deeply Pipelined) Custom Instructions,  
e.g. generated using High-Level Synthesis <sup>20</sup>

# Custom Instruction Example: Deeply Pipelined Computation

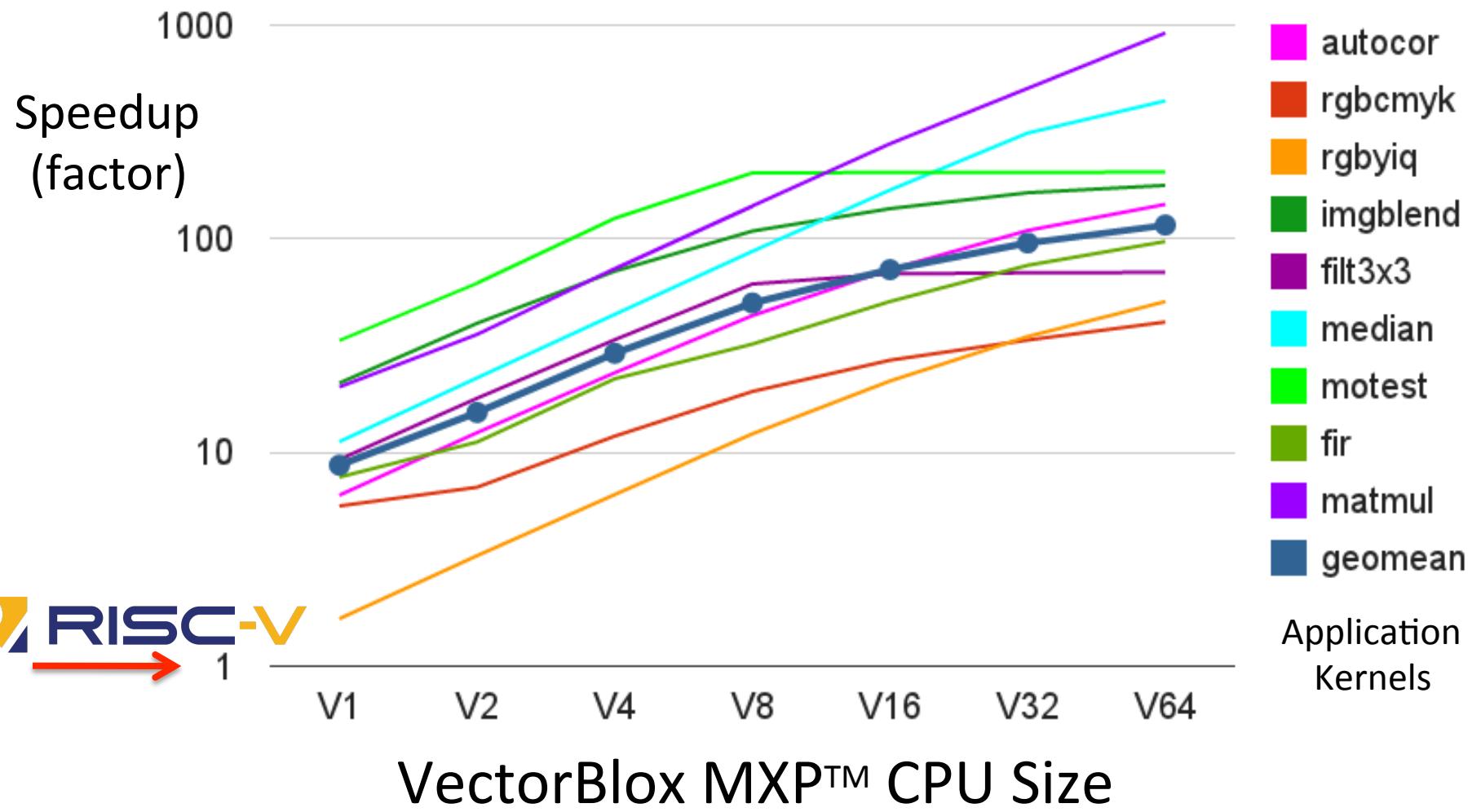


15 Operators  
75 Clock Cycles Deep  
~ 10,000x speedup over Soft Processors

# MicroBenchmarks



**VectorBlox**  
embedded supercomputing



# How Does MXP Get 10x Speedup with just 1 ALU?



## Example: FIR filter (double-nested loop)

### 1. RISC V code

```
4c:    lw      a3 ,0(a1)
50:    lw      a6 ,0(a2)
54:    addi   a5 ,a2 ,4
58:    mv      a7 ,a1
5c:    mul    a6 ,a3 ,a6
60:    sw      a6 ,0(a0)
64:    ble    a4 ,t5 ,0x88
68:    lw      a3 ,4(a7)
6c:    lw      t1 ,0(a5)
70:    addi   a5 ,a5 ,4
74:    addi   a7 ,a7 ,4
78:    mul    a3 ,a3 ,t1
7c:    add    a6 ,a6 ,a3
80:    sw      a6 ,0(a0)
84:    bne    t3 ,a5 ,0x68
88:    addi   a0 ,a0 ,4
8c:    addi   a1 ,a1 ,4
90:    bne    t4 ,a0 ,00x4c
```

### 2. RISC V + MXP code (idealized)

```
10:    slli   a3 ,a3 ,0x2
14:    add    a3 ,a1 ,a3
18:    vset2d0a a6 ,a7
1c:    vset2d1a a6 ,a5
20:    v1len  r4
24:    v1src.www a1 ,a2
28:    vmul.vv.1d.sss.acc a0 ,a4
```

#### Advantage:

- Double-nested loop is 1 instruction

Nested-loop set-up code

#### Overhead:

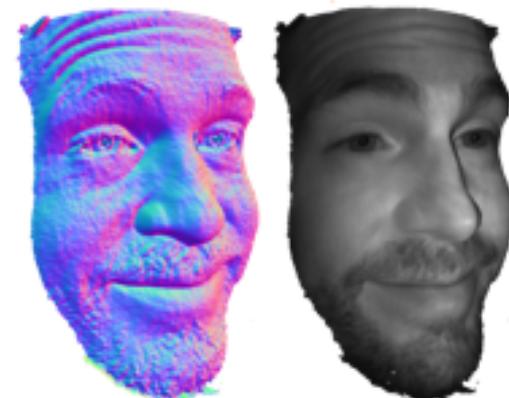
- 2 lw, 1 sw instructions (+ cache miss delays)
- 2 addi instructions (addr generation)
- Mul+add (could be 1 instruction)
- Branch (+ any misprediction penalty)

# Customer Design Example



## 3D Depth Capture

- Original
  - i7-4860HQ + GTX 980M GPU
  - < 180fps @ 640x480
- VectorBlox MXP (PCIe card)
  - 600fps @ 640x480





# Programming the MXP

# Programming APIs

- Three programming layers
  - Low-level C
  - Low-level C++ (inferred types)
  - High-level C++ (objects)
- Layers can be mixed together
- Simple rules to follow

# Programming Interface

# Scratchpad Rules

- Scratchpad is a precious resource
  - Stores all vector data to be operated on
  - DMA data in and out
    - vbx\_dma\_to\_vector( sp\_dest, mm\_src, num\_bytes);
    - vbx\_dma\_to\_host( mm\_dest, sp\_src, num\_bytes);
  - DMAs run in program order,  
but asynchronously in background
    - Vector engine automagically stalls on use
  - **Limitation: no DDR or DMA Engine here**

# Scratchpad Data Types

- Native integer data types

(signed)

`vbx_word_t`

(unsigned)

`vbx_uword_t`

// 32 bits

`vbx_half_t`

`vbx_uhalf_t`

// 16 bits

`vbx_byte_t`

`vbx_ubyte_t`

// 8 bits

# Scratchpad Notes

- Scratchpad “best practices”
  - Allocate using `vbx_sp_malloc()`
  - Use `vbx_sp_free()` to deallocate entire scratchpad
  - In library code, allocate/deallocate like a stack, not heap
    - Use `vbx_sp_push()` saves current allocation level
    - Use `vbx_sp_pop()` restores to last saved allocation level
    - Want to maintain speed, avoid internal fragmentation
    - **Allows COMPOSITION of vector library code**

# Basic C Vector Instructions

- All vector instructions use same format

```
vbx( mode, instr, dest, srcA, srcB );
```

- Parameters

– mode	provides type, data size information
– instr	the instruction opcode
– dest	pointer to vector in scratchpad
– srcA	pointer to vector in scratchpad, or scalar value
– srcB	pointer to vector in scratchpad, or ignored

- Minor restriction

- mode and instr must be compile-time constants

# Basic Vector Instructions

- All vector instructions use same format

```
vbx( mode, instr, dest, srcA, srcB );
```

- **mode** specifier

- Compile-time constant
- 3 to 5 letter symbol; each letter has meaning
- Broken down into three fields: *wxy*

*w*      is one of { **VV**, **SV**, **VE**, **SE** }

*x*      is one of { **B**, **H**, **W**, **BH**, **BW**, **HB**, **HW**, **WB**, **WH** }

*y*      is one of { **S**, **U** }, defaults to { **S** } if unspecified

# Basic C++ Vector Instructions

- All vector instructions use same format

```
vbxx( mode, instr, dest, srcA, srcB );
```

- **mode** specifier is inferred
  - Based on data types of dest, srcA, srcB

# Basic Vector Instructions

- All vector instructions use same format

```
vbx( mode, instr, dest, srcA, srcB );
```

- **instr** specifier
  - Compile-time constant
  - Bitwise: VAND, VOR, VXOR, VSHL, VSHR, VRROT, VRROTR
  - Arithmetic: VADD, VSUB, VADDC, VSUBB, VABSDIFF, VMUL, VMULLO, VMULHI, VMULFXP
  - Movement: VMOV
  - Conditional move: VCMV\_LEZ, VCMV\_GEZ, VCMV\_NZ, VCMV\_FC  
VCMV\_LTZ, VCMV\_GTZ, VCMV\_Z, VCMV\_FS

# Reductions (eg. Dot Products)

- All vector instructions use same format

```
vbx_acc( mode, op, dest, srcA, srcB );
```

- The vbx\_acc() does sum reduction

```
dest(0) = sum_reduce( srcA(i) op srcB(i) );
```

- 2D: repeats reduction, producing 1D vector
- 3D: repeats 2D, producing 2D matrix
  - Reduction done in the innermost loop

# Overall Software Process

1. Allocate vectors in scratchpad
2. ~~Move data from memory → scratchpad~~
3. Set vector length register
4. Perform vector operations
5. ~~Move data from scratchpad → memory~~

# Example: FIR

```
static vbx_half_t *v_taps;
static vbx_half_t *v_inp;
static vbx_half_t *v_out;
static vbx_word_t *v_mac;
```

- **Allocate vectors in scratchpad**

```
void filter_init() {
    v_inp = vbx_sp_malloc( SND_LEN * sizeof(*v_inp) );
    v_out = vbx_sp_malloc( SND_LEN * sizeof(*v_out) );
    v_taps = vbx_sp_malloc( NTAPS * sizeof(*v_taps) );
    v_mac = vbx_sp_malloc( sizeof(*v_mac) );
```

- **Move data from memory → scratchpad (normally DMA)**

```
for( idx=0; idx < NTAPS; idx++ ) {
    v_taps[idx] = filter_taps[idx];
}
```

# Example: FIR (easy, stalls)

```
static vbx_half_t *v_taps;
static vbx_half_t *v_inp;
static vbx_half_t *v_out;
static vbx_word_t *v_mac;
```

- Set vector length and do vector ops

```
void vector_filter() {
    int i;
    for( i=0; i < SND_LEN-NTAPS; i++ ) {
        vbx_set_vl( NTAPS );
        vbx_acc( VVHW, VMUL, v_mac, (v_inp+i), v_taps );
        vbx_set_vl( 1 );
        vbx_sync();
        v_out[i] = (vbx_half_t) ( (v_mac[0]) >> 16 );
    }
}
```

# Example: FIR (no stalls)

```
static vbx_half_t *v_taps;
static vbx_half_t *v_inp;
static vbx_half_t *v_out;
static vbx_word_t *v_mac;
```

- **Set vector length and do vector ops**

```
void vector_filter() {
    int i;
    for( i=0; i < SND_LEN-NTAPS; i++ ) {
        vbx_set_vl( NTAPS );
        vbx_acc( VVHW, VMUL, v_mac, (v_inp+i), v_taps );
        vbx_set_vl( 1 );
        vbx( SVW, VSHR, v_mac, 16, v_mac );
        vbx( VVWH, VMOV, v_out+i, v_mac, 0 );
    }
    vbx_sync(); // move sync out of loop → faster
}
```

# Vector Assembly Code

```
// vector code
77c:    0107a023      sw      a6,0(a5) # 20000000 <st_twConfig+0x1ffd18fc>
780:    00a7a023      sw      a0,0(a5)
784:    00a7a023      sw      a0,0(a5)
788:    0007a023      sw      zero,0(a5)
78c:    0117a023      sw      a7,0(a5)
790:    00e7a023      sw      a4,0(a5)
794:    0057a023      sw      t0,0(a5)
798:    00d7a023      sw      a3,0(a5)
79c:    0107a023      sw      a6,0(a5)
7a0:    00b7a023      sw      a1,0(a5)
7a4:    00b7a023      sw      a1,0(a5)
7a8:    0007a023      sw      zero,0(a5)
7ac:    0067a023      sw      t1,0(a5)
7b0:    01f7a023      sw      t6,0(a5)
7b4:    00d7a023      sw      a3,0(a5)
7b8:    00d7a023      sw      a3,0(a5)
7bc:    01e7a023      sw      t5,0(a5)
7c0:    00d7a023      sw      a3,0(a5)
7c4:    0007a023      sw      zero,0(a5)
7c8:    00c7a023      sw      a2,0(a5)
7cc:    00270713      addi   a4,a4,2
7d0:    20000eb7      lui    t4,0x20000
7d4:    00260613      addi   a2,a2,2
7d8:    fbc712e3      bne   a4,t3,77c <vector_filter+0x40>
```

# Scalar Assembly Code

```
// scalar code

708:      00079703      lh      a4 , 0(a5)
70c:      00069583      lh      a1 , 0(a3)
710:      00278793      addi    a5 , a5 , 2
714:      00268693      addi    a3 , a3 , 2
718:      02b70733      mul     a4 , a4 , a1
71c:      00e60633      add     a2 , a2 , a4
720:      fef514e3      bne    a0 , a5 , 708 <scalar_filter+0x28>
```

# Example: Adding 3 Vectors

```
#include "vbx.h"

int main()
{
    const int length = 8;
    int A[length] = {1,2,3,4,5,6,7,8};
    int B[length] = {10,20,30,40,50,60,70,80};
    int C[length] = {100,200,300,400,500,600,700,800};
    int D[length];

    vbx_dcache_flush_all(); // for systems with cache

    const int data_len = length * sizeof(int);
    vbx_word_t *va = (vbx_word_t*)vbx_sp_malloc( data_len );
    vbx_word_t *vb = (vbx_word_t*)vbx_sp_malloc( data_len );
    vbx_word_t *vc = (vbx_word_t*)vbx_sp_malloc( data_len );

    vbx_dma_to_vector( va, A, data_len );
    vbx_dma_to_vector( vb, B, data_len );
    vbx_dma_to_vector( vc, C, data_len );

    vbx_set_vl( length );
    vbx( VVW, VADD, vb, va, vb );
    vbx( VVW, VADD, vc, vb, vc );

    vbx_dma_to_host( D, vc, data_len );

    vbx_sync();
    vbx_sp_free();
}
```

# Example: Transposed FIR

```
// transposed FIR filter
void vector_fir( input_type *input, output_type *output, input_type *coeffs, int sample_size, int num_taps )
{
    // need 5 vectors of size chunk_size+scratchpad+padding, round to 1/8th of remaining scratchpad memory
    int chunk_size      = (vbx_sp_getfree() >> 3)/OUTPUT_WIDTH;

    input_type  *sample_on_vpu = (input_type  *)vbx_sp_malloc( (           chunk_size+num_taps)*INPUT_WIDTH  );
    output_type *mult         = (output_type *)vbx_sp_malloc( (           chunk_size+num_taps)*OUTPUT_WIDTH );
    output_type *dest_on_vpu  = (output_type *)vbx_sp_malloc( (num_taps+chunk_size+num_taps)*OUTPUT_WIDTH );

    dest_on_vpu  += num_taps;

    int j;
    for( chunk_start=0; chunk_start < sample_size; chunk_start += chunk_size ) {

        vbx_dma_to_vector( sample_on_vpu, input+chunk_start, (chunk_size+num_taps)*INPUT_WIDTH );

        output_type *temp_dest = dest_on_vpu;
        vbx_set_vl( chunk_size+num_taps ); // sets vector length

        vbx( SVHU, VMULLO, temp_dest, coeffs[0], sample_on_vpu );
        for( j = 1; j < num_taps; j++ ) {
            temp_dest--;
            vbx( SVHU, VMULLO, mult,      coeffs[j], sample_on_vpu );
            vbx( VVHU, VADD,   temp_dest, temp_dest, mult );
        }

        vbx_dma_to_host( output+chunk_start, dest_on_vpu, chunk_size*OUTPUT_WIDTH );
    }

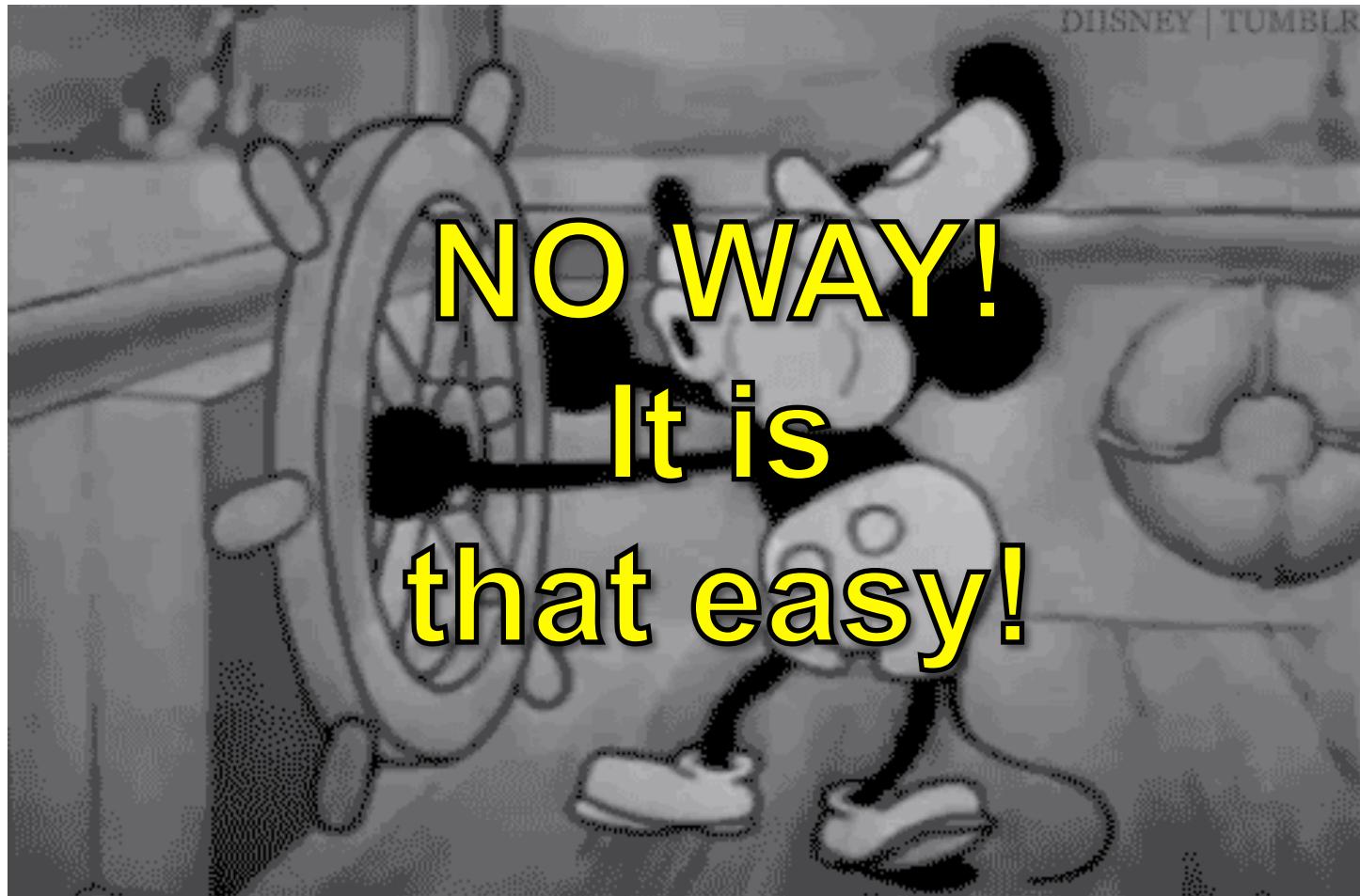
    vbx_sync();
    vbx_sp_free();
}
```

# Documentation

- MXP Programming Reference
  - Specifies each programming layer
- MXP Guide
  - Customized for FPGA vendor
  - Slightly more introductory, incomplete as a reference
  - Used for simulator too

# How to Program FPGAs

it's easy! <http://demo.vectorblox.ca>



# Additional Access

<http://www.github.com/vectorblob/mxp>

Hosted on GitHub:

- All software source code (BSD license)
- All documentation
- Pregenerated systems (bitstreams)
  - Select Altera boards (DE2-115, DE4)
  - Select Xilinx boards (ZedBoard, ZC706)
  - **NEW!** Microsemi 060 board
- Functional simulator (needs license to run)
- Encrypted IP (needs license)

# Conditional Execution

- Example code

- Saturate vector `v_val[]` to 100

```
vbx( SVB, VSUB,      v_sub, 100, v_val );  
vbx( SVB, VCMV_LTZ, v_val, 100, v_sub );
```

- Rough idea

```
if( 100 - v_val[i] < 0 ) v_val[i] = 100;
```

- Formal equivalent

```
for( i=0; i<VL; i++ )  
    v_sub[i] = 100 - v_val[i];  
    if( 100 < v_val[i] )  
        v_val[i] = 100;
```

# 2D/3D Matrices or Submatrices

- Matrices and submatrices
  - 2D matrices use standard C packed format
    - Row-major order
    - Padding between rows allowed
  - 3D matrices
    - One 2D matrix packed after another
    - Padding between matrices allowed
  - MXP can access sub-matrices (1D, 2D, 3D)
    - Use vector length < row length
    - Specify strides, number of iterations

# Mask Registers

- GPUs – control flow divergence
  - Not all elements “active”
  - Some turned into NOPs
  - Waste of performance
- Density time masking
  - Memorize position of NOPs
  - Skip over them (sequence of NOPs)
- Speeds up “early exit” algorithms
  - Needs some locality of NOPs

# Example Code: 2D Matrix

2D Matrix VBX code:

```
vbx_half_t *vdest, *vsrcl, *vsrc2;  
vbx_set_vl( vl );  
vbx_set_2D( numRows, iD2, iA2, iB2 );  
vbx_2D( VVH, VADD, vdest, vsrcl, vsrcl );
```

Equivalent C code:

```
vbx_half_t *dest, *srcA, *srcB;  
for( r = 0; r < numRows; r++ ) {  
    dest = (vbx_half_t*) ( (vbx_byte_t*) vdest + (r*iD2) );  
    srcA = (vbx_half_t*) ( (vbx_byte_t*) vsrcl + (r*iA2) );  
    srcB = (vbx_half_t*) ( (vbx_byte_t*) vsrc2 + (r*iB2) );  
    for( c=0; c < vl; c++ ) {  
        dest[c] = srcA[c] + srcB[c];  
    }  
}
```

VBX 2D code to add two matrices of halfwords, and the equivalent C code.

# Example Code: 3D Matrix

3D Matrix VBX code:

```
vbx_half_t *vdest, *vsrcl, *vsrc2;  
vbx_set_vl( vl );  
vbx_set_2D( numRows, iD2, iA2, iB2 );  
vbx_set_3D( numMats, iD3, iA3, iB3 );  
vbx_3D( VVH, VADD, vdest, vsrcl, vsrc2 );
```

Equivalent C code:

```
vbx_half_t *dest, *srcA, *srcB;  
for( m = 0; m < numMats; m++ ) {  
    for( r = 0; r < numRows; r++ ) {  
        dest = (vbx_half_t*) ( (vbx_byte_t*) vdest + (m*iD3) + (r*iD2) );  
        srcA = (vbx_half_t*) ( (vbx_byte_t*) vsrcl + (m*iA3) + (r*iA2) );  
        srcB = (vbx_half_t*) ( (vbx_byte_t*) vsrc2 + (m*iB3) + (r*iB2) );  
        for( c=0; c < vl; c++ ) {  
            dest[c] = srcA[c] + srcB[c];  
        }  
    }  
}
```

VBX 3D code to add two volumes of halfwords, and the equivalent C code. 51



## Using Online MXP Simulator

demo@localhost: ~

demo.vectorblox.ca

Search

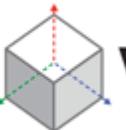
localhost login: demo  
Password:  
Last login: Tue Sep 1 02:59:24 UTC 2015 on pts/3  
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 4.1.5-x86\_64-linode61 x86  
64)  
  
\* Documentation: <https://help.ubuntu.com/>  
  
demo@localhost:~\$

user: demo  
password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to demo@demo.vectorblox.ca)
2. copy the read-only simulator directory into your own unique working directory  
cp -r mxp-simulator\_or DIR





**Vector**  
embedded super

user: demo  
password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to `demo@demo.vectorblox.ca`)
2. copy the read-only simulator directory into your own unique working directory  
`cp -r mxp-simulator_only DIR`

```
localhost login: demo
Password:
Last login: Tue Sep  1 05:19:51 UTC 2015 on pts/2
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 4.1.5-x86_64-linode61 x86
64)

 * Documentation:  https://help.ubuntu.com/

demo@localhost:~$ ls
mxp-simulator_only
demo@localhost:~$ cp -r mxp-simulator_only user-guy
demo@localhost:~$ cd user-guy
demo@localhost:~/user-guy$ ls
docs examples id README.md repository
demo@localhost:~/user-guy$ cd examples/software/simple/
demo@localhost:~/user-guy/examples/software/simple$ ls
Makefile README.md test-c.c test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$
```

demo@localhost: ~/user-guy/... +

demo.vectorblox.ca Search

**VectorBlo**x  
embedded supercomputing

user: demo  
password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to `demo@demo.vectorblox.ca`)
2. copy the read-only simulator directory into your own unique working directory  
`cp -r mxp-simulator_only DIR`
3. change into the software directory  
`cd DIR/examples`  
`/software/simple`
4. edit the code  
`vim test-c.c`  
or  
`vim test-cpp.cpp`
5. build for the simulator  
`make SIMULATOR=true`
6. run the program  
`./test-c.elf`  
or  
`./test-cpp.elf`

```
localhost login: demo
Password:
Last login: Tue Sep  1 05:19:51 UTC 2015 on pts/2
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 4.1.5-x86_64-linode61 x86
_64)

 * Documentation:  https://help.ubuntu.com/

demo@localhost:~$ ls
mxp-simulator_only
demo@localhost:~$ cp -r mxp-simulator_only user-guy
demo@localhost:~$ cd user-guy
demo@localhost:~/user-guy$ ls
docs examples id README.md repository
demo@localhost:~/user-guy$ cd examples/software/simple/
demo@localhost:~/user-guy/examples/software/simple$ ls
Makefile README.md test-c.c test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$ vim test-c.c
demo@localhost:~/user-guy/examples/software/simple$
```

 **VectorBlox**  
 embedded supercomputing

user: demo  
 password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to `demo@demo.vectorblox.ca`)
2. copy the read-only simulator directory into your own unique working directory  
`cp -r mxp-simulator_only DIR`
3. change into the software directory  
`cd DIR/examples /software/simple`
4. edit the code  
`vim test-c.c`  
 or  
`vim test-cpp.cpp`
5. build for the simulator  
`make SIMULATOR=true`
6. run the program  
`./test-c.elf`  
 or  
`./test-cpp.elf`

```

#include <stdio.h>
#include "vbx.h"

const int num_elements=10;

int main()
{
#if VBX_SIMULATOR==1
    //initialize with 4 lanes, and 64kb of sp memory
    //word, half, byte fraction bits 16,15,4 respectively
    vbxsim_init( 4, 64, 256, 6, 5, 4 );
#endif

    //Allocate vectors in scratchpad
    vbx_word_t* a = vbx_sp_malloc( num_elements*sizeof(vbx_word_t) );
    vbx_word_t* b = vbx_sp_malloc( num_elements*sizeof(vbx_word_t) );
    vbx_word_t* c = vbx_sp_malloc( num_elements*sizeof(vbx_word_t) );

    //Set vector length, then compute 4*[1,2,3,...,10]
    vbx_set_vl(num_elements);
    vbx(SEW,VADD,a,1,0); //a = [1,2,3,...,10]
    vbx(SVW,VMOV,b,4,0); //b = [4,4,...,4]
    vbx(VWW,VMUL,c,a,b); //c = a * b

    //wait for all vector instructions to finish
    vbx_sync();

    //print out vector c
    printf( "%8d", c[0] );
    for( int i=1; i<num_elements; i++ ) {
        printf( ",%8d", c[i] );
    }
    printf( "\n" );

    return 0;
}
~"test-c.c" 38L, 886C
  
```

1,1      All

demo@localhost: ~/user-guy/... +

demo.vectorblox.ca Search

**VectorBlo**x  
embedded supercomputing

user: demo  
password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to `demo@demo.vectorblox.ca`)
2. copy the read-only simulator directory into your own unique working directory  
`cp -r mxp-simulator_only DIR`
3. change into the software directory  
`cd DIR/examples /software/simple`
4. edit the code  
`vim test-c.c`  
 or  
`vim test-cpp.cpp`
5. build for the simulator  
`make SIMULATOR=true`
6. run the program  
`./test-c.elf`  
 or  
`./test-cpp.elf`

```

localhost login: demo
Password:
Last login: Tue Sep  1 05:20:47 UTC 2015 on pts/2
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 4.1.5-x86_64-linode61 x86_64)

 * Documentation: https://help.ubuntu.com/

demo@localhost:~$ ls
mxp-simulator_only
demo@localhost:~$ cp -r mxp-simulator_only user-guy
demo@localhost:~$ cd user-guy/
demo@localhost:~/user-guy$ ls
docs examples id README.md repository
demo@localhost:~/user-guy$ cd examples/software/simple/
demo@localhost:~/user-guy/examples/software/simple$ ls
Makefile README.md test-c.c test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$ vim test-c.c
demo@localhost:~/user-guy/examples/software/simple$ make SIMULATOR=true
make -C ../../repository/lib/vbxapi/ SIMULATOR=true
make[1]: Entering directory '/home/demo/user-guy/repository/lib/vbxapi'
mkdir -p obj/SIMULATOR
gcc -xc -MD -c -g -O3 -Wall -DVBX_SIMULATOR \
     -MD -o"obj/SIMULATOR/vbx_api.c.o" "vbx_api.c"
gcc -xc++ -MD -c -g -O3 -Wall -DVBX_SIMULATOR -fno-rtti -fno-exceptions \
     -MD -o"obj/SIMULATOR/Vector.cpp.o" "Vector.cpp"
ar -r "obj/SIMULATOR/libvbxapi.a" obj/SIMULATOR/vbx_api.c.o obj/SIMULATOR/Vector.cpp.o
ar: creating obj/SIMULATOR/libvbxapi.a
cp obj/SIMULATOR/libvbxapi.a libvbxapi.a
make[1]: Leaving directory '/home/demo/user-guy/repository/lib/vbxapi'
make -C ../../repository/lib/vbxsim/ SIMULATOR=true
make[1]: Entering directory '/home/demo/user-guy/repository/lib/vbxsim'
cp libvbxsim_linux.a libvbxsim.a
make[1]: Leaving directory '/home/demo/user-guy/repository/lib/vbxsim'
g++ -Wall -g -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-cpp.elf test-cpp.cpp
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
gcc -Wall -g -std=c99 -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-c.elf test-
c.c ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
demo@localhost:~/user-guy/examples/software/simple$ 
```

 **VectorBlox**  
 embedded supercomputing

user: demo  
 password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to `demo@demo.vectorblox.ca`)
2. copy the read-only simulator directory into your own unique working directory  
`cp -r mxp-simulator_only DIR`
3. change into the software directory  
`cd DIR/examples /software/simple`
4. edit the code  
`vim test-c.c`  
 or  
`vim test-cpp.cpp`
5. build for the simulator  
`make SIMULATOR=true`
6. run the program  
`./test-c.elf`  
 or  
`./test-cpp.elf`

```

Password:  

Last login: Tue Sep  1 05:20:47 UTC 2015 on pts/2  

Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 4.1.5-x86_64-linode61 x86_64)

 * Documentation: https://help.ubuntu.com/

demo@localhost:~$ ls
mxp-simulator_only
demo@localhost:~$ cp -r mxp-simulator_only user-guy
demo@localhost:~$ cd user-guy/
demo@localhost:~/user-guy$ ls
docs examples id README.md repository
demo@localhost:~/user-guy$ cd examples/software/simple/
demo@localhost:~/user-guy/examples/software/simple$ ls
Makefile README.md test-c.c test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$ vim test-c.c
demo@localhost:~/user-guy/examples/software/simple$ make SIMULATOR=true
make -C ../../repository/lib/vbxapi/ SIMULATOR=true
make[1]: Entering directory `/home/demo/user-guy/repository/lib/vbxapi'
mkdir -p obj/SIMULATOR
gcc -xc -MD -c -g -O3 -Wall -DVBX_SIMULATOR \
     -MD -o"obj/SIMULATOR/vbx_api.c.o" "vbx_api.c"
gcc -xc++ -MD -c -g -O3 -Wall -DVBX_SIMULATOR -fno-rtti -fno-exceptions \
     -MD -o"obj/SIMULATOR/Vector.cpp.o" "Vector.cpp"
ar -r "obj/SIMULATOR/libvbxapi.a" obj/SIMULATOR/vbx_api.c.o obj/SIMULATOR/Vector.cpp.o
ar: creating obj/SIMULATOR/libvbxapi.a
cp obj/SIMULATOR/libvbxapi.a libvbxapi.a
make[1]: Leaving directory `/home/demo/user-guy/repository/lib/vbxapi'
make -C ../../repository/lib/vbxsim/ SIMULATOR=true
make[1]: Entering directory `/home/demo/user-guy/repository/lib/vbxsim'
cp libvbxsim_linux.a libvbxsim.a
make[1]: Leaving directory `/home/demo/user-guy/repository/lib/vbxsim'
g++ -Wall -g -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-cpp.elf test-cpp.cpp
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
gcc -Wall -g -std=c99 -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-c.elf test-c.c
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
demo@localhost:~/user-guy/examples/software/simple$ ./test-c.elf
License validated.
        4,          8,         12,         16,         20,         24,         28,         32,         36,         40
demo@localhost:~/user-guy/examples/software/simple$ 
  
```

demo@localhost: ~/user-guy/... +

demo@localhost ~ user-guy

Search

 **VectorBlox**  
embedded supercomputing

user: demo  
password: vbxdemo

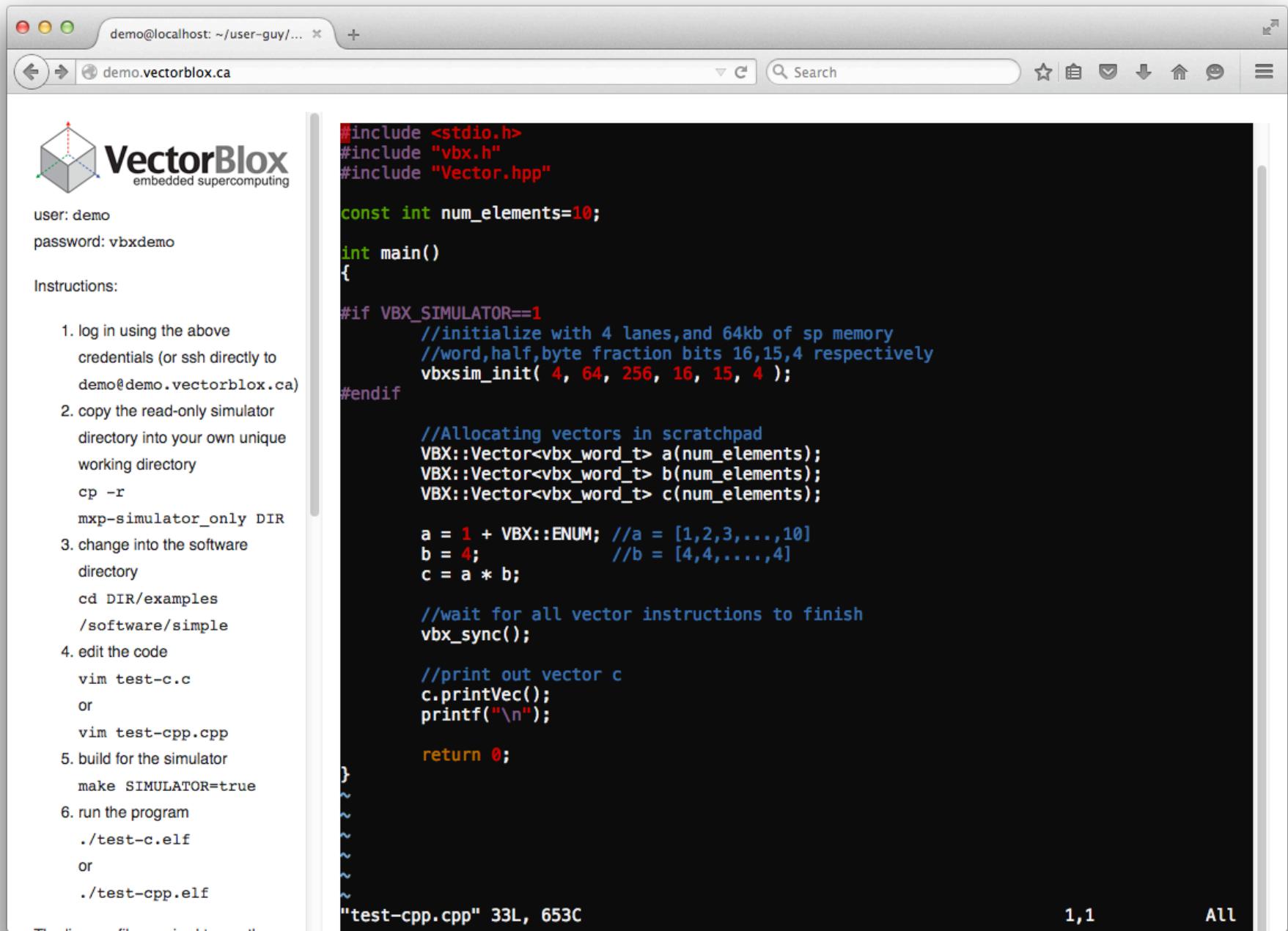
Instructions:

1. log in using the above credentials (or ssh directly to `demo@demo.vectorblox.ca`)
2. copy the read-only simulator directory into your own unique working directory  
`cp -r mxp-simulator_only DIR`
3. change into the software directory  
`cd DIR/examples /software/simple`
4. edit the code  
`vim test-c.c`  
 or  
`vim test-cpp.cpp`
5. build for the simulator  
`make SIMULATOR=true`
6. run the program  
`./test-c.elf`  
 or  
`./test-cpp.elf`

```
Last login: Tue Sep 1 05:20:47 UTC 2015 on pts/2
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 4.1.5-x86_64-linode61 x86_64)

 * Documentation: https://help.ubuntu.com/

demo@localhost:~$ ls
mxp-simulator_only
demo@localhost:~$ cp -r mxp-simulator_only user-guy
demo@localhost:~$ cd user-guy/
demo@localhost:~/user-guy$ ls
docs examples id README.md repository
demo@localhost:~/user-guy$ cd examples/software/simple/
demo@localhost:~/user-guy/examples/software/simple$ ls
Makefile README.md test-c.c test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$ vim test-c.c
demo@localhost:~/user-guy/examples/software/simple$ make SIMULATOR=true
make -C ../../repository/lib/vbxapi/ SIMULATOR=true
make[1]: Entering directory `/home/demo/user-guy/repository/lib/vbxapi'
mkdir -p obj/SIMULATOR
gcc -xc -MD -c -g -O3 -Wall -DVBX_SIMULATOR \
    -MD -o"obj/SIMULATOR/vbx_api.c.o" "vbx_api.c"
gcc -xc++ -MD -c -g -O3 -Wall -DVBX_SIMULATOR -fno-rtti -fno-exceptions \
    -MD -o"obj/SIMULATOR/Vector.cpp.o" "Vector.cpp"
ar -r "obj/SIMULATOR/libvbxapi.a" obj/SIMULATOR/vbx_api.c.o obj/SIMULATOR/Vector.cpp.o
ar: creating obj/SIMULATOR/libvbxapi.a
cp obj/SIMULATOR/libvbxapi.a libvbxapi.a
make[1]: Leaving directory `/home/demo/user-guy/repository/lib/vbxapi'
make -C ../../repository/lib/vbxsim/ SIMULATOR=true
make[1]: Entering directory `/home/demo/user-guy/repository/lib/vbxsim'
cp libvbxsim_linux.a libvbxsim.a
make[1]: Leaving directory `/home/demo/user-guy/repository/lib/vbxsim'
g++ -Wall -g -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-cpp.elf test-cpp.cpp
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
gcc -Wall -g -std=c99 -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-c.elf test-c.c
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
demo@localhost:~/user-guy/examples/software/simple$ ./test-c.elf
License validated.
          4,          8,         12,         16,         20,         24,         28,         32,         36,         40
demo@localhost:~/user-guy/examples/software/simple$ vim test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$
```



demo@localhost: ~user-guy/... +

demo@localhost: ~user-guy/... Search

 **VectorBlox**  
embedded supercomputing

user: demo  
password: vbxdemo

Instructions:

1. log in using the above credentials (or ssh directly to demo@demo.vectorblox.ca)
2. copy the read-only simulator directory into your own unique working directory  
cp -r mxp-simulator\_only DIR
3. change into the software directory  
cd DIR/examples/software /simple
4. edit the code  
vim test-c.c  
or  
vim test-cpp.cpp
5. build for the simulator  
make SIMULATOR=true
6. run the program  
. ./test-c.elf  
or  
. ./test-cpp.elf

\* Documentation: <https://help.ubuntu.com/>

```
demo@localhost:~$ ls
mxp-simulator_only
demo@localhost:~$ cp -r mxp-simulator_only user-guy
demo@localhost:~$ cd user-guy
demo@localhost:~/user-guy$ ls
docs examples id README.md repository
demo@localhost:~/user-guy$ cd examples/software/simple/
demo@localhost:~/user-guy/examples/software/simple$ ls
Makefile README.md test-c.c test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$ vi test-c.c
demo@localhost:~/user-guy/examples/software/simple$ make
make -C ../../repository/lib/vbxapi/ SIMULATOR=true
make[1]: Entering directory `/home/demo/user-guy/repository/lib/vbxapi'
mkdir -p obj/SIMULATOR
gcc -xc -MD -c -g -O3 -Wall -DVBX_SIMULATOR \
      -MD -o"obj/SIMULATOR/vbx_api.c.o" "vbx_api.c"
gcc -xc++ -MD -c -g -O3 -Wall -DVBX_SIMULATOR -fno-rtti -fno-exceptions \
      -MD -o"obj/SIMULATOR/Vector.cpp.o" "Vector.cpp"
ar -r "obj/SIMULATOR/libvbxapi.a" obj/SIMULATOR/vbx_api.c.o obj/SIMULATOR/Vector.cpp.o
ar: creating obj/SIMULATOR/libvbxapi.a
cp obj/SIMULATOR/libvbxapi.a libvbxapi.a
make[1]: Leaving directory `/home/demo/user-guy/repository/lib/vbxapi'
make -C ../../repository/lib/vbxsim/ SIMULATOR=true
make[1]: Entering directory `/home/demo/user-guy/repository/lib/vbxsim'
cp libvbxsim_linux.a libvbxsim.a
make[1]: Leaving directory `/home/demo/user-guy/repository/lib/vbxsim'
g++ -Wall -g -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-cpp.elf test-cpp.cpp \
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
gcc -Wall -g -std=c99 -DVBX_SIMULATOR -I../../repository/lib/vbxapi -o test-c.elf test-c.c \
    ../../repository/lib/vbxapi/libvbxapi.a ../../repository/lib/vbxsim/libvbxsim.a
demo@localhost:~/user-guy/examples/software/simple$ ./test-c.elf
License validated.
        4,          8,         12,         16,         20,         24,         28,         32,         36,         40
demo@localhost:~/user-guy/examples/software/simple$ vi test-cpp.cpp
demo@localhost:~/user-guy/examples/software/simple$ ./test-cpp.elf
License validated.
        4,          8,         12,         16,         20,         24,         28,         32,         36,         40
demo@localhost:~/user-guy/examples/software/simple$
```

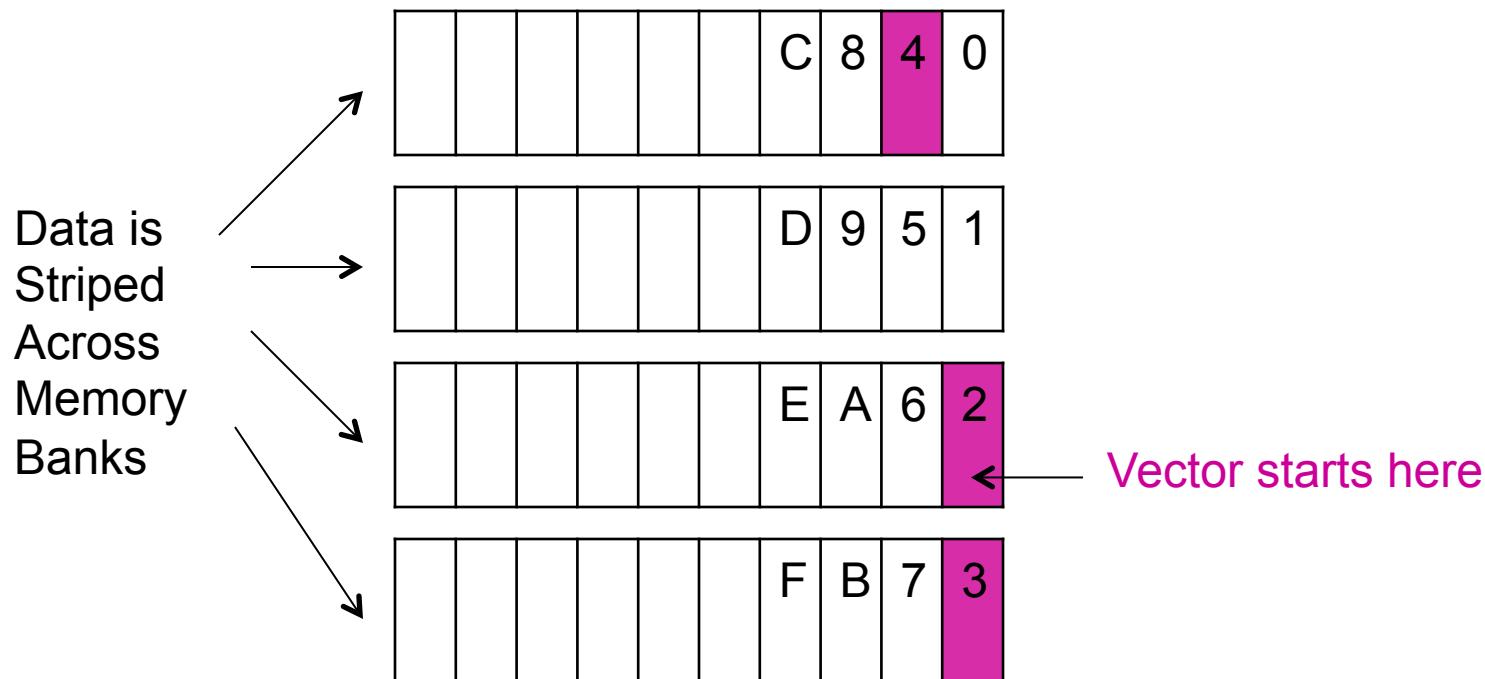
# Summary

- VectorBlox MXP™ advantages
  - Easy to use/deploy
  - Easy to program/edit/debug on FPGAs
  - Faster time to market
  - SW vs HW focus – no RTL, no synthesis, no P&R
  - Scalable performance (area vs speed)
    - Speedups up to 1000x
  - No hardware recompiling necessary
    - Rapid algorithm development
    - Hardware purely ‘sandboxed’ from algorithm



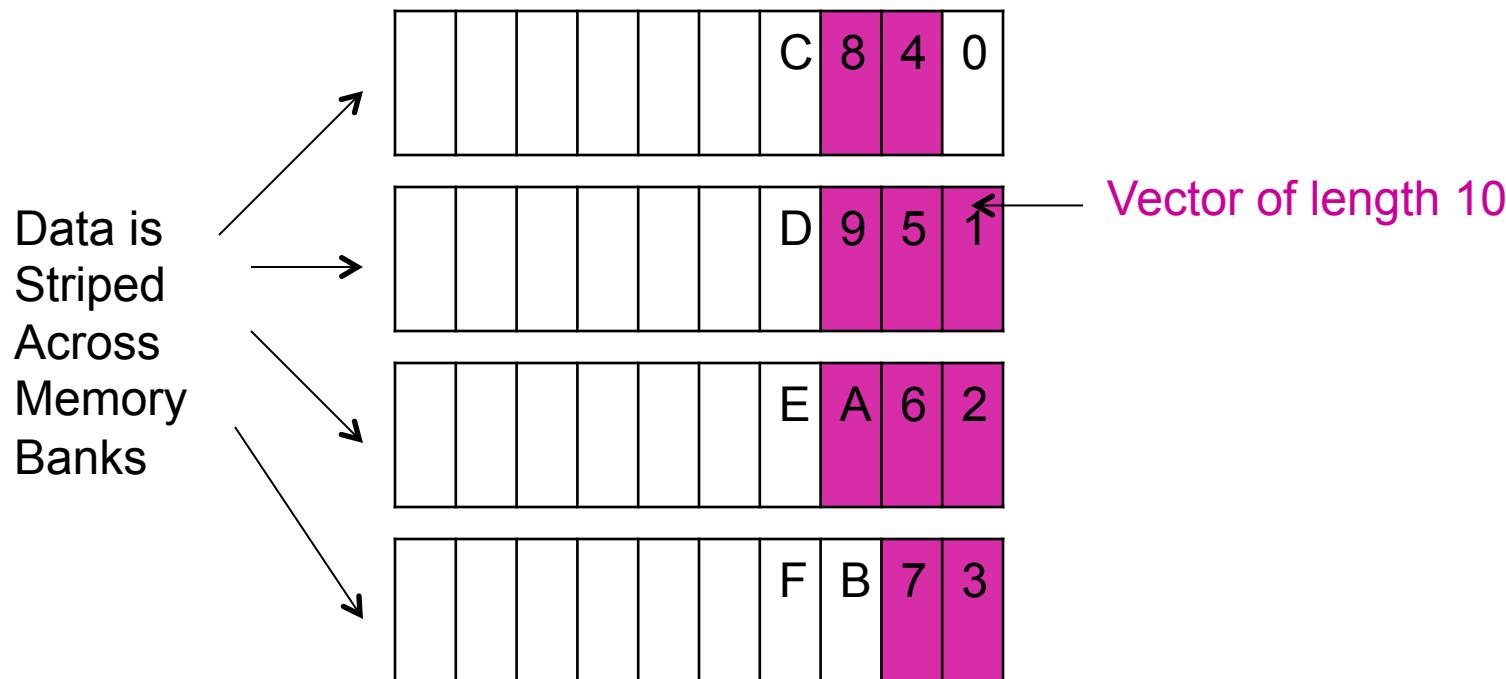
# Scratchpad Memory

- Multi-banked, parallel access
  - Addresses striped across banks, like RAID disks
  - `vbx_sp_malloc(12)` allocates 3 words in Scratchpad
  - Vector can start at any location



# Scratchpad Memory

- Multi-banked, parallel access
  - Addresses striped across banks, like RAID disks
  - `vbx_sp_malloc(12)` allocates 3 words in Scratchpad
  - Vector can start at any location, can have any length



# Scratchpad Memory

- Multi-banked, parallel access
  - Vector can start at any location, can have any length
  - Every cycle, can read one “wave” of elements striped across all banks (up to the end of the vector)

