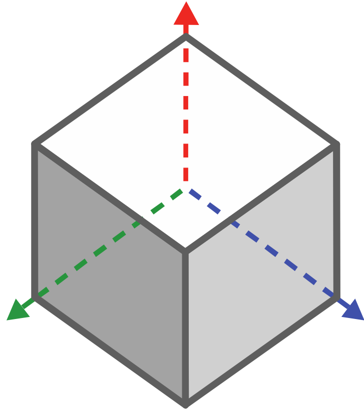


VectorBlox
embedded supercomputing

FPGA-based Embedded Supercomputing with the VectorBlox MXP™ Matrix Processor

Presentation by Guy Lemieux

glemieux@vectorblox.com

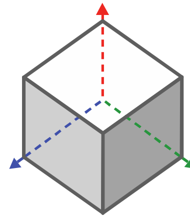
http://www.vectorblox.com

© 2012 VectorBlox Computing Inc.

Overview

- Typical Usage and Motivation
- System Design
- Programmer's View
- Programming Examples

© 2012 VectorBlox Computing Inc.



Medical

```
FORCE_INLINE void MurmurHash3_128_vec4()
{
  uint32_t *vc1    = sp->vc1;
  uint32_t *vc2    = sp->vc2;
  uint32_t *vc3    = sp->vc3;
  uint32_t *vh     = sp->vh;
  uint32_t *vkrot  = sp->vkrot;
  uint32_t *vhrot  = sp->vhrotamt;

  // force initial seeds to zero
  vbx_set_vl( 4 );
  vbx( VVW, VXOR,   sp->vh, sp->vh, sp->vh );

  int i;
  for( i=0; i < MMVL; i += 4 ) {
    uint32_t *vk = sp->vk + i;
    vbx( VVW , VMOV,   vh+0, vh+1, NULL );
    vbx( VVW , VMOV,   vh+7, vh,   NULL );

    vbx( VVWU, VMULLO, vk,   vk,    vc1 );
    vbx( VVWU, VROTL,  vk,   vkrot, vk  );
    vbx( VVWU, VMULLO, vk,   vk,    vc2 );
    vbx( VVW , VXOR,   vh,   vh,    vk  );
    vbx( VVWU, VROTL,  vh,   vhrot, vh  );

    vbx( VVW , VADD,   vh,   vh, vh+4 );
    vbx( SVWU, VMULLO, vh,   5, vh    );
    vbx( VVW , VADD,   vh,   vh, vc3  );
  }
}
                                   1,1          All
```

Imaging

Industrial

Telecom

© 2012 VectorBlox Computing Inc.

Typical Usage and Motivation

- Embedded processing
  - FPGAs often control custom devices
    - Imaging, audio, radio, screens
  - Heavy data processing requirements
- FPGA tools for data processing
  - VHDL too difficult to learn and use
  - C-to-hardware tools too “VHDL-like”
  - FPGA-based CPUs (Nios/MicroBlaze) too slow
- Complications
  - Very slow recompiles of FPGA bitstream
  - Device control circuits may have sensitive timing requirements

© 2012 VectorBlox Computing Inc.

A New Tool

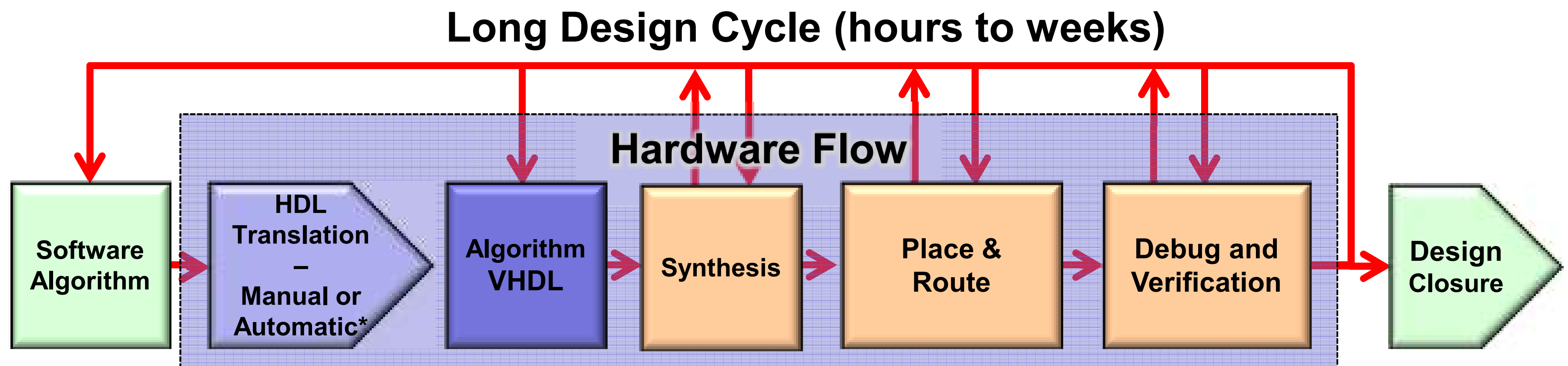
- **MXP™ Matrix Processor**
  - Performance
    - 100x – 1000x over Nios II/f, MicroBlaze
  - Easy to use, pure software
    - Just C, no VHDL/Verilog !
  - No FPGA recompilation for each algorithm change
    - Save time (FPGA place+route can take hours, run out of space, etc)
  - Correctness
    - Easy-to-debug, e.g. printf() or gdb
    - Simulator runs on PC, eg regression testing
    - Run on real FPGA hardware, eg real-time testing

© 2012 VectorBlox Computing Inc.

SYSTEM DESIGN WITH MXP™

© 2012 VectorBlox Computing Inc.

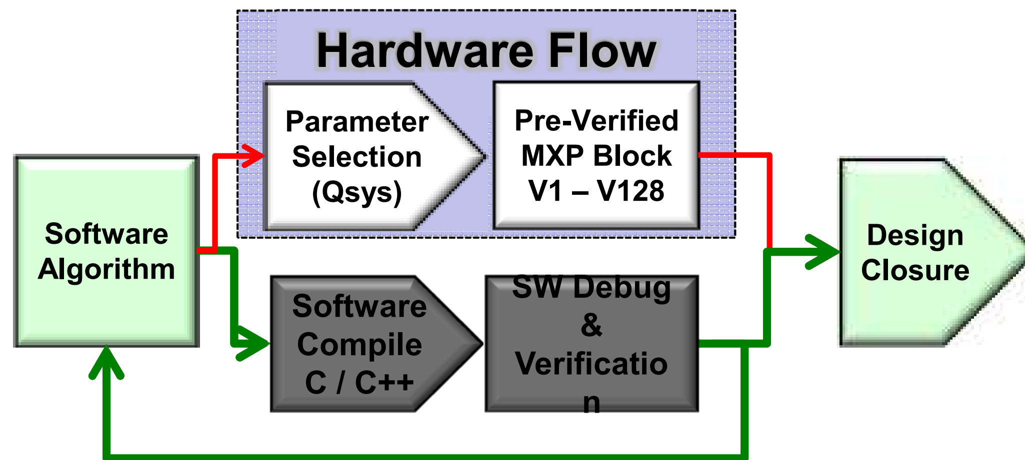
Typical Hardware Design Flow



*** Automatic = C to HDL (VHDL) programs**

© 2012 VectorBlox Computing Inc.

Design Flow using MXP Matrix Processor



Key Benefits

- Pre Verified HW Blocks
- Decouples HW & SW
- NO VHDL required
- Iterate quickly in SW
- Low cost changes
- Faster time to market
- Field upgradable

© 2012 VectorBlox Computing Inc.

Altera and Xilinx Support

- Use with Altera Nios II/f or Xilinx MicroBlaze

  Nios/MicroBlaze + MXP Accelerator = MXP™ Matrix Processor

- Nios/MicroBlaze runs regular C code

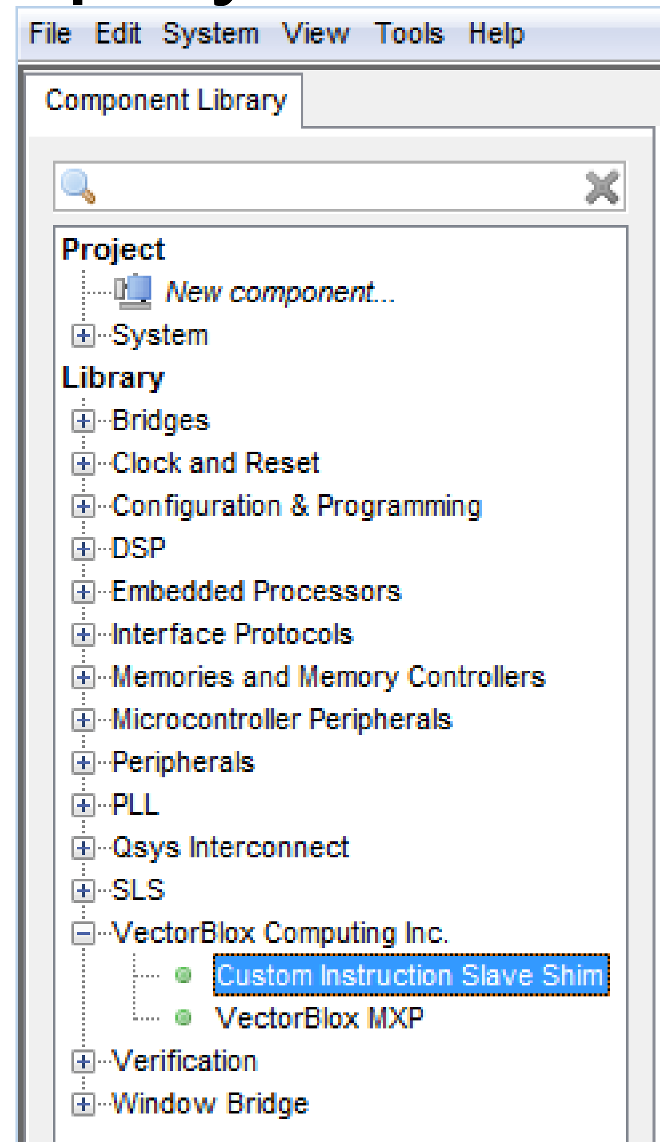
- MXP Accelerator runs special C function calls
  - Must organize data into contiguous vectors
  - Execution decoupled from Nios/MicroBlaze

- Rest of talk: assumes Altera Nios environment

© 2012 VectorBlox Computing Inc.

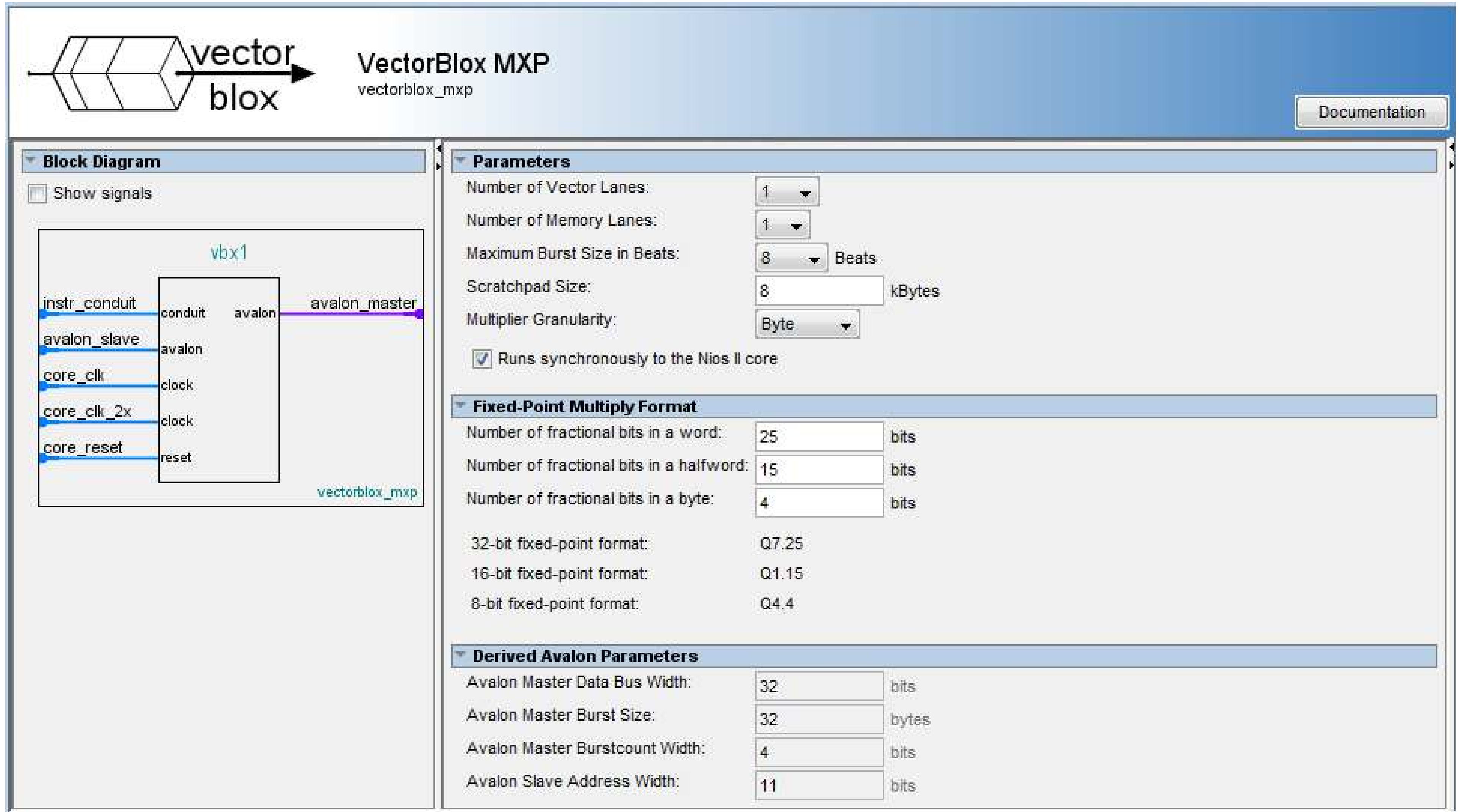
Easy to Deploy

- Add processor to system using Qsys



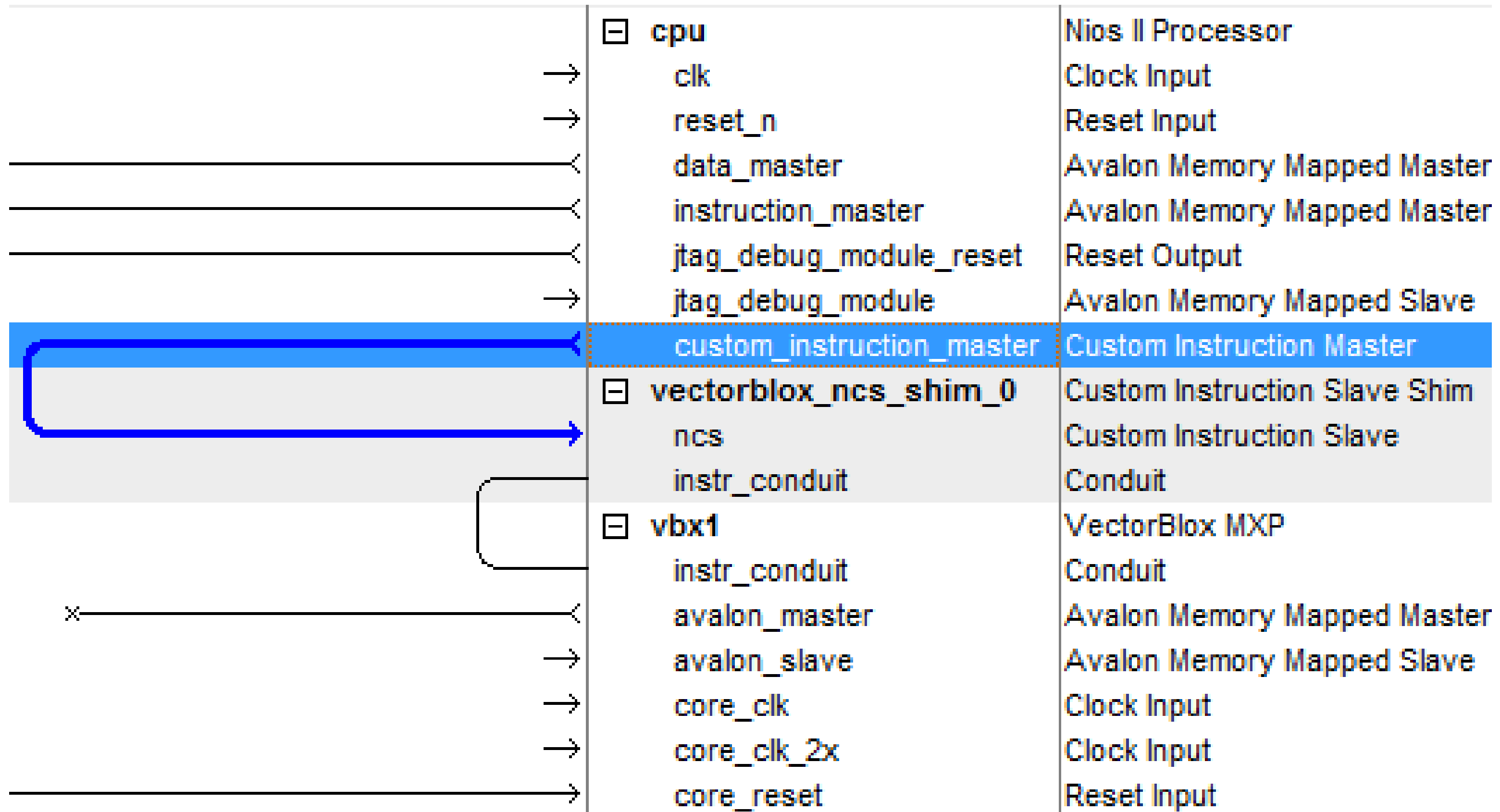
© 2012 VectorBlox Computing Inc.

Configure the MXP™ Processor



© 2012 VectorBlox Computing Inc.

Connecting to Nios II



© 2012 VectorBlox Computing Inc.

Summary

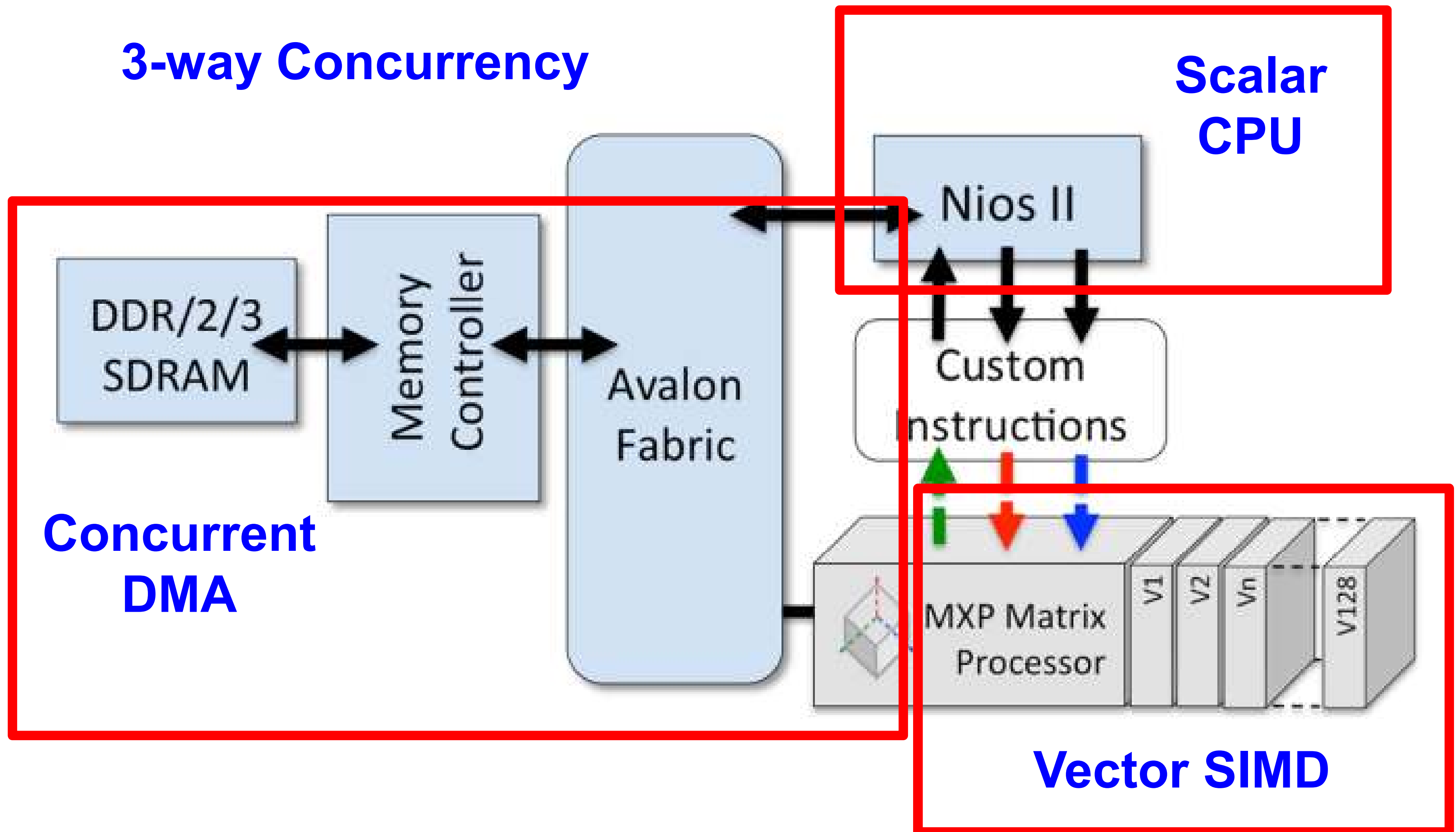
- HW and SW development is decoupled
- Select HW parameters and go
  - No VHDL required for computing
    - Still need VHDL to control custom devices
- Design SW with these main concepts
  - Vector
  - Scratchpad based computing
  - Same software can run on any FPGA

© 2012 VectorBlox Computing Inc.

MXP™ MATRIX PROCESSOR

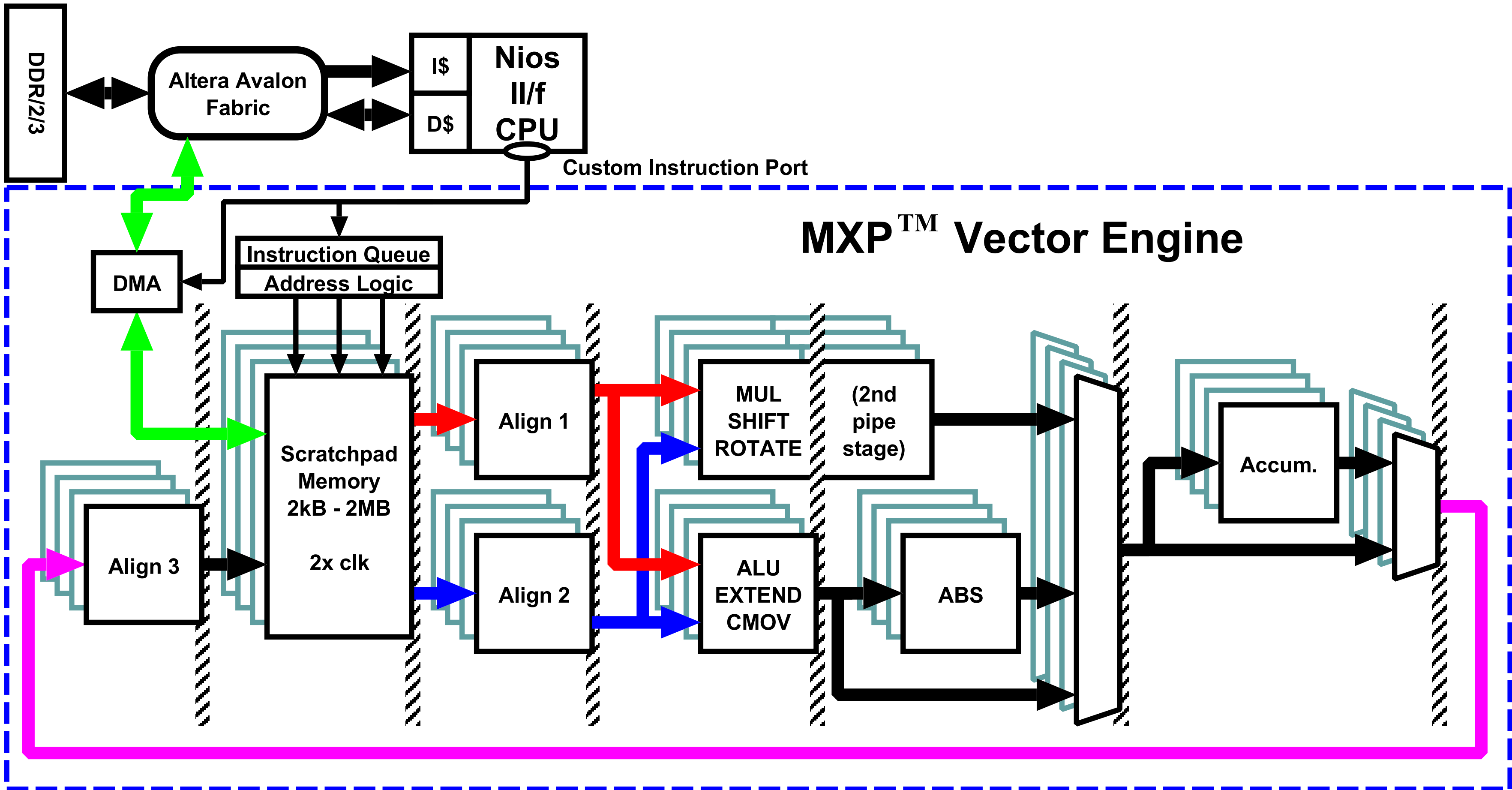
© 2012 VectorBlox Computing Inc.

MXP™ System Architecture



© 2012 VectorBlox Computing Inc.

MXP Internal Architecture



© 2012 VectorBlox Computing Inc.

MXP Performance

- Configurable Area ~ Performance

  # of Lanes ~ each lane contains 32b ALU + scratchpad

  V1: one 32b lane ~~ size of Nios/MicroBlaze

  V2: two 32b lanes ~~ twice size of Nios/MB

  V4, V8, ...

  V64 ~ largest size we've built to date

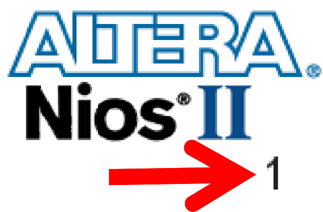
- Performance Features
  - High-bandwidth Scratchpad (~4-8kB per lane, V16 ~ 128kB)
  - Asynchronous DMA
  - Fracturable Lanes (1x32, 2x16, 4x8 bit operations)
  - Variable-length Vectors

© 2012 VectorBlox Computing Inc.

Performance Examples

Speedup (factor)

1000
100
10
1



V1 V2 V4 V8 V16 V32 V64

VectorBlox MXP™ Processor Size

Application Kernels

- autocor
- rgbcmyk
- rgbyiq
- imgblend
- filt3x3
- median
- motest
- fir
- matmul
- geomean

© 2012 VectorBlox Computing Inc.

Chip Area Requirements

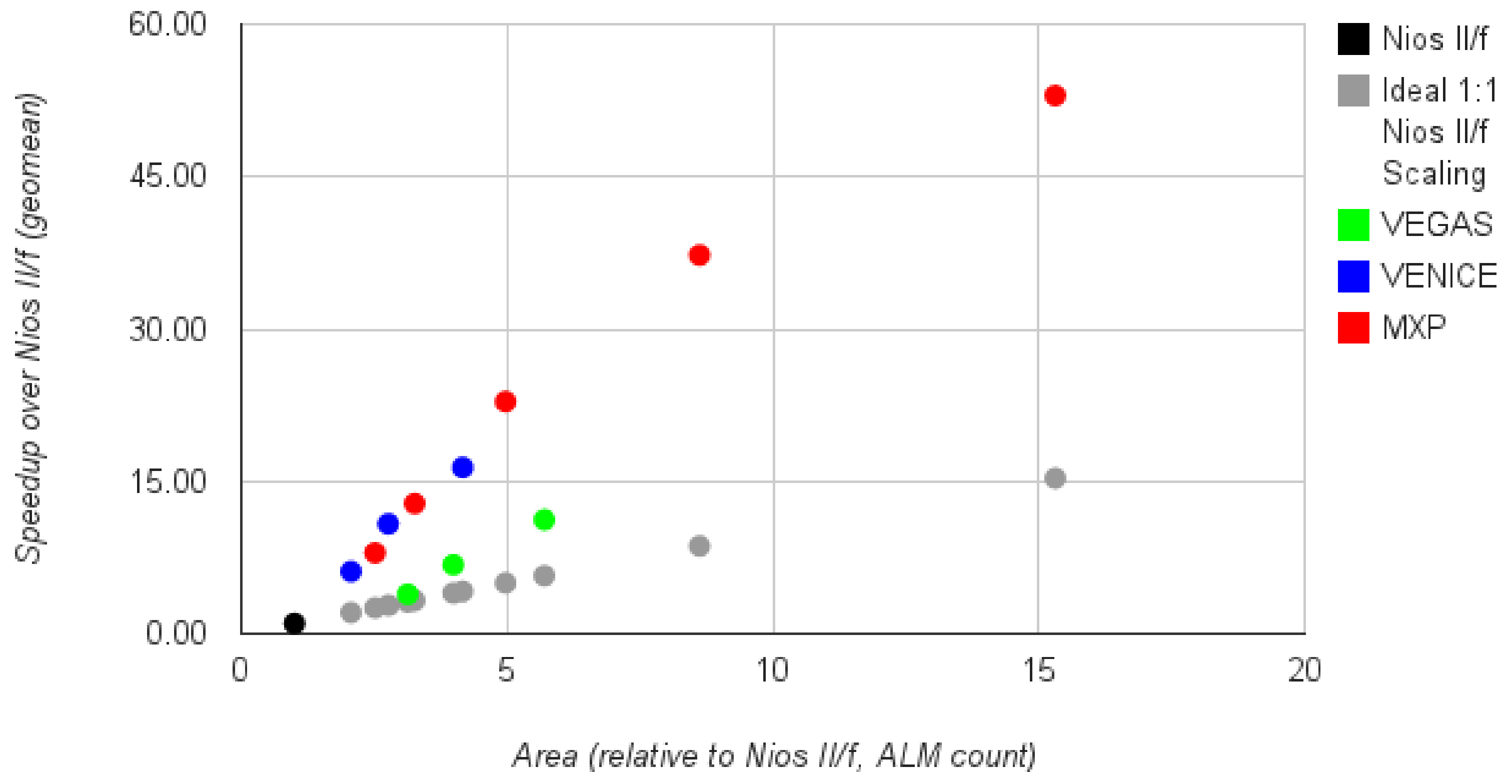
	Nios II/f				V8 32k	V16 64k	V32 128k		Stratix IV-230
ALMs	1,286	-	-	-	11,053	21,211	45,138	-	91,200
DSPs	4	-	-	-	12	22	43	-	161
M9Ks	18	-	-	-	43	78	145	-	1,235

	Nios II/f	V1 4k	V2 8k	V4 16k	V8 32k	V16 64k	V32 128k	Cyclone IV-115
LEs	2,898	4,467	6,958	11,927	22,118	45,035	89,436	114,480
DSPs	4	12	24	48	96	192	388	532
M9Ks	21	11	16	25	42	76	144	432

© 2012 VectorBlox Computing Inc.

Average Speedup vs. Area

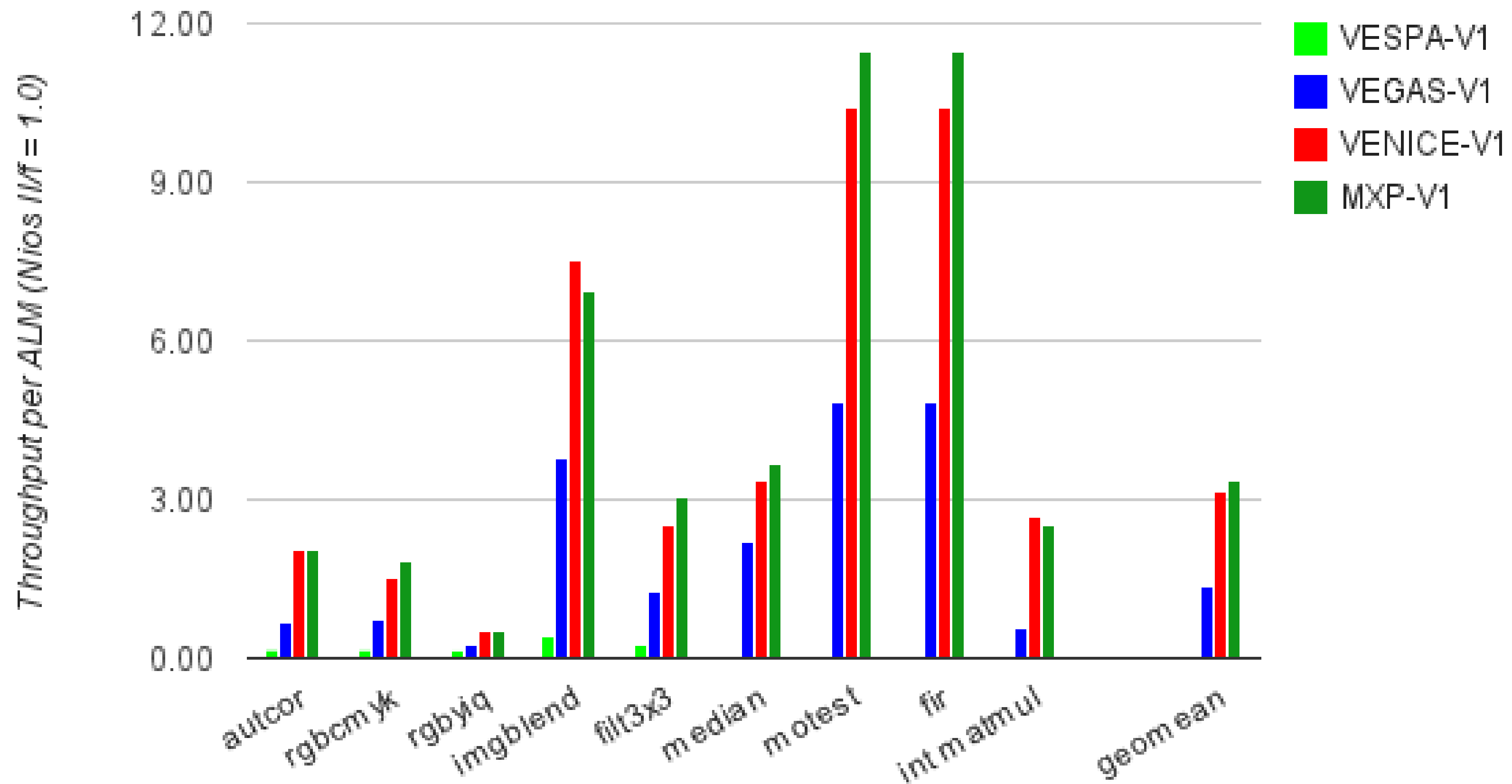
(Relative to Nios II/f = 1.0)



© 2012 VectorBlox Computing Inc.

MXP Performance Density

(Performance per ALM, relative to Nios II/f = 1.0)



© 2012 VectorBlox Computing Inc.

Benchmark Characteristics

Benchmark	Data Type		Benchmark Properties		
	In/Out	Intermed.	Data Set Size	Taps	Origin
autocor	halfword	word	1024	16	EEMBC
rgbcmyk	byte		896 606		EEMBC
rgbyiq	byte	word	896 606		EEMBC
imgblend	halfword		320 240		VIRAM
filt3x3	byte	halfword	320 240	3 3	VIRAM
median	byte		128 21	5 5	custom
motest	byte		32 32	16 16	custom
fir	halfword		4096	16	custom
matmul	word		1024 1024		custom

© 2012 VectorBlox Computing Inc.

Application Performance

Comparison to Intel i7-2600

(running on one 3.4GHz core, without SSE/AVX instructions)

CPU	Fir	2Dfir	Life	Imgblend	Median	Motion Estimation	Matrix Multiply
Intel i7-2600	0.05s	0.36s	0.13s	0.09s	9.86s	0.25s	50.0s
MXP	0.05s	0.43s	0.19s	0.50s	2.50s	0.21s	15.8s
Speedup	1.0x	0.8x	0.7x	0.2x	3.9x	1.7x	3.2x

© 2012 VectorBlox Computing Inc.

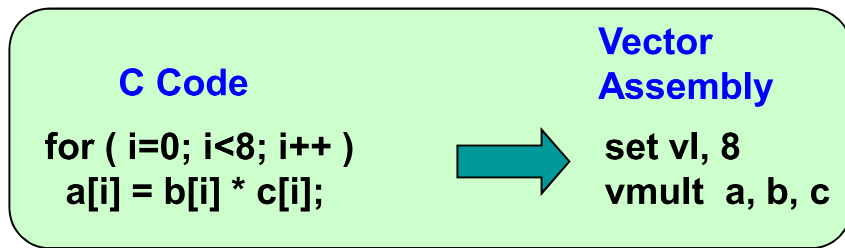
SOFTWARE DESIGN FOR MXP™

VectorBlox
embedded supercomputing

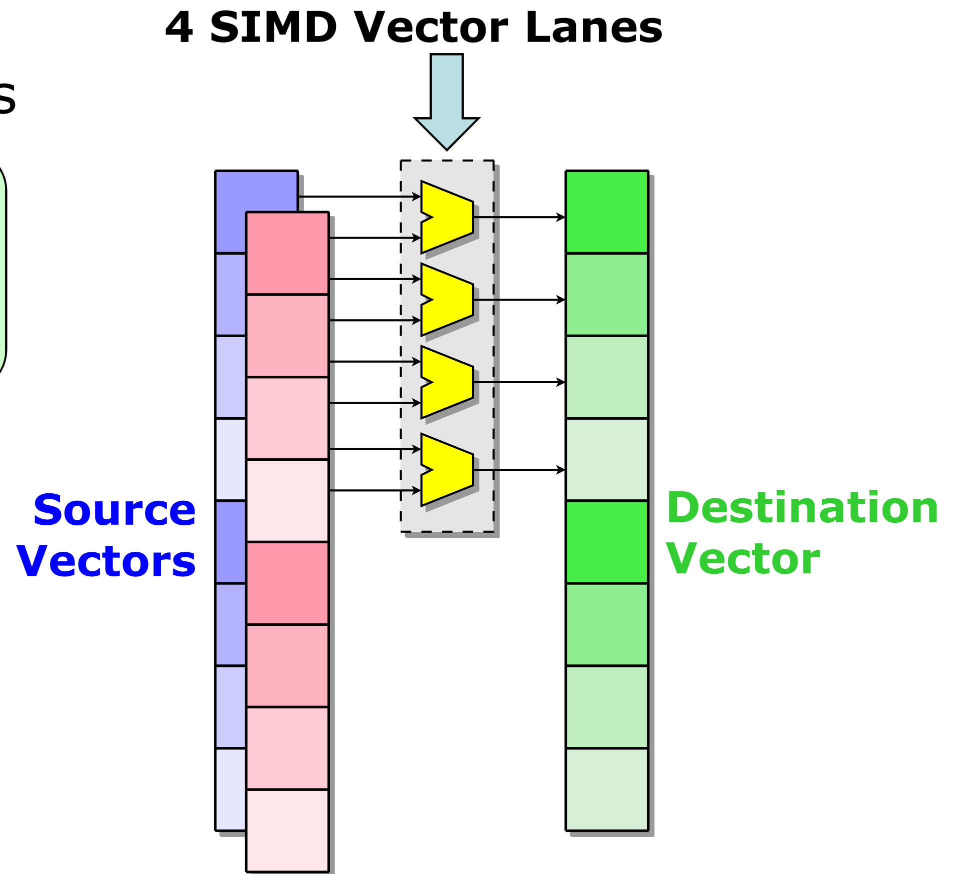
© 2012 VectorBlox Computing Inc.

Vector Processing

- **Data-level parallelism**
- Organize data as long vectors



- Vector instruction execution
  - Multiple vector lanes (SIMD)
  - Hardware automatically repeats SIMD operation over entire length of vector



© 2012 VectorBlox Computing Inc.

Performance Ideals

- Long, contiguous vectors for performance
  - Longer vectors = better performance
- Apply same operation(s) to entire vector
  - Skipping elements possible, but slower
  - Conditionals possible, but slower
- Integer or fixed-point only
  - Bytes, words, or halfwords (16b)
  - Signed or unsigned
  - No floating-point, no divide, no modulo

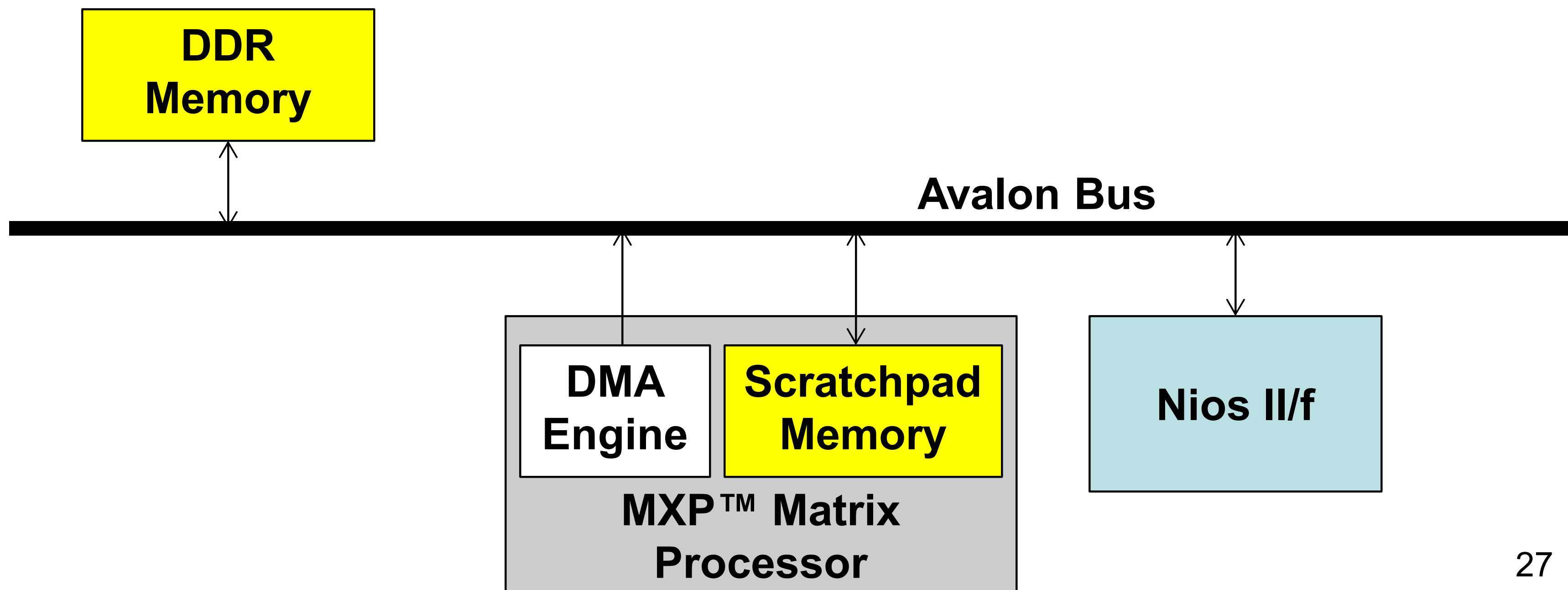
© 2012 VectorBlox Computing Inc.

Execution and Memory Model

- Nios processor runs C code
  - Memory-mapped address ranges

    eg 0x0000 0000 – 0x00FF FFFF 16 MB off-chip DDR

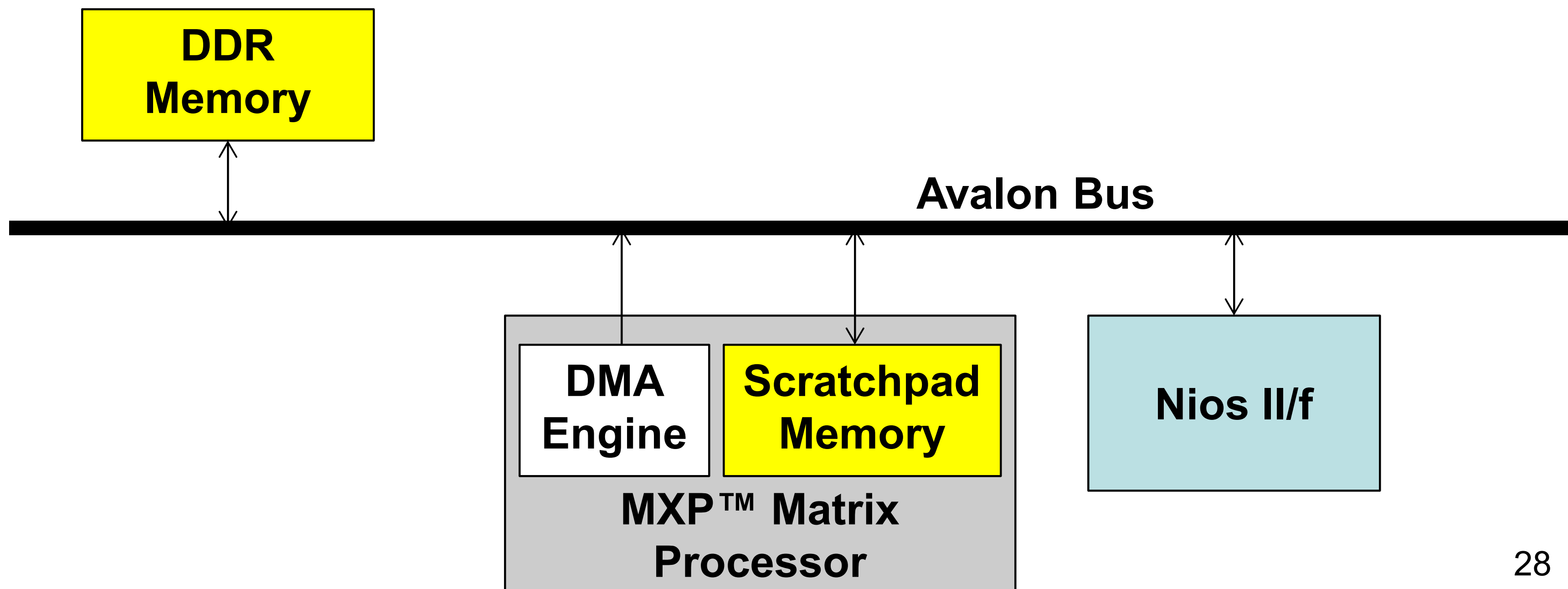
    0x0100 0000 – 0x0100 FFFF 64 kB scratchpad in MXP™



© 2012 VectorBlox Computing Inc.

Execution and Memory Model

- Nios processor runs C code
  - Memory-mapped address ranges
  - DMA Engine copies data between DDR and Scratchpad



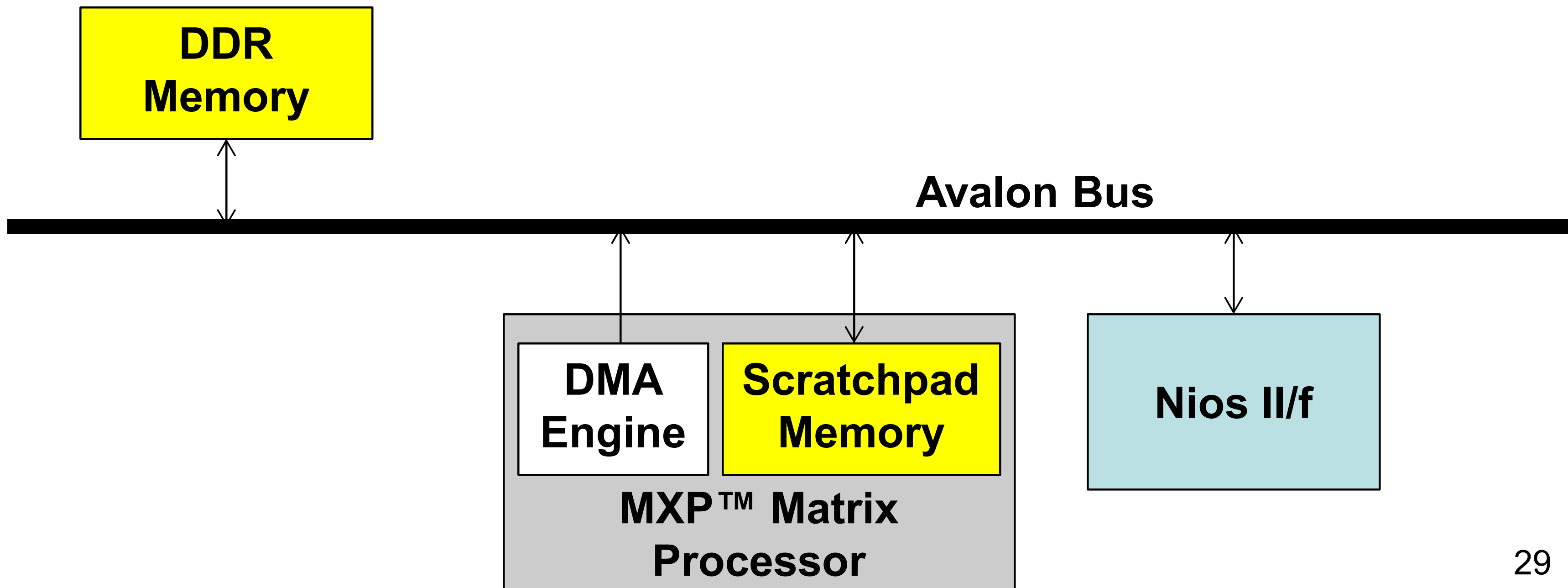
© 2012 VectorBlox Computing Inc.

Execution and Memory Model

- Nios processor runs C code
  - vbx() macros used invoke vector operations, eg

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

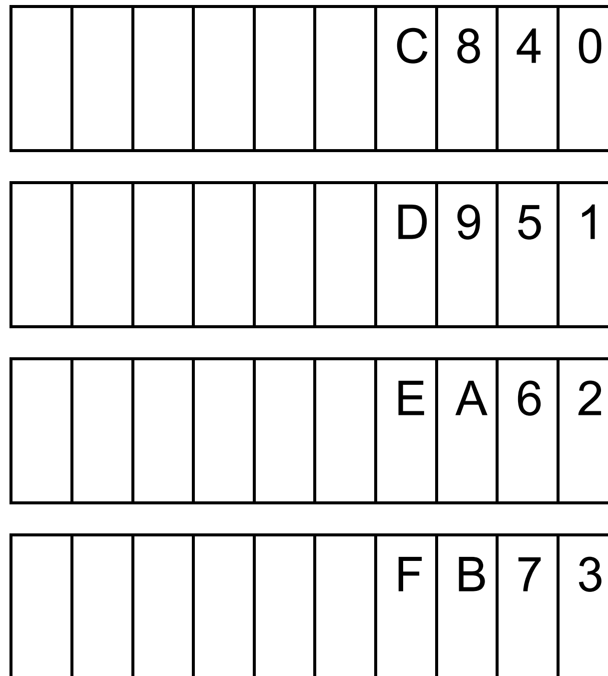
    vdst, **vsrc1**, **vsrc2** must be pointers into scratchpad



© 2012 VectorBlox Computing Inc.

Scratchpad Memory

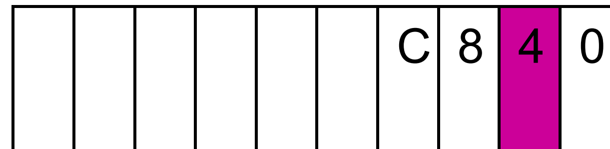
- Multi-banked, parallel access
  - One bank per lane
  - Addresses striped across banks, like RAID disks

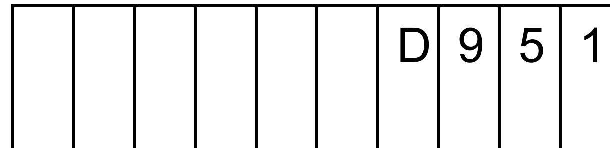


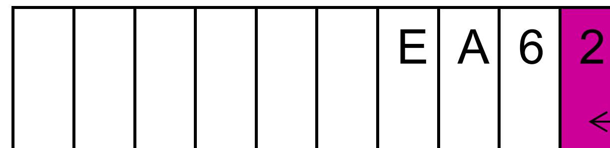
© 2012 VectorBlox Computing Inc.

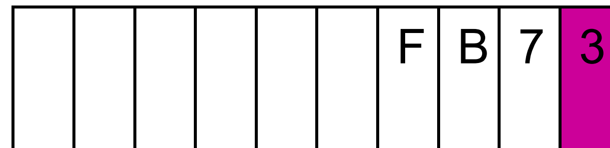
Scratchpad Memory

- Multi-banked, parallel access
  - Vector can start at any location





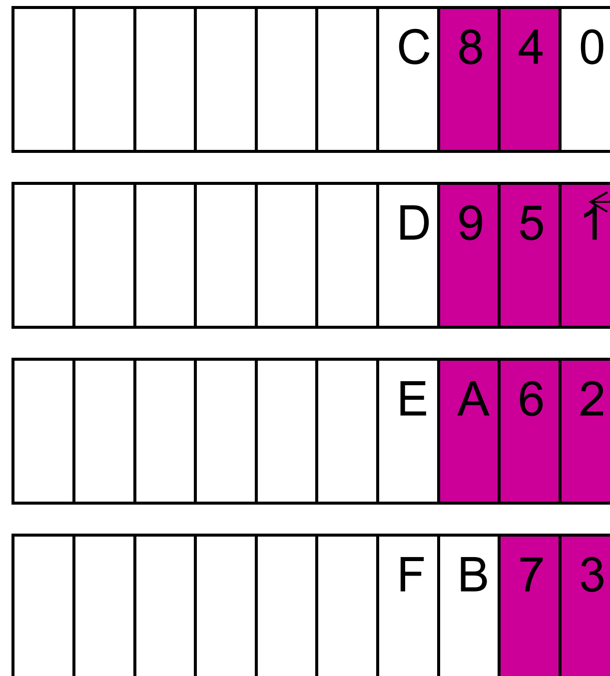




© 2012 VectorBlox Computing Inc.

Scratchpad Memory

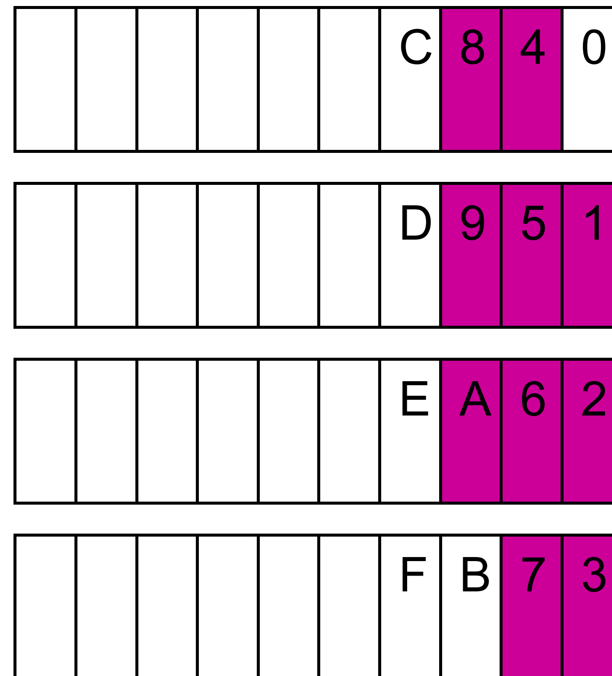
- Multi-banked, parallel access
  - Vector can start at any location
  - Vector can have any length

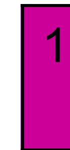


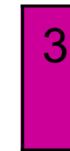
© 2012 VectorBlox Computing Inc.

Scratchpad Memory

- Multi-banked, parallel access
  - Vector can start at any location
  - Vector can have any length
  - One "wave" of elements can be read every cycle







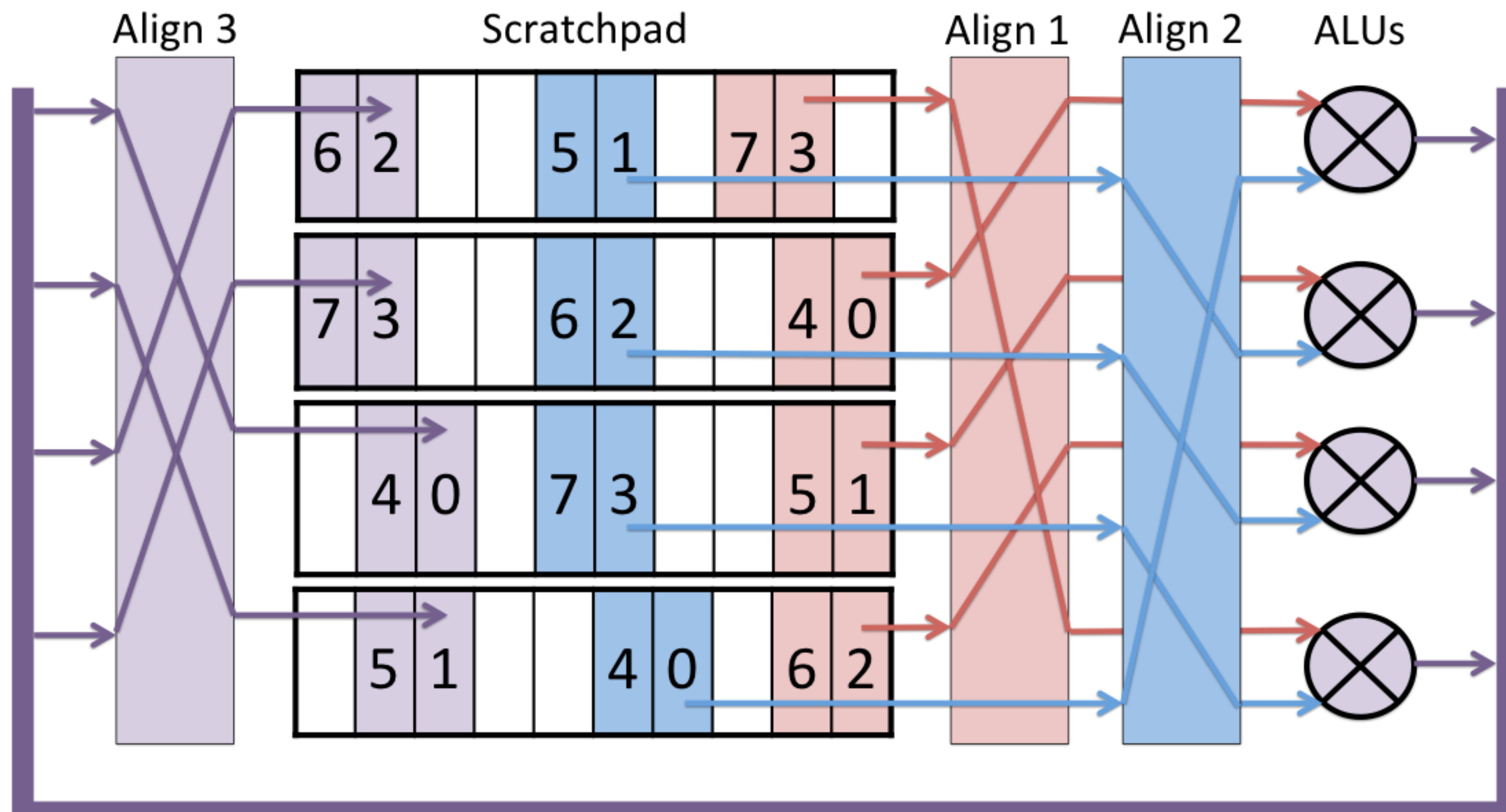
One-cycle parallel access to one full "wave" of vector elements

© 2012 VectorBlox Computing Inc.

Scratchpad-based Computing

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

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

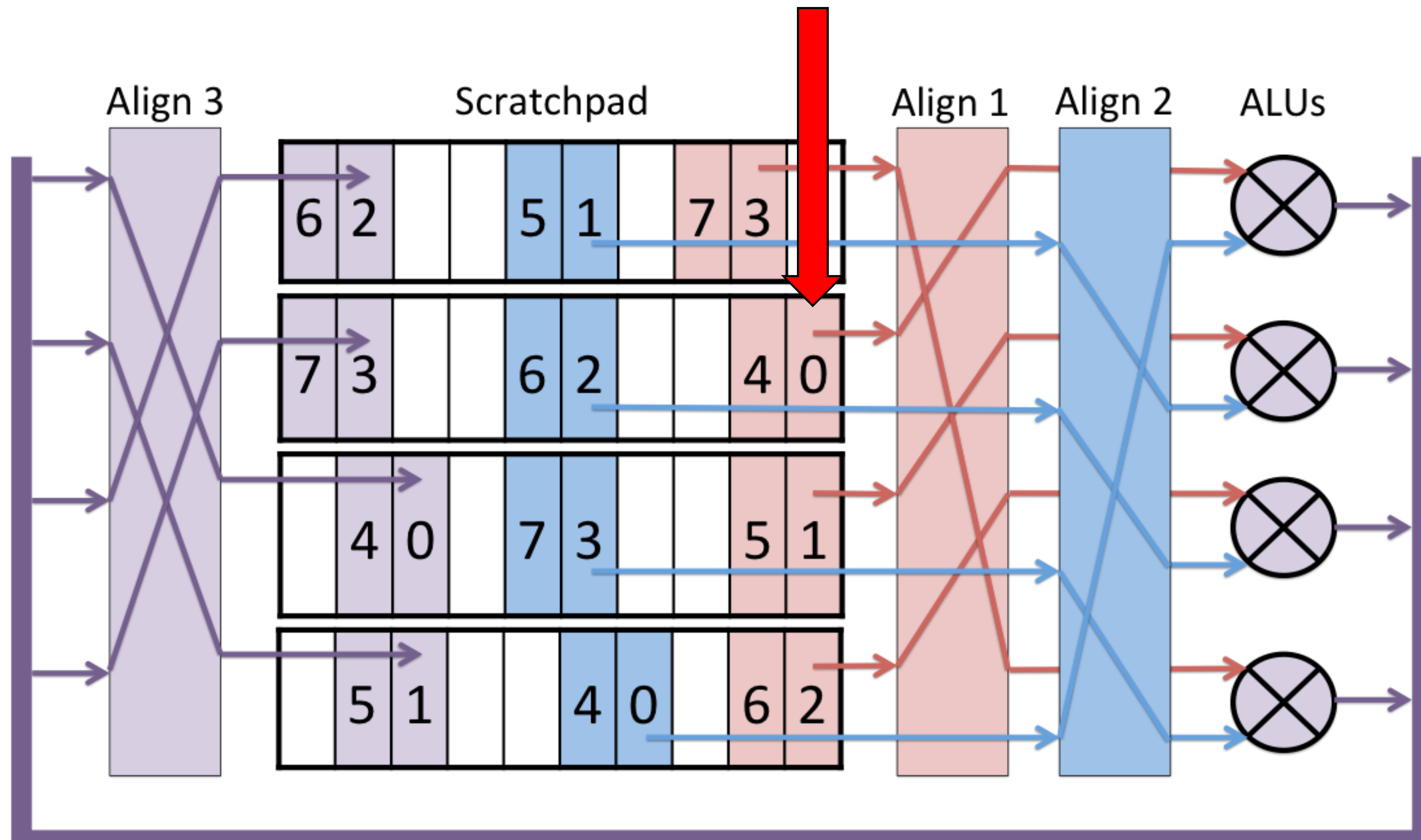


© 2012 VectorBlox Computing Inc.

Scratchpad-based Computing

vbx_word_t *vdst, *vsrc1, *vsrc2;

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

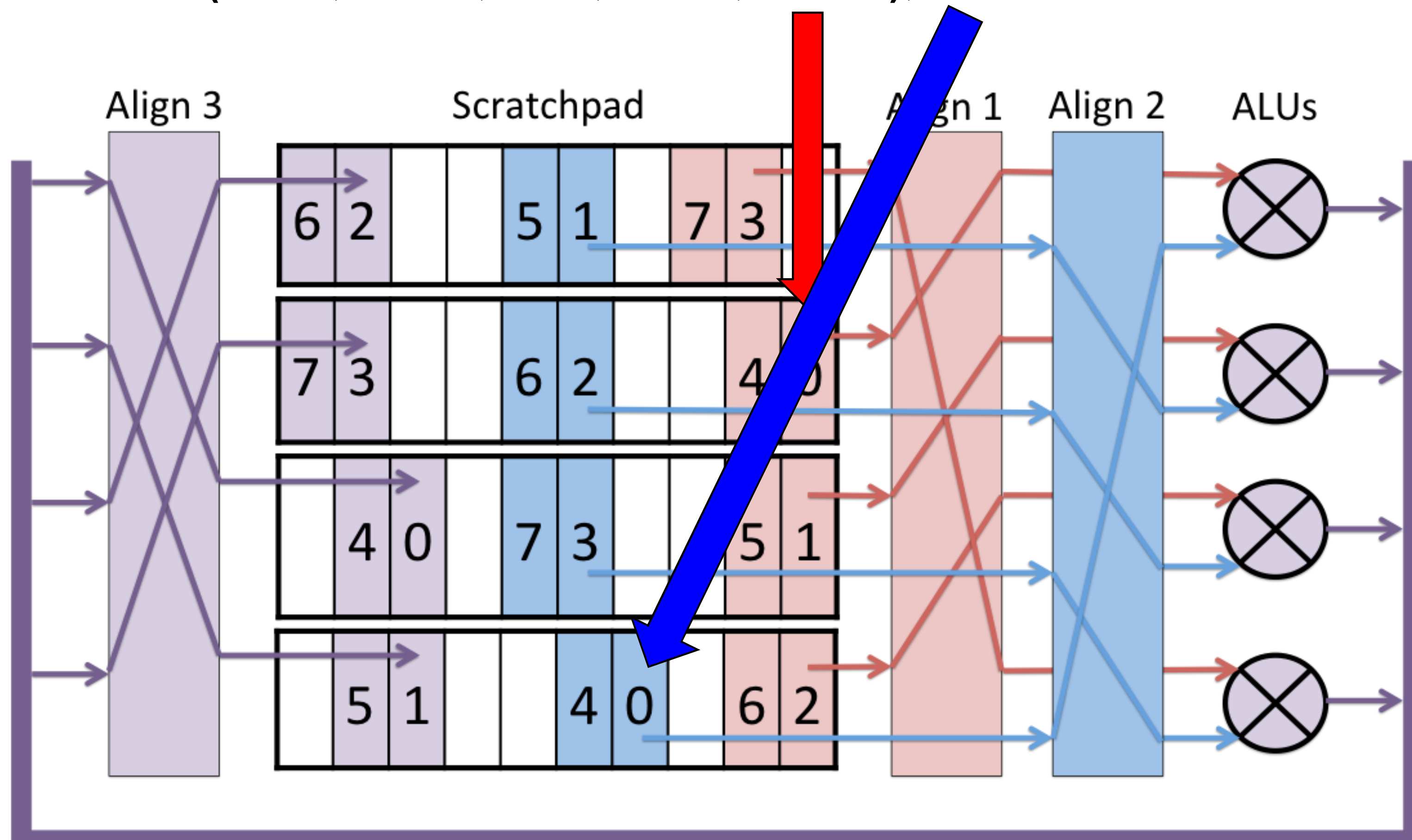


© 2012 VectorBlox Computing Inc.

Scratchpad-based Computing

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

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

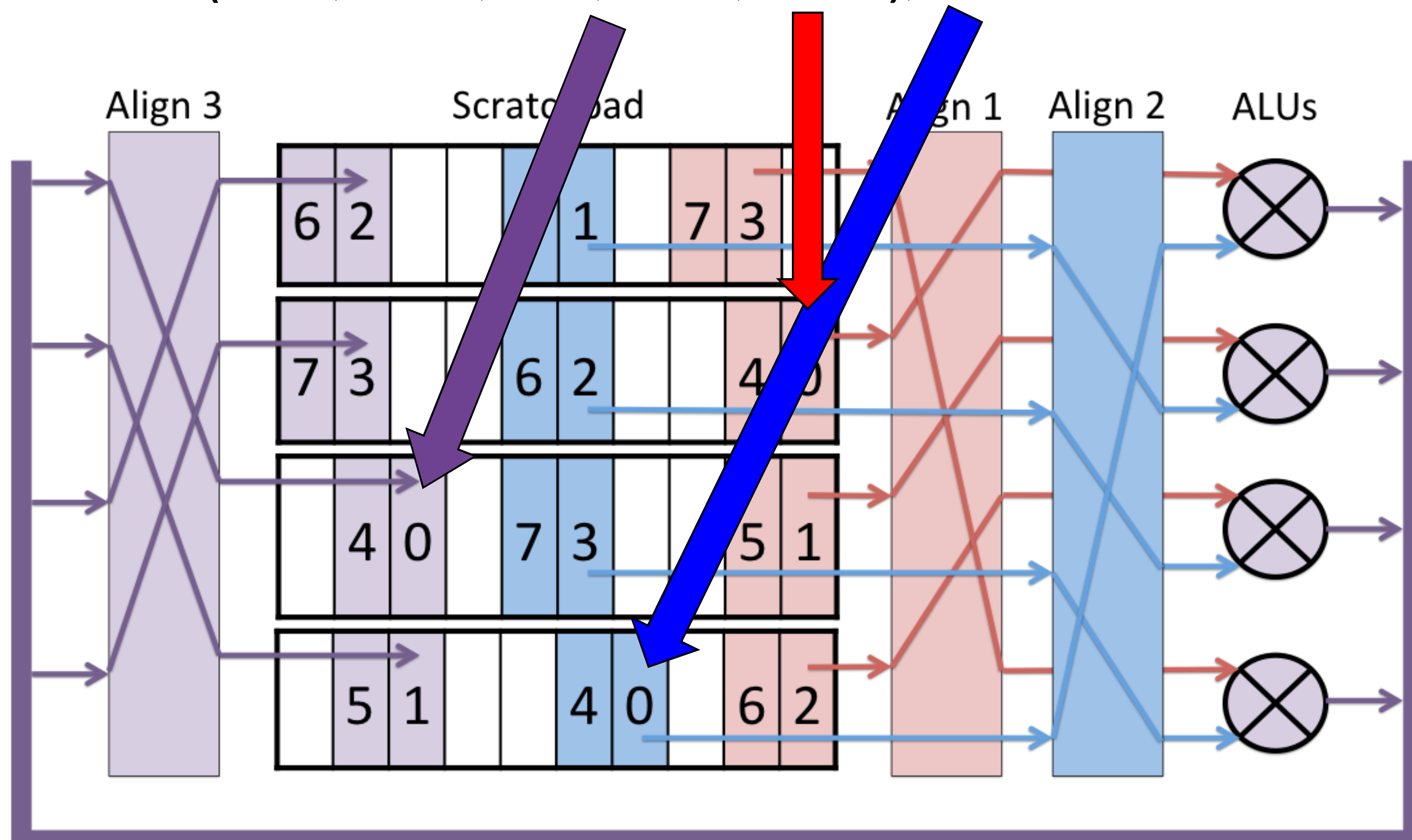


© 2012 VectorBlox Computing Inc.

Scratchpad-based Computing

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

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



© 2012 VectorBlox Computing Inc.

Software Examples

© 2012 VectorBlox Computing Inc.

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

© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- **Allocate vectors in scratchpad**

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                  // 128 words, in scratchpad
```

- **Move data from memory → scratchpad**

```
vbx_dcache_flush( data, 128*4 );                // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );        // copy from 'data'
```

- **Set vector length register**

```
vbx_set_vl( 128 );                              // # of elements
```

- **Perform vector operation**

```
vbx( SVW, VMULLO, vdata, multiplier, vdata ); // only 1 instruction
```

- **Move data from scratchpad → memory**

```
vbx_dma_to_host( data, vdata, 128*4 );          // copy results back
vbx_sync();                                     // wait vector/DMA to finish
```


© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```


© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad
- Move data from memory → scratchpad
- Set vector length register
- Perform vector operation
- Move data from scratchpad → memory

© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                // 128 words, in scratchpad
```

- Move data from memory → scratchpad
- Set vector length register
- Perform vector operation
- Move data from scratchpad → memory

© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );              // 128 words, in scratchpad
```

- Move data from memory → scratchpad

```
vbx_dcache_flush( data, 128*4 );             // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );     // copy from 'data'
```

- Set vector length register
- Perform vector operation
- Move data from scratchpad → memory

© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                // 128 words, in scratchpad
```

- Move data from memory → scratchpad

```
vbx_dcache_flush( data, 128*4 );               // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );       // copy from 'data'
```

- **Set vector length register**

```
vbx_set_vl( 128 );                             // # of elements
```

- Perform vector operation
- Move data from scratchpad → memory

© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                    // 128 words, in scratchpad
```

- Move data from memory → scratchpad

```
vbx_dcache_flush( data, 128*4 );                  // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );          // copy from 'data'
```

- Set vector length register

```
vbx_set_vl( 128 );                                // # of elements
```

- **Perform vector operation**

```
vbx( SVW, VMULLO, vdata, multiplier, vdata ); // only 1 instruction
```

- Move data from scratchpad → memory

© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                 // 128 words, in scratchpad
```

- Move data from memory → scratchpad

```
vbx_dcache_flush( data, 128*4 );               // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );       // copy from 'data'
```

- Set vector length register

```
vbx_set_vl( 128 );                             // # of elements
```

- Perform vector operation

```
vbx( SVW, VMULLO, vdata, multiplier, vdata ); // only 1 instruction
```

- Move data from scratchpad → memory

```
vbx_dma_to_host( data, vdata, 128*4 );         // copy results back
vbx_sync();                                    // wait vector/DMA to finish
```


© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- Allocate vectors in scratchpad

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                // 128 words, in scratchpad
```

- Move data from memory → scratchpad

```
vbx_dcache_flush( data, 128*4 );               // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );       // copy from 'data'
```

- Set vector length register

```
vbx_set_vl( 128 );                             // # of elements
```

- Perform vector operation

```
vbx( SVW, VMULLO, vdata, multiplier, vdata ); // only 1 instruction
```

- Move data from scratchpad → memory

```
vbx_dma_to_host( data, vdata, 128*4 );         // copy results back
vbx_sync();                                    // wait vector/DMA to finish
```


© 2012 VectorBlox Computing Inc.

Example: Vector * Constant

```
int data[128] = { 0, 1, 2, 3, 4, 5, ... , 127 };
int multiplier = 3;
```

- **Allocate vectors in scratchpad**

```
vbx_word_t *vdata;
vdata = vbx_sp_malloc( 128*4 );                // 128 words, in scratchpad
```

- **Move data from memory → scratchpad**

```
vbx_dcache_flush( data, 128*4 );              // remove data from cache
vbx_dma_to_vector( vdata, data, 128*4 );      // copy from 'data'
```

- **Set vector length register**

```
vbx_set_vl( 128 );                            // # of elements
```

- **Perform vector operation**

```
vbx( SVW, VMULLO, vdata, multiplier, vdata ); // only 1 instruction
```

- **Move data from scratchpad → memory**

```
vbx_dma_to_host( data, vdata, 128*4 );        // copy results back
vbx_sync();                                   // wait vector/DMA to finish
```


© 2012 VectorBlox Computing Inc.

```
#include "vbx.h"

int main()
{
    int A[] = {1, 2, 3, 4};
    int B[] = {5, 6, 7, 8};
    int C[4];

    int vector_len = 4;
    int num_bytes = vector_len * sizeof(int);

    /* step 1 */
    vbx_word_t *va = vbx_sp_malloc( num_bytes );
    vbx_word_t *vb = vbx_sp_malloc( num_bytes );
    vbx_word_t *vc = vbx_sp_malloc( num_bytes );

    /* step 2 */
    vbx_dma_to_vector( va, A, num_bytes );
    vbx_dma_to_vector( vb, B, num_bytes );

    /* step 3 */
    vbx_set_vl( vector_len );
    vbx( VVW, VADD, vc, va, vb );

    /* step 4 */
    vbx_dma_to_host( C, vc, num_bytes );

    /* step 5 */
    vbx_sp_free();

    printf( "C[] = %d, %d, %d, %d\n", C[0], C[1], C[2], C[3] );
    return 0;
}
```

VectorBlox MXP Starter Code for Adding Two Vectors

© 2012 VectorBlox Computing Inc.

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();

    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();
}
```

© 2012 VectorBlox Computing Inc.

Example: Transposed FIR

```
// transposed FIR filter
void vector_fir( input_type *input, output_type *output, input_type *coeffs, int sample_size, int num_taps )
{

   //5 vectors size chunk_size+scratchpad+padding, round to 1/8th vector memory
   int chunk_size    = (VECTOR_MEMORY_SIZE >> 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();
}
```

© 2012 VectorBlox Computing Inc.

Notes 1

- Scratchpad “best practices”
  - Allocate using vbx_sp_malloc()
  - Use like a stack, not heap
    - Want to avoid internal fragmentation
  - Use vbx_sp_free() to reset entire scratchpad
  - DMA using unaligned locations in scratchpad can be slower
    - Use vbx_sp_malloc_align() if critical

© 2012 VectorBlox Computing Inc.

Basic Vector Instructions

- All vector instructions use same format

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


© 2012 VectorBlox Computing Inc.

Basic 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

© 2012 VectorBlox Computing Inc.

Basic 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

© 2012 VectorBlox Computing Inc.

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

© 2012 VectorBlox Computing Inc.

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, VROTL, VROTR
  - Arithmetic: VADD, VSUB, VADDC, VSUBB, VABSDIFF, VMUL, VMULLO, VMULHI, VMULFXP
  - Movement: VMOV
  - Conditional move: VCMV_LEZ, VCMV_GTZ, VCMV_LTZ, VCMV_GEZ, VCMV_Z, VCMV_NZ

© 2012 VectorBlox Computing Inc.

Notes 2

- Matrices and submatrices
  - 2D matrices use standard C packed format
    - Row-major order
    - One row*col chunk, no row pointers / ragged rows
  - 3D matrices
    - One 2D matrix packed after another
    - One mat*row*col chunk
  - MXP can access sub-matrices
    - Can use vector length < row length
    - Specify row stride (> vector length), number of rows

© 2012 VectorBlox Computing Inc.

Example Code: 2D Matrix

2D Matrix VBX code:

```
vbx_half_t *vdest, *vsrc1, *vsrc2;
vbx_set_vl( vl );
vbx_set_2D( numRows, iD2, iA2, iB2 );
vbx_2D( VVH, VADD, vdest, vsrc1, vsrc2 );
```

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*)vsrc1 + (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.

© 2012 VectorBlox Computing Inc.

Example Code: 3D Matrix

3D Matrix VBX code:

```
vbx_half_t *vdest, *vsrc1, *vsrc2;
vbx_set_vl( vl );
vbx_set_2D( numRows, iD2, iA2, iB2 );
vbx_set_3D( numMats, iD3, iA3, iB3 );
vbx_3D( VVH, VADD, vdest, vsrc1, 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*)vsrc1 + (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.

© 2012 VectorBlox Computing Inc.

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 )
```

- Equivalent (non-vector) C code

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

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


© 2012 VectorBlox Computing Inc.

Conclusions

- Vector processing
  - Easy to use/deploy
  - Easy to program FPGAs
  - Scalable performance (area vs speed)
    - Speedups > 200x
  - No hardware recompiling necessary
    - Rapid algorithm development
    - Hardware purely ‘sandboxed’ from algorithm

© 2012 VectorBlox Computing Inc.