NC STATE UNIVERSITY

SIMD AND DSP WITH ARM CORTEX-M

Alexander G. Dean - agdean@ncsu.edu

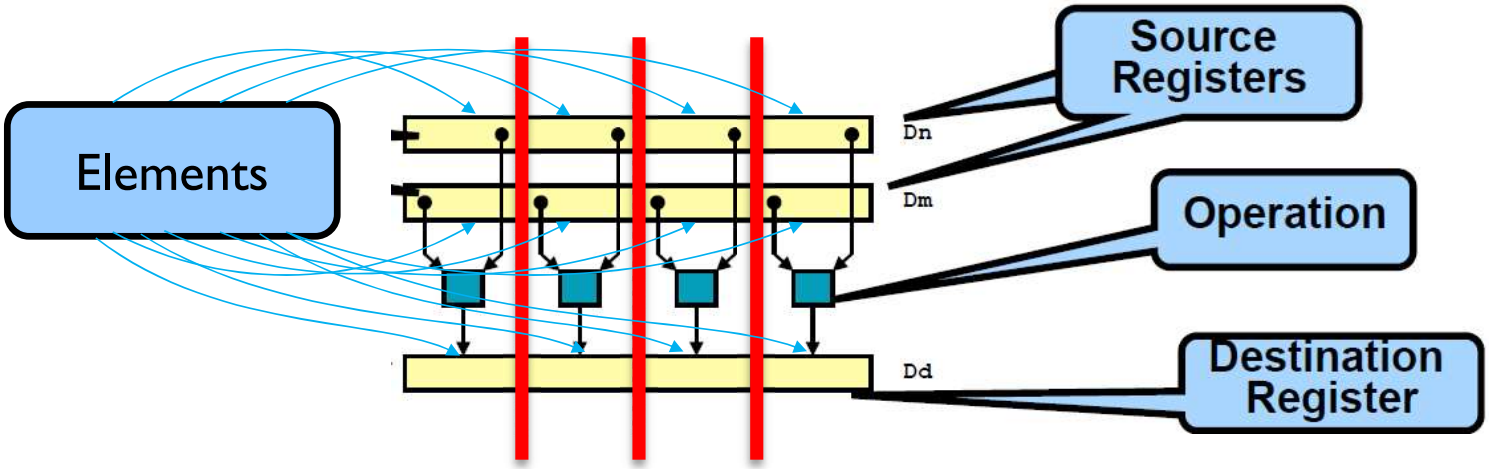
Instruction Set Enhancements for SIMD and *DSP*

- ARMv7-M
  - Cortex-M3 adds…
    - Hardware divide instruction
    - Conditional execution using IT instruction
    - *MAC (Multiply-Accumulate) instruction, multi-cycle*
  - Cortex-M4 adds…
    - *Saturating arithmetic*
    - *DSP instructions (more later)*
    - *More MAC instructions, single-cycle*
    - 32-bit SIMD processing – single instruction, multiple data (mini-vector)
  - Cortex-M4F adds…
    - Single-precision floating-point math unit
      - Instructions
      - 32 S registers (32-bits each)
  - Cortex-M7F adds…
    - Additional floating-point math instructions
    - Optional double-precision floating-point instructions
- ARMv8.1-M (e.g. Cortex-M55) adds…
  - Low-overhead branch instructions
  - Half-precision floating point
  - Helium (M-Profile Vector Extension)
    - 128-bit SIMD processing

Alexander G. Dean - agdean@ncsu.edu

SIMD PROCESSING

SIMD: Single Instruction Performed Simultaneously on Multiple Data items



- Data path in CPU is 32 bits wide
  - Registers, arithmetic/logic unit, memory interface
- Interpret and process those 32 bits as multiple elements of a vector
  - E.g. two 16-bit values, four 8-bit values packed into 32 bits
- Now a single instruction can operate on multiple elements
  - Up to 2x or 4x speed-up
- Available on Cortex-M4, M7, etc.

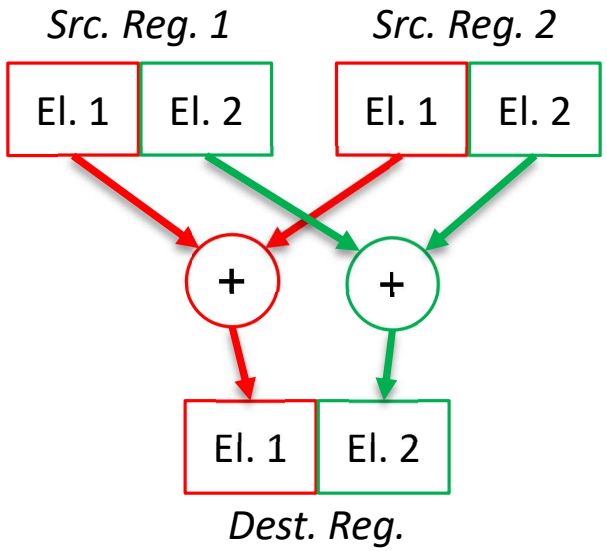
SIMD Data Types and Instructions Available

- 32-bit SIMD in ARMv7-M (M4,M7)
  - Data types and operation variants
    - Size: 8 bit (byte) or 16 bit (halfword)
    - Signed or unsigned (s, u)
    - Saturating (q): result does not overflow/underflow, but instead is clipped
    - Halving (h): result is divided by two, eliminating overflow possibility
  - Operations
    - Add, subtract, multiply, exchange, absolute value, accumulate, select
    - Not all operations are available for all data types
  - Full information in Armcc User Guide, Chapter 12 (SIMD Instruction Intrinsics)
- Helium: Advanced SIMD defined in ARMv8.1-M (e.g. Cortex-M55)
  - Very high performance 128-bit data path
  - Uses FP register file as 8 quadwords, 16 doublewords, or 32 words
  - Data types
    - 8, 16, 32 bits
    - Integer, optional float
  - Over 150 new instructions
    - Data processing
    - Data reformatting: Load and store support interleaving, gather load, scatter store operations

Preview: How to Use Them?

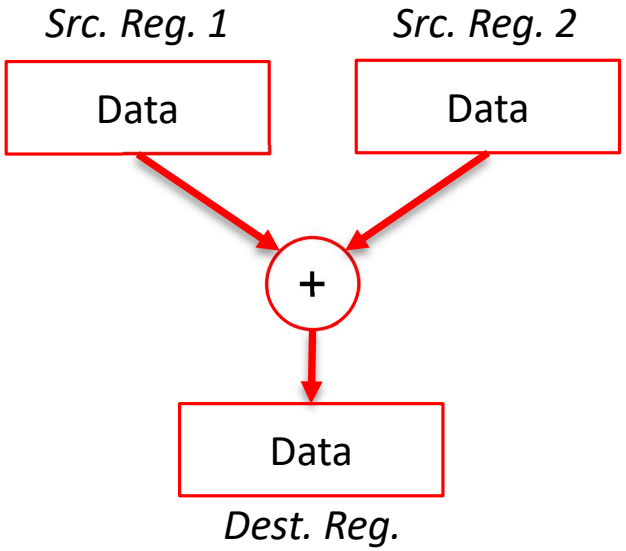
- Rely on compiler to vectorize code and generate SIMD instructions
- Write assembly code with SIMD instructions
- Write C code and use libraries which support SIMD
- Write C code with *compiler intrinsics* to specify SIMD operations

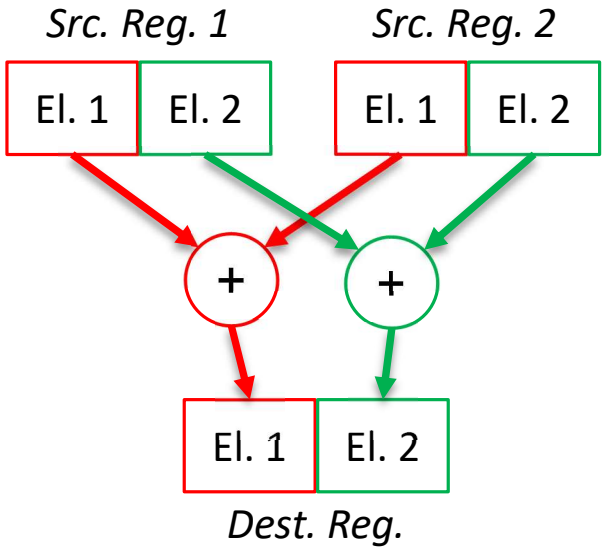
SIMD CONCEPTS



Alexander G. Dean - agdean@ncsu.edu

SIMD (Mini-Vector) Concepts

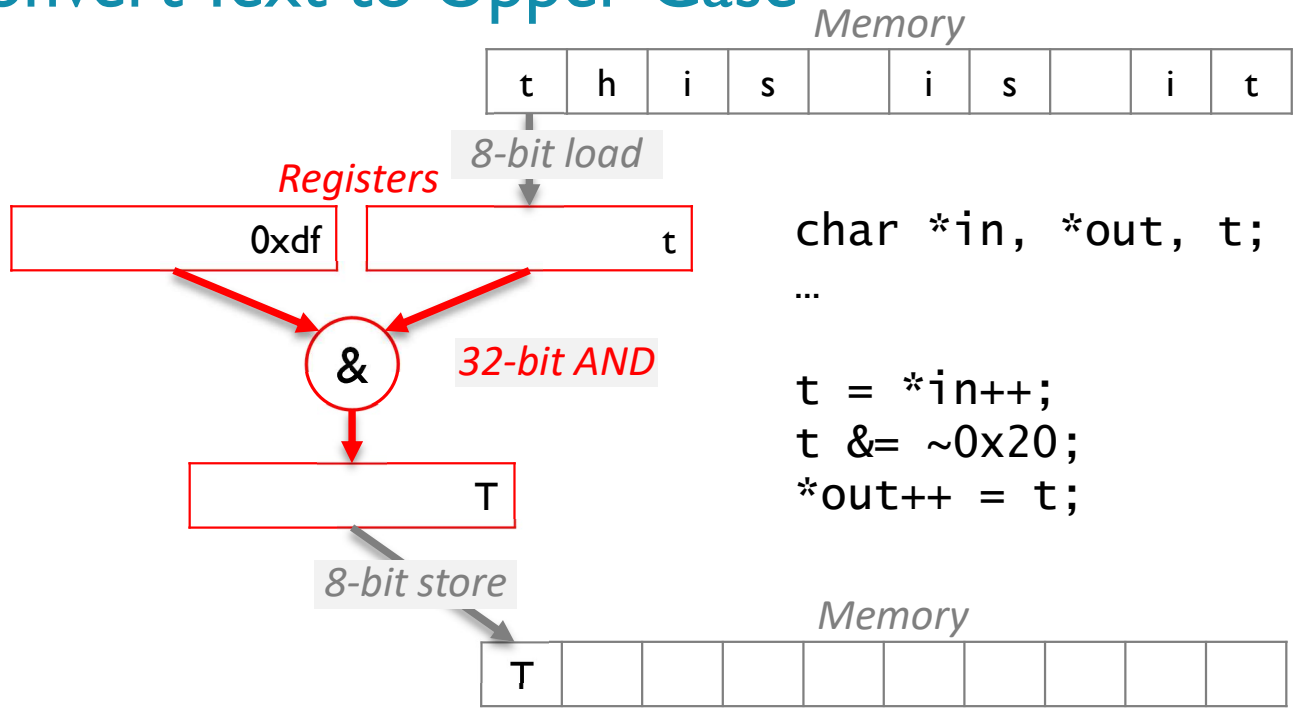




- Data path (registers, ALU, buses, etc.) is 32 bits wide
  - Can we pack multiple data items into a single 32-bit value?
- SIMD: Single Instruction is applied to Multiple Data values simultaneously
  - One register has multiple lanes, each holding a data value
  - 32 1-bit lanes, four 8-bit lanes, two 16-bit lanes?

Example Application: Convert Text to Upper Case

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
@	64	0100	0x40	`	96	0140	0x60
A	65	0101	0x41	a	97	0141	0x61
B	66	0102	0x42	b	98	0142	0x62
C	67	0103	0x43	c	99	0143	0x63
D	68	0104	0x44	d	100	0144	0x64
E	69	0105	0x45	e	101	0145	0x65
F	70	0106	0x46	f	102	0146	0x66
G	71	0107	0x47	g	103	0147	0x67
H	72	0110	0x48	h	104	0150	0x68
I	73	0111	0x49	i	105	0151	0x69
J	74	0112	0x4a	j	106	0152	0x6a
K	75	0113	0x4b	k	107	0153	0x6b
L	76	0114	0x4c	l	108	0154	0x6c
M	77	0115	0x4d	m	109	0155	0x6d



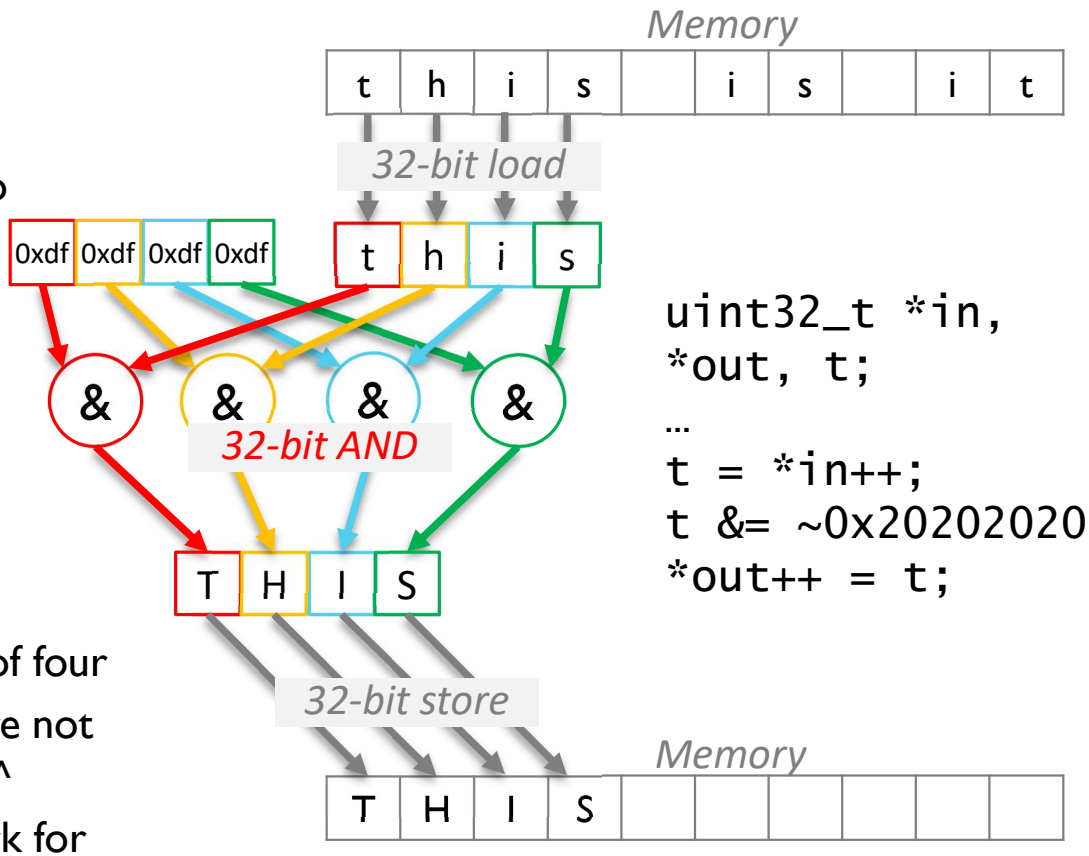

```
char *in, *out, t;
…

t = *in++;
t &= ~0x20;
*out++ = t;
```

- Text represented with 8-bit ASCII data
- Clear bit 6 to convert from lower to upper case
  - AND with ~0x20 (0xdf)
- Converts one character at a time
- Processor has 32-bit data path, which can hold four 8-bit lanes. Can we do better?

Alexander G. Dean - agdean@ncsu.edu

Example Application: Convert Text to Upper Case

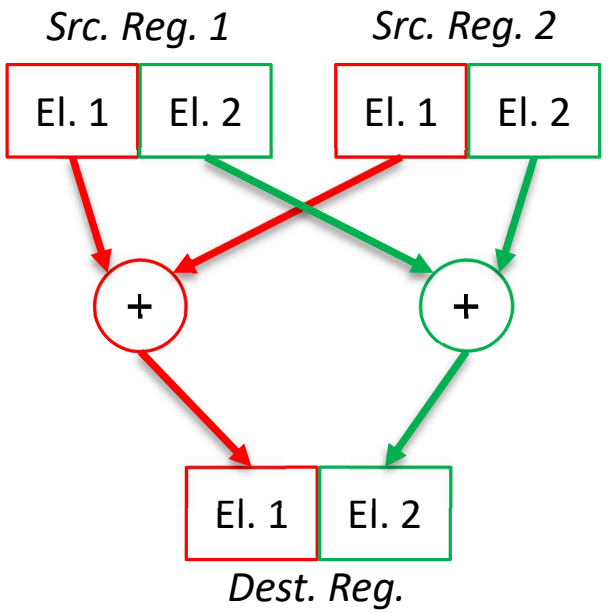
- Have processor interpret data four bytes at a time (as uint32_t)
  - Data in memory is arranged sequentially, no reorganization needed
  - Convert inputs, temps, outputs to 32 bits
  - Replicate constant ~0x20 across all lanes (~0x20202020)
  - Pointers will automatically be incremented by 4 instead of 1
- Restrictions
  - Assumes number of data items is multiple of four
  - Will also convert some symbols if inputs are not tested to be characters: {→[, }→], |→\,~→^
  - Is AND a special operation, or will this work for every operation?




```
uint32_t *in,
*out, t;
…
t = *in++;
t &= ~0x20202020;
*out++ = t;
```

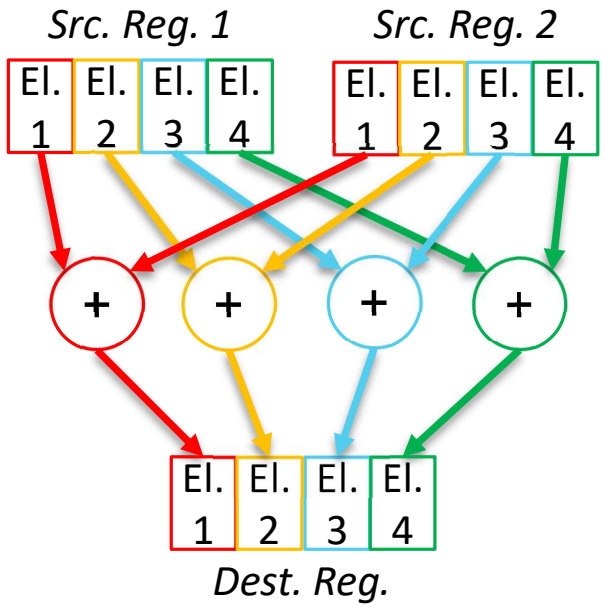
Generalization to Other Operations?

- This approach works if lanes are independent (one lane cannot affect another)
- Independent lanes give inherently SIMD operations
  - AND, NAND, OR, XOR, NOR, NOT
- Other instructions have dependence between lanes, preventing SIMD operations
  - Rotate, shift
  - Add (carry), subtract (borrow), multiply, divide
- Need special versions of these operations
- ARMv7-M provides some 32-bit SIMD instructions based on ADD, SUB, and MUL



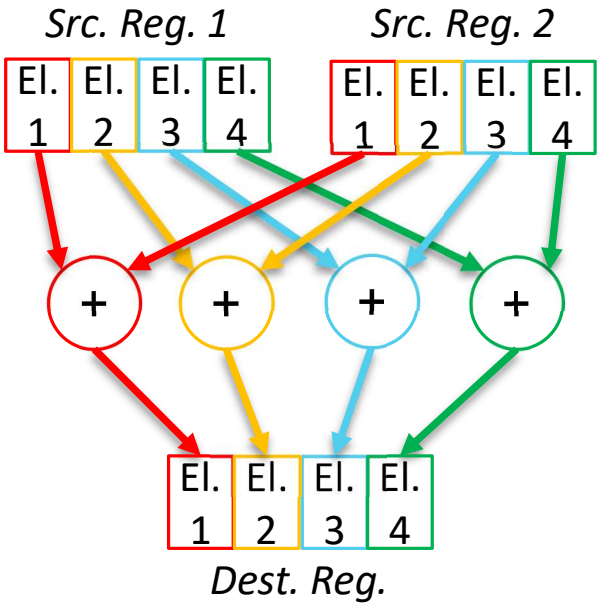
Alexander G. Dean - agdean@ncsu.edu

ARMV7-M SIMD AND DSP SUPPORT

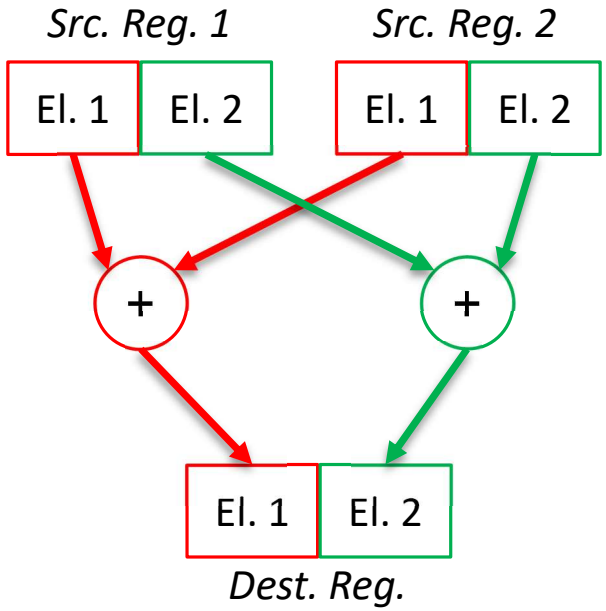


Alexander G. Dean - agdean@ncsu.edu

Data Sizes for 32-bit SIMD Instructions



- Four eight-bit lanes

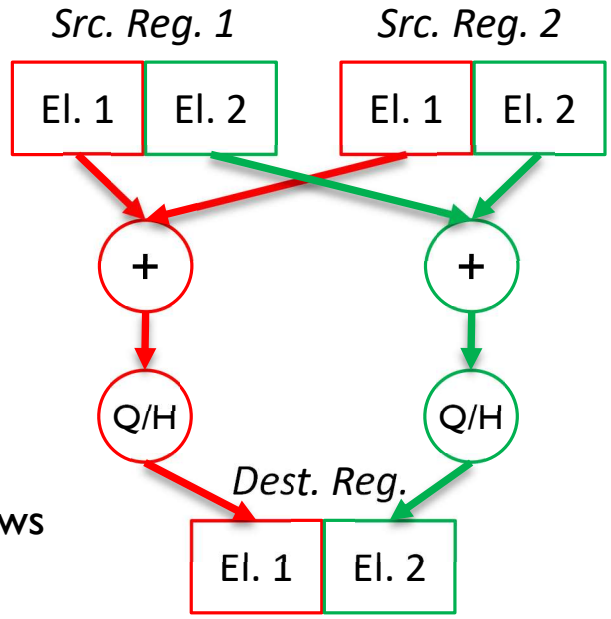


- Two sixteen-bit lanes

Alexander G. Dean - agdean@ncsu.edu

32-bit SIMD Arithmetic Instructions

- Basic Instructions
  - ADD[8|16]: Byte-wise or halfword-wise addition
  - SUB[8|16]: Byte-wise or halfword-wise subtraction
- Result status bits in program status register
  - Four bits GE[0-3], corresponding to each lane
  - SADD, SSUB: sets lane bit to 1 if lane result ≥ 0
  - UADD, USUB: sets lane bit to 1 if lane result overflows or underflows



- Prefixes
  - Signed (S): signed math, updates GE bits
  - Unsigned (U): unsigned math, updates CPSR GE bits
  - Saturating (Q): Limit value to closest valid value
  - Halving (H): Divide result by two

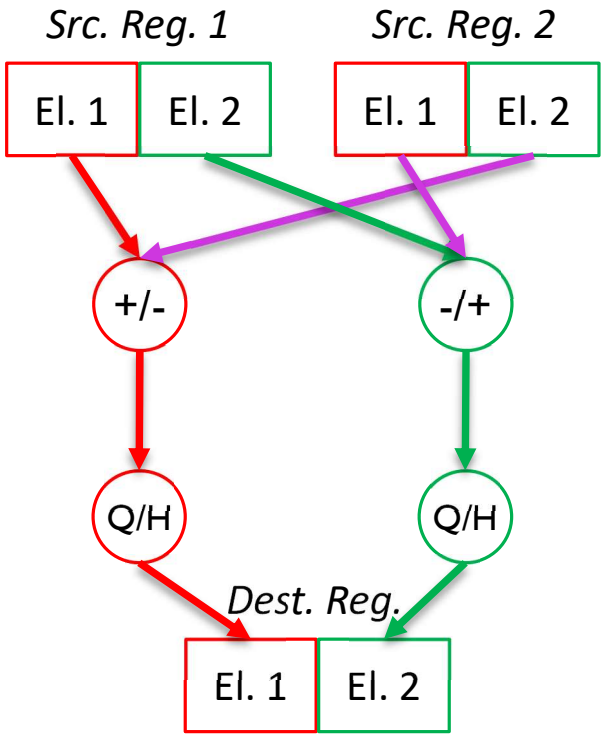
Prefixes for Parallel Instructions	
S	Signed arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results

Alexander G. Dean - agdean@ncsu.edu

32-bit SIMD Arithmetic Instructions

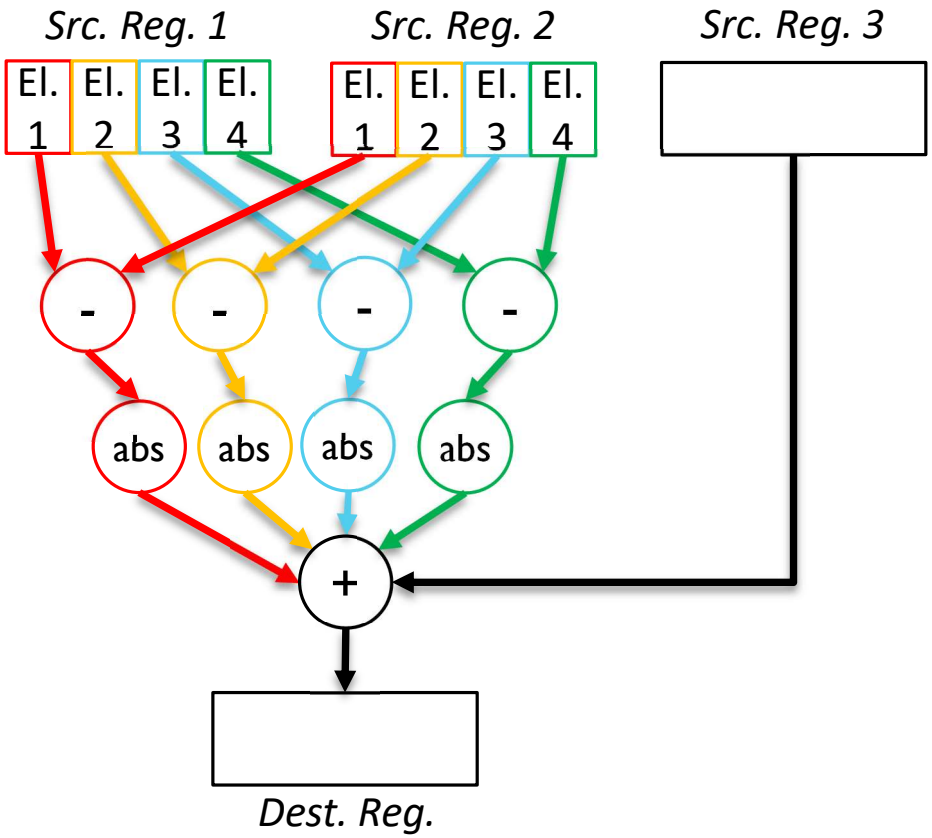
- More Instructions
  - ASX: Halfword-wise exchange, add, subtract
  - SAX: Halfword-wise exchange, subtract, add
- Prefixes
  - Saturating
  - Halving

Prefixes for Parallel Instructions	
S	Signed arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results



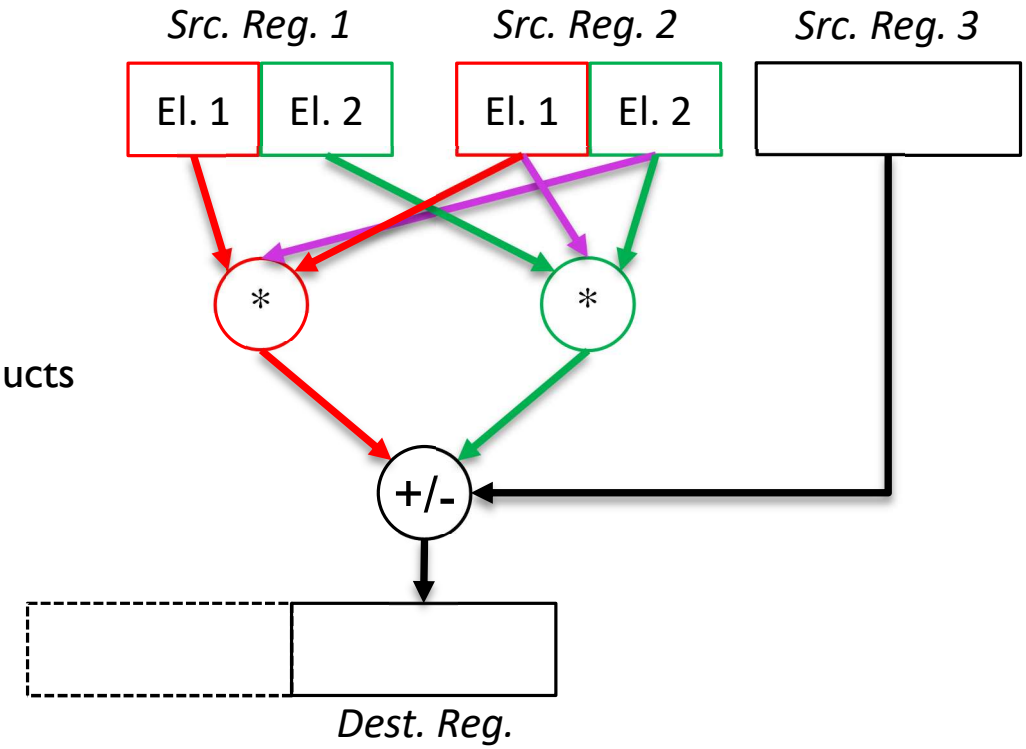
32-bit SIMD Arithmetic Instructions

- More Instructions
  - USAD8: Unsigned sum of absolute differences
  - USADA8: Unsigned sum of absolute differences and accumulate



32-bit SIMD Multiplication Instructions

- SM[U|L][A|S]{L}D{X}: Dual halfword signed multiply, add/subtract products
- [U|L] Options
  - U: SMU – Base operation: multiply halfwords, add/subtract products
  - L: SML – Accumulate sum (or difference) of products in 32-bit register
- [A|S] Options
  - A: Add products
  - S: Subtract products
- {L} Option
  - L: Accumulate to 64-bit register
- {X} Option
  - X: Exchange halfwords of one operand before multiplications

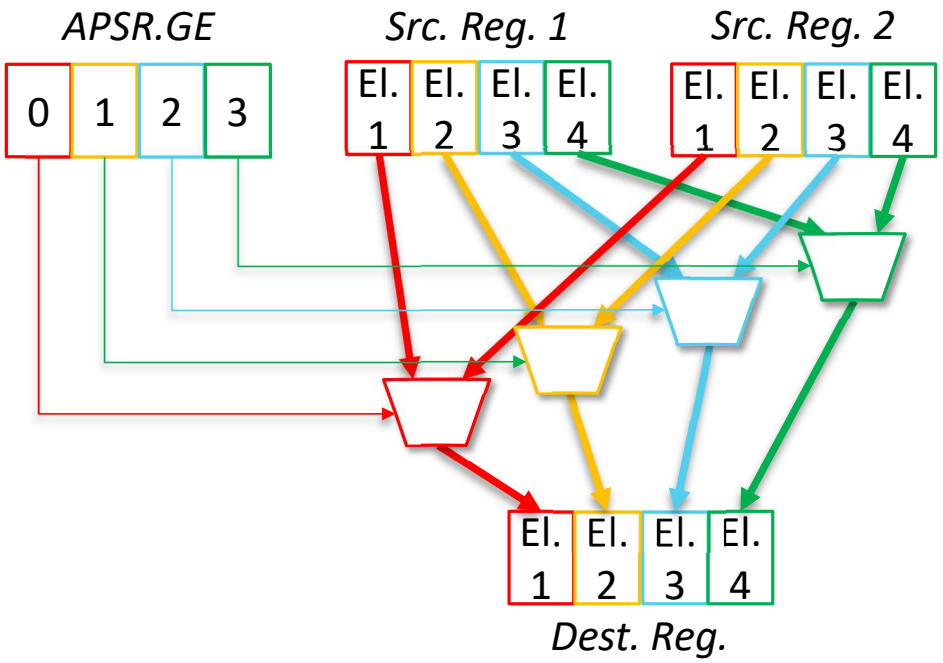


Alexander G. Dean - agdean@ncsu.edu

32-bit SIMD Miscellaneous Instructions

- Saturation
  - SSAT16: Saturate halfwords to range -2^{n-1} to $2^{n-1}-1$, with n as argument
  - USAT16: Saturate halfwords to range 0 to $2^{n}-1$, with n as argument
- Extraction with extension (and optional addition)
  - UXT{A}B16: extract low byte of each half-word, zero extend to 16 bits, optional add to first operand
  - SXT{A}B16: extract low byte of each half-word, sign extend to 16 bits, optional add to first operand
- Packing
  - PKHBT: pack halfword, bottom and left-shifted top (LSL)
  - PKHTB: pack halfword, top and right-shifted bottom (ASR)

32-bit SIMD Miscellaneous Instructions: Selection



- SEL: Select bytes based on GE (greater than or equal) flags
  - APSR GE flags updated by [U|S][ADD|SUB][8|16]

- Example 1:
  - SADD16 R0, R1, R2: Signed halfword add
  - SEL R3, R4, R5
    - R3[15:0] = R4[15:0] if SADD16 low-word is ≥ 0, else R5[15:0]
    - R3[31:16] = R4[31:16] if SADD16 high-word is ≥ 0, else R5[31:16]
- Example 2:
  - UADD8 R0, R1, R2: Signed byte add
  - SEL R3, R4, R5
    - R3[7:0] = R4[7:0] if UADD8 low byte result overflowed, else R5[7:0]
    - Similar for other bytes

References

- MDK Armcc User Guide: DUI0375
  - Chapter 12: ARMv6 SIMD Instruction Intrinsics
- ARM C Language Extensions (ACLE): IHI0053 (different syntax, not used for armcc v5)
  - 9.3: 16-bit multiplications
  - 9.4: Saturating intrinsics
  - 9.5: 32-bit SIMD intrinsics
  - 11: Instruction generation

Alexander G. Dean - agdean@ncsu.edu

USING THE SIMD INSTRUCTIONS

Alexander G. Dean - agdean@ncsu.edu

How Can We Use These SIMD Instructions?

- Write **C code, rely on the compiler** to generate SIMD instructions
  - Depends on compiler's ability to vectorize
  - "How can I get the compiler to do what I want?"
  - Sometimes manual provides *idioms* (code structures) which compiler can process more easily
- Write C code, call functions from **SIMD libraries**
  - SIMD-optimized libraries needed for your application, such as CMSIS-DSP
- Write **C code with compiler intrinsics** to specify SIMD instructions
  - Gives more control but handles many details
  - Need clear understanding of data layout and processing flow
- Write a separate **SIMD assembly code** module, link it with our C code
  - Provides full control but you must manage all the details
  - Need clear understanding of data layout and processing flow

Vectorizing the Code

- Definitions
  - **Scalar code**: operates on one set of operands at a time
  - **Vector code**: operates on multiple sets of operands at a time
  - **Vectorization**: converting code from scalar to vector form
- Vectorization is **main** compiler optimization enabling use of SIMD instructions
  - Others possible, but don't work on as much code, harder to implement in compiler
- Best to try to vectorize loops first
  - Innermost loops often dominate execution time
  - Arrangement of instructions and data make vectorization easier (than the general case, e.g. straight-line code)
- Vectorization of loops is built on loop unrolling

Loop Unrolling: Selecting Loops

- Select an inner-most loop
  - With data in arrays
  - Without
    - Subroutine calls
    - Conditional control flow
    - Data dependencies on recent iterations
- Determine loop unroll factor (and vector size) F
  - ARM registers are 32 bits wide, so options are:
    - 4 element vector of bytes
    - 2 element vector of half-words
  - Any loop carried dependencies must be > F iterations away

```
int sum_ints( int * x, int n)
      int i, sum_val = 0;

      for (i=0; i<n; i++) {
            sum_val += x[i];
      }
      return sum_val;
}
```


Alexander G. Dean - agdean@ncsu.edu

Loop Iteration Count

- Unrolling a loop with L iterations by a factor of F
  - **Unrolled loop** performs floor(L/F) iterations of the unrolled loop (performing F times as much work per iteration)
    - This unrolled loop will later be **vectorized**
  - **Clean-up loop** performs L modulo F remaining iterations of the original loop (performing 1x work per iteration)
- Compiler must generate code which operates correctly regardless of whether L is a multiple of F or not
  - Typically involves generating code to determine if there are at least F more iterations of work to perform
  - Can be simplified if compiler can determine if L is a multiple of F

Loop Unrolling and Vectorization Process

1. Create prelude
   1. Create vector values (and loop-independent variables) from scalars
2. Unroll loop body
   1. Modify loop control code
      1. Test: confirm at least F more iterations remain
      2. Increment: Scale update by factor of F
   2. Unroll loop by factor of vector size
      1. Modify data processing instructions
         1. Unrolling: Make F-1 copies of loop body instructions
         2. Vectorizing: replace F scalar instructions with one vector instruction
      2. Update references to data: Add 1 to F-1 to data value indices. May update pointers by factor of F.
3. Create postlude
   1. Reduce (gather, condense, sum) data from vector to scalar form
4. Clean-up
   1. Implement remaining iterations with non-vectorized code

Example Program with Loop Unrolling

Scalar Code

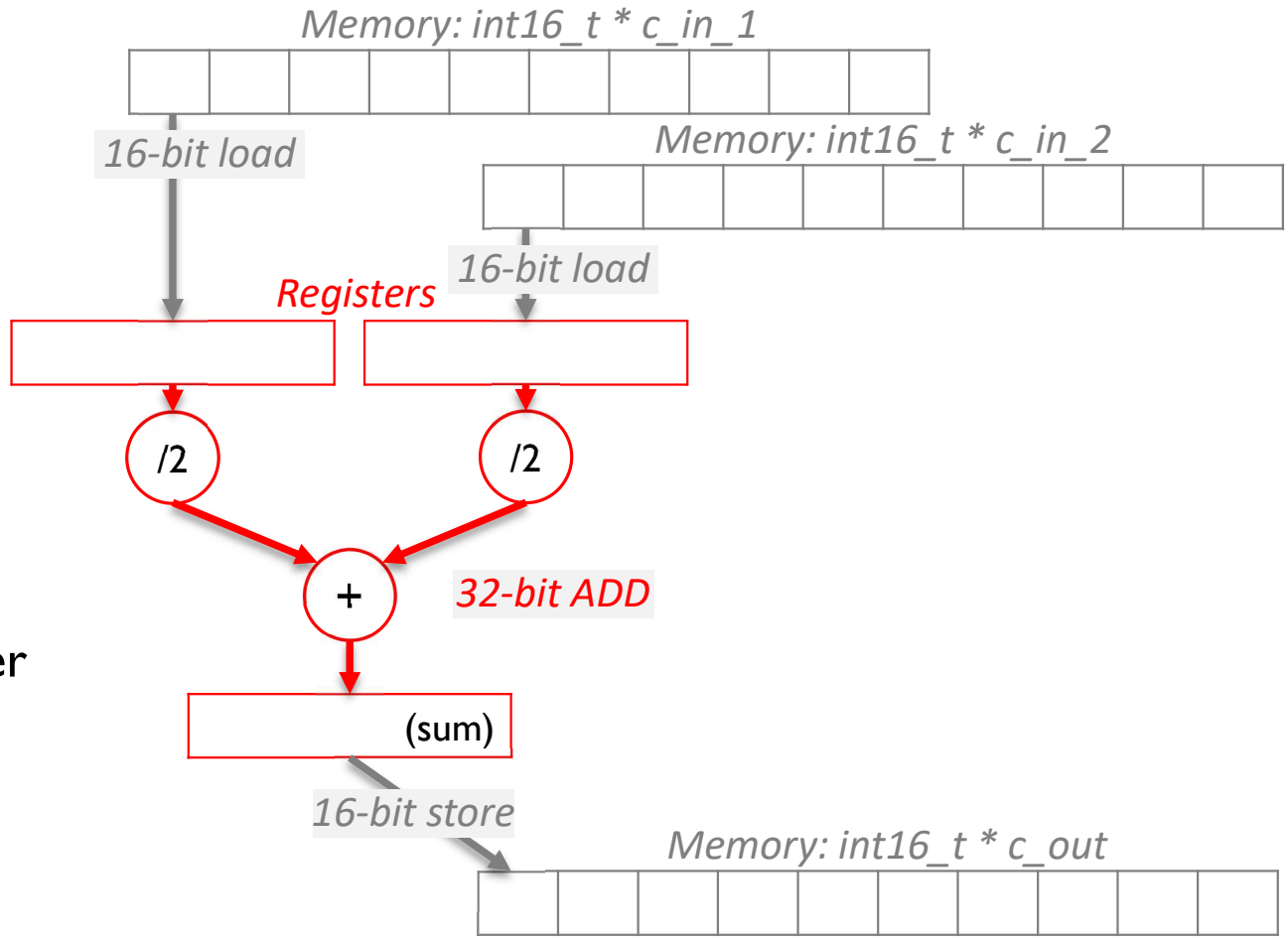
```
int sum_ints( int * x, int n)
     int i, sum_val = 0;

     for (i=0; i<n; i++) {
          sum_val += x[i];
     }
     return sum_val;
}
```

Unrolled Scalar Code

```
int sum_ints( int * x, int n)
     int i, sum_val = 0;

     for (i=0; i<n-3; i+=4) {
          sum_val += x[i];
          sum_val += x[i+1];
          sum_val += x[i+2];
          sum_val += x[i+3];
     }
     for ( ; i<n; i++) {
          sum_val += x[i];
     }
     return sum_val;
}
```

Example Application: Mix Two Audio Channels

```
void mix_channels(
 int16_t * c_in_1,
 int16_t * c_in_2,
 int16_t * c_out,
 int n) {
  int i;
  for (i=0; i<n; i++) {
    *c_out++ = (*c_in_1++)/2 +
               (*c_in_2++)/2;
  }
}
```



- Mix two audio channels together
  - 16-bit signed data
  - c_out ← c_in_1/2 + c_in_2/2

Step 1: Unroll Loop, Add Clean-Up Loop

```
void mix_channels(int16_t * c_in_1, int16_t * c_in_2,
   int16_t * c_out, int n) {
   int i;
```

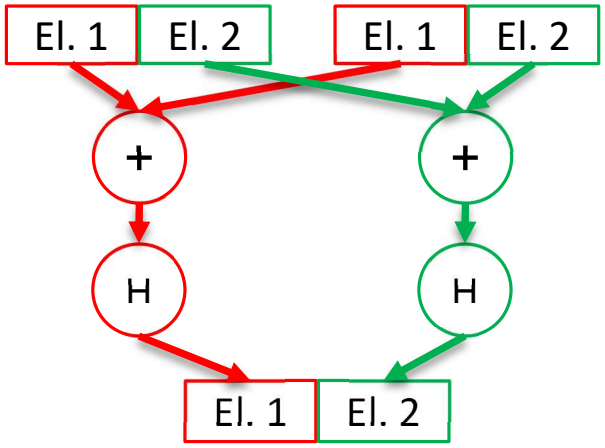
Original

```
   for (i=0; i<n; i++) {
      *c_out++ = (*c_in_1++)/2 + (*c_in_2++)/2;
   }
```

Replacement

```
   for (i=0; i<n-1; i+=2) { // Unroll factor F=2, n-1=n-(F-1)
      *c_out++ = (*c_in_1++)/2 + (*c_in_2++)/2;
      *c_out++ = (*c_in_1++)/2 + (*c_in_2++)/2;
   }
   for (; i<n; i++) {
      *c_out++ = (*c_in_1++)/2 + (*c_in_2++)/2;
   }
```

Step 2: Identify SIMD Instruction(s), Evaluate Data Layout



- Good match for signed halving ADD16 instruction (SHADD16)
- SHADD16 operates on two data elements packed into 32-bit register
- Memory system is 32 bits wide
- Data in memory is laid out sequentially, so we can load and store two elements (32 bits) at a time

c_in_1[0]	c_in_1[1]
c_in_1[2]	c_in_1[3]
c_in_1[4]	c_in_1[5]

c_in_2[0]	c_in_2[1]
c_in_2[2]	c_in_2[3]
c_in_2[4]	c_in_2[5]

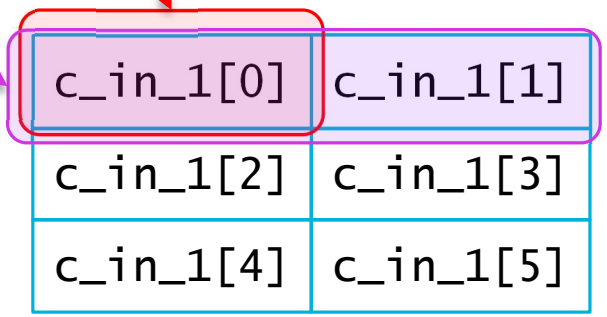
c_out[0]	c_out[1]
c_out[2]	c_out[3]
c_out[4]	c_out[5]

Alexander G. Dean - agdean@ncsu.edu

Step 3: Adjust Code Referencing Packed Data

- Currently input data and output data passed through pointers to int16_t
- Need pointers to packed 32-bit values
  - Will load and store correctly (4 bytes at a time)
  - Will increment correctly (4 byte increment)
- So, set up additional pointers to packed SIMD data (32 bits)
- This often generates no additional code!

```
void mix_channels(int16_t * c_in_1, int16_t
* c_in_2, int16_t * c_out, int n) {

  int32_t * c_in_1_v, * c_in_2_v, *c_out_v;

 c_in_1_v = (int32_t *)c_in_1;
 c_in_2_v = (int32_t *)c_in_2;
 c_out_v  = (int32_t *)c_out;
```



Step 4: Convert Vectorized Loop To Use SIMD Instruction(s)

- Refer to compiler user guide for details on intrinsic syntax

Syntax

`unsigned int __shadd16(unsigned int val1, unsigned int val2)`

Where:

val1

holds the first two 16-bit summands

val2

holds the second two 16-bit summands.

Return value

The `__shadd16` intrinsic returns:

- The halved addition of the low halfwords from each operand, in the low halfword of the return value.
- The halved addition of the high halfwords from each operand, in the high halfword of the return value.

- Replace unrolled loop body with SIMD code

```
for (i=0; i<n-1; i+=2) { // unroll factor 2
        *c_out_v++ = __shadd16(*c_in_1_v++, *c_in_2_v++);
}
```

Step 5: Update Scalar Pointers in Clean-Up Loop

- Have scalar clean-up loop do remaining work,
- Need to update scalar pointers to pick up where vectorized loop finished
- Casting pointers will likely create no new code

```
// Clean-Up Loop
if (i<n) {
      // Update scalar ptrs to match vector ptrs
      c_in_1 = (int16_t *) c_in_1_v;
      c_in_2 = (int16_t *) c_in_2_v;
      c_out = (int16_t *) c_out_v;
      for (; i<n; i++) {
            *c_out++ = *c_in_1++/2 + *c_in_2++/2;
            }
}
```

Resulting Loop Body Object Code and Performance

Scalar Code

```
    *c_out++ = (*c_in_1++)/2+
                  (*c_in_2++)/2;
L1: LDRSH  r8,[r3,#0x02]
    LDRSH  r7,[r2,#0x02]
    ADD    r8,r8,r8,LSR #31
    ASR    r8,r8,#1
    ADD    r7,r7,r7,LSR #31
    ADD    r7,r8,r7,ASR #1
    STRH   r7,[r1,#0x02]
    LDRSH  r7,[r3,#0x04]!
    LDRSH  r8,[r2,#0x04]!
    ADD    r7,r7,r7,LSR #31
    ASR    r7,r7,#1
    ADD    r8,r8,r8,LSR #31
    ADD    r7,r7,r8,ASR #1
    STRH   r7,[r1,#0x04]!
    SUBS   r0,r0,#1
    BNE    L1
```

Signed divide by 2

Signed divide by 2

Compiler unrolled loop by factor of two

Signed divide by 2

Signed divide by 2

SIMD Code

```
    *c_out_v++ = __shadd16(*c_in_1_v++,
                           *c_in_2_v++);
L1: LDR      r9,[r3],#0x04
    LDR      r10,[r2],#0x04
    SHADD16  r9,r9,r10
    ADDS     r0,r0,#2
    STR      r9,[r1],#0x04
    CMP      r0,r8
    BLT      L1
```

	Scalar Code	SIMD Code
Total Duration	11.78 µs	5.5 µs
Time per element	92 ns	43
Clock cycles per element	11	5.1

LIMITS TO VECTORIZATION

Conditional Control Flow in Loops

- SIMD – Single Instruction, Multiple Data
- Conditions (if, ?:, etc.) usually introduce conditional control-flow in the loop body
- Multiple control-flow operations -> Multiple PCs -> Multiple Instruction
  - Not allowed in SIMD!

Alexander G. Dean - agdean@ncsu.edu

ISA May Help Eliminate Conditional Control Flow

- More complex instruction may absorb conditional control flow
  - Saturating math instructions. No overflow test and clean-up code needed!
  - Select instruction: copy r0 or r1 into r2, based on value in r3
- Predication: conditional execution
  - Instruction is processed, but predicate register controls if results are written to destinations
  - Conditional branch is simple example:
    - CMP r0, r3: Compare values, write condition code flags
    - BNE label: Branch writes label to PC if flags indicate NE (r0 != r3), otherwise it has no effects (acts like NOP)
  - Could make other instructions conditional:
    - ADDNE r4, r5, r6: Put r5+r6 into r4 if flags indicate NE (e.g. previous comparison resulted in Not Equal condition)
- ARMv7-M Features
  - Saturating math
  - Select operation
  - Bitwise logic operations
- ARMv8.1-M/Helium Features
  - Instruction predication: Conditional execution for some instructions based on condition code flags
  - Lane predication: Conditional execution for some lanes based on VPR register contents (set by vector compare instructions)
  - Rounding and saturating shift instructions

Loop-Carried Dependencies

- Loop-carried dependency exists if a calculation in iteration *n* depends on the result of any previous iteration *m*, where *m*<*n*
- This dependency prevents vectorization
  - Can't do multiple iterations simultaneously, but may be able to overlap them (software pipelining) to reduce total time
- Sometimes is possible to restructure code to remove it, but not always

```
float x[N], y[N];

for (n=1; n<N; n++) {
      x[n] = y[n] * x[n-1];
}


// Unrolling once leads to this
for (n=1; n<N; n+=2) {
      x[n] = y[n] * x[n-1];
      x[n+1] = y[n+1] * x[n];
}
```