SIMD Engines Comparative Reference

Konstantinos Margaritis
markos@freevec.org

June 9, 2014

Chapter 1

Introduction

Chapter 2

SIMD Engines Overview

2.1 Intel SSE family

2.1.1 Intel AVX, AVX2 & AVX512

2.2 ARM NEON (armv7 & armv8)

2.2.1 ARMv6 SIMD

2.3 PowerPC Altivec/VMX and derivatives

2.3.1 VMX128

2.3.2 VSX

2.4 Others

2.4.1 Cell SPU

2.5 Detection

2.5.1 Compile-time Detection

2.5.2 Runtime Detection

Chapter 3

Data types

3.1 SSE and AVX

3.2 ARM NEON

3.3 PowerPC Altivec

Chapter 4

Comparing Functionality

4.1 Addition, modulo/wrap-around

Modulo/wrap-around addition is the most common amongst CPUs, where in case of an overflow, the value is wrapped around to the minimum value. In case of 8-bit unsigned bytes (unsigned char), adding 1 to 255 using modulo/wrap-around addition would produce the result 0.

4.1.1 8-bit Integers

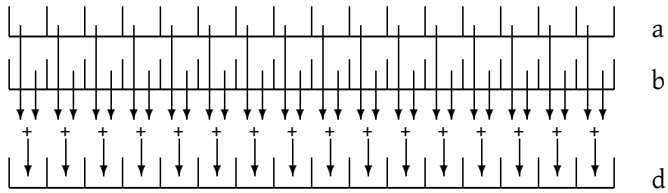

Figure 4.1: Addition of 16 8-bit integer elements in 128-bit vectors

C Intrinsics

SIMD	Code	Input	Output
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_add_pi8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi8(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_add_epi8(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vadd_u8(a, b)`	`uint8x8_t a, b;`	`uint8x8_t d;`
NEON(128-bit)	`d = vaddq_u8(a, b)`	`uint8x16_t a, b;`	`uint8x16_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
VSX[1]	`d = vec_add(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
8-bit signed			
NEON(64-bit)	`d = vadd_s8(a, b)`	`int8x8_t a, b;`	`int8x8_t d;`
NEON(128-bit)	`d = vaddq_s8(a, b)`	`int8x16_t a, b;`	`int8x16_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`
VSX[2]	`d = vec_add(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`

[1] the VMX instruction will be used
[2] the VMX instruction will be used

Notes

Assembly

SIMD	Code	Description	Notes
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`paddb mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddb xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddb xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddb ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vadd.i8 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vaddq.i8 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`add Vd.8B, Vn.8B, Vm.8B`	Adds Vn.8B, Vm.8B, result in Vd.8B	d1,d2,d3 are 64-bit NEON registers
NEON(armv8, 128-bit)	`add Vd.16B, Vn.16B, Vm.16B`	Adds Vn.16B, Vm.16B, result in Vd.16B	Vn.16B, Vm.16B, Vd.16B are 128-bit NEON registers
AltiVec/VMX	`vaddubm vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are VMX 128-bit integer registers
VSX	`vaddubm vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
8-bit signed			
NEON(64-bit)	`vadd.s8 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vaddq.s8 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
__m128i b = _mm_set_epi8(15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_add_epi8(a, b);
```

NEON (128-bit)

```
uint8x16_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
uint8x16_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
uint8x16_t d = vaddq_u8(a, b);
```

AltiVec/VMX

```
vector uint8_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
vector uint8_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
vector uint8_t d = vec_add(a, b);
```

4.1.2 16-bit integers

TODO: insert diagram for 16-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
16-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_add_pi16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi16(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_add_epi16(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vadd_u16(a, b)`	`uint16x4_t a, b;`	`uint16x4_t d;`
NEON(128-bit)	`d = vaddq_u16(a, b)`	`uint16x8_t a, b;`	`uint16x8_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
VSX[3]	`d = vec_add(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
16-bit signed			
NEON(64-bit)	`d = vadd_s16(a, b)`	`int16x4_t a, b;`	`int16x4_t d;`
NEON(128-bit)	`d = vaddq_s16(a, b)`	`int16x8_t a, b;`	`int16x8_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`
VSX[4]	`d = vec_add(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`

[3] the VMX instruction will be used
[4] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`paddw mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddw xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddw xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddw ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vadd.i16 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vaddq.i16 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`add Vd.4H, Vn.4H, Vm.4H`	Adds Vn.4H, Vm.4H, result in Vd.4H	Vn.4H, Vm.4H, Vd.4H are 64-bit NEON registers
NEON(armv8, 128-bit)	`add Vd.8H, Vn.8H, Vm.8H`	Adds Vn.8H, Vm.8H, result in Vd.8H	Vn.8H, Vm.8H, Vd.8H are 128-bit NEON registers
AltiVec/VMX	`vadduhm vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vadduhm vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
8-bit signed			
NEON(64-bit)	`vadd.s16 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vaddq.s16 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi16(0, 1, 2, 3, 4, 5, 6, 7);
__m128i b = _mm_set_epi16(7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_add_epi16(a, b);
```

NEON (128-bit)

```
uint16x8_t a = {0, 1, 2, 3, 4, 5, 6, 7};
uint16x8_t b = {7, 6, 5, 4, 3, 2, 1, 0};
uint16x8_t d = vaddq_u16(a, b);
```

AltiVec/VMX

```
vector uint16_t a = {0, 1, 2, 3, 4, 5, 6, 7};
vector uint16_t b = {7, 6, 5, 4, 3, 2, 1, 0};
vector uint16_t d = vec_add(a, b);
```

4.1.3 32-bit integers

TODO: insert diagram for 32-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_add_pi32(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi32(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_add_epi32(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vadd_u32(a, b)`	`uint32x2_t a, b;`	`uint32x2_t d;`
NEON(128-bit)	`d = vaddq_u32(a, b)`	`uint32x4_t a, b;`	`uint32x4_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
VSX[5]	`d = vec_add(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
8-bit signed			
NEON(64-bit)	`d = vadd_s32(a, b)`	`int32x2_t a, b;`	`int32x2_t d;`
NEON(128-bit)	`d = vaddq_s32(a, b)`	`int32x4_t a, b;`	`int32x4_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`
VSX[6]	`d = vec_add(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`

[5] the VMX instruction will be used
[6] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`paddd mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddd xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddd xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddd ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vadd.i32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vaddq.i32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`add Vd.2S, Vn.2S, Vm.2S`	Adds Vn.2S, Vm.2S, result in Vd.2S	Vn.2S, Vm.2S, Vd.2S are 64-bit NEON registers
NEON(armv8, 128-bit)	`add Vd.4S, Vn.4S, Vm.4S`	Adds Vn.4S, Vm.4S, result in Vd.4S	Vn.4S, Vm.4S, Vd.4S are 128-bit NEON registers
AltiVec/VMX	`vadduwm vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vadduwm vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
8-bit signed			
NEON(64-bit)	`vadd.s32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vaddq.s32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi32(0, 1, 2, 3);
__m128i b = _mm_set_epi32(3, 2, 1, 0);
__m128i d = _mm_add_epi32(a, b);
```

NEON (128-bit)

```
uint32x4_t a = {0, 1, 2, 3};
uint32x4_t b = {3, 2, 1, 0};
uint32x4_t d = vaddq_u32(a, b);
```

AltiVec/VMX

```
vector uint32_t a = {0, 1, 2, 3};
vector uint32_t b = {3, 2, 1, 0};
vector uint32_t d = vec_add(a, b);
```

4.1.4 64-bit integers

TODO: insert diagram for 64-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_add_si64(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_add_epi64(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_add_epi64(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vadd_u64(a, b)`	`uint64x1_t a, b;`	`uint64x1_t d;`
NEON(128-bit)	`d = vaddq_u64(a, b)`	`uint64x2_t a, b;`	`uint64x2_t d;`
AltiVec/VMX[7]	N/A	–	–
VSX	`d = vec_add(a, b)`	`vector uint64_t a, b;`	`vector uint64_t d;`
8-bit signed			
NEON(64-bit)	`d = vadd_s64(a, b)`	`int64x1_t a, b;`	`int64x1_t d;`
NEON(128-bit)	`d = vaddq_s64(a, b)`	`int64x2_t a, b;`	`int64x2_t d;`
VSX	`d = vec_add(a, b)`	`vector int64_t a, b;`	`vector int64_t d;`

Assembly

SIMD	Code	Description	Notes
8-bit unsigned/signed			
MMX(64-bit)	`paddq mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddq xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddq xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddq ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vadd.i64 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vaddq.i64 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`add Dd, Dn, Dm`	Adds Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`add Vd.2D, Vn.2D, Vm.2D`	Adds Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
AltiVec/VMX	N/A	–	–
VSX	`vadduqm vD, vA, vB`	Adds vA, vB, result in vD	VSX extension
8-bit signed			
NEON(64-bit)	`vadd.s64 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vaddq.s64 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

[7] AltiVec/VMX does not support 64-bit integer arithmetic

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi64(0, 1);
__m128i b = _mm_set_epi64(3, 2);
__m128i d = _mm_add_epi64(a, b);
```

AVX (256-bit)

```
__m256i a = _mm256_set_epi64(0, 1, 2, 3);
__m256i b = _mm256_set_epi64(3, 2, 1, 0);
__m256i d = _mm256_add_epi64(a, b);
```

NEON (128-bit)

```
uint64x2_t a = {0, 1};
uint64x2_t b = {3, 2};
uint64x2_t d = vaddq_u64(a, b);
```

VSX

```
vector uint64_t a = {0, 1};
vector uint64_t b = {1, 0};
vector uint64_t d = vec_add(a, b);
```

4.1.5 32-bit floats

TODO: insert diagram for 32-bit float vector addition

C Intrinsics

SIMD	Code	Input	Output
MMX(64-bit)[8]	N/A	–	–
SSE(64-bit)	`d = _mm_add_ss(a, b)`	`__m128 a, b;`	`__m128 d;`
SSE(128-bit)	`d = _mm_add_ps(a, b)`	`__m128 a, b;`	`__m128 d;`
AVX(256-bit)	`d = _mm256_add_ps(a, b)`	`__m256 a, b;`	`__m256 d;`
NEON(64-bit)	`d = vadd_f32(a, b)`	`float32x2_t a, b;`	`float32x2_t d;`
NEON(128-bit)	`d = vaddq_f32(a, b)`	`float32x4_t a, b;`	`float32x4_t d;`
AltiVec/VMX	`d = vec_add(a, b)`	`vector float a, b;`	`vector float d;`
VSX[9]	`d = vec_add(a, b)`	`vector float a, b;`	`vector float d;`

[8] MMX does not support float arithmetic
[9] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
MMX(64-bit)	N/A	–	–
SSE(64-bit)	`addss xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit double registers, but the operation takes place only at the lower 64-bits
SSE(128-bit)	`addps xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit float registers
AVX(128-bit)	`addps xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`addps ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vadd.f32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vaddq.f32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`fadd Vd.2S, Vn.2S, Vm.2S`	Adds Vn.2S, Vm.2S, result in Vd.2S	Vn.2S, Vm.2S, Vd.2S are 64-bit NEON registers
NEON(armv8, 128-bit)	`fadd Vd.4S, Vn.4S, Vm.4S`	Adds Vn.4S, Vm.4S, result in Vd.4S	Vn.4S, Vm.4S, Vd.4S are 128-bit NEON registers
AltiVec/VMX	`vaddfp vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vaddfp vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
VSX	`xsaddsp vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are VSX registers holding double values, but the operation takes place only at the lower 64-bits (FIXME: better description)

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128 a = _mm_set_ps(1.0, 2.0, 3.0, 4.0);
__m128 b = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
__m128 d = _mm_add_ps(a, b);
```

AVX (256-bit)

```
__m256 a = _mm256_set_ps(1.0, 2.0, 3.0, 4.0);
__m256 b = _mm256_set_ps(4.0, 3.0, 2.0, 1.0);
__m256 d = _mm256_add_ps(a, b);
```

NEON (128-bit)

```
float32x4_t a = {0, 1, 2, 3};
float32x4_t b = {3, 2, 1, 0};
float32x4_t d = vaddq_f32(a, b);
```

AltiVec/VMX

```
vector float a = {0, 1, 2, 3};
vector float b = {3, 2, 1, 0};
vector float d = vec_add(a, b);
```

4.1.6 64-bit doubles

TODO: insert diagram for 64-bit float vector addition

C Intrinsics

SIMD	Code	Input	Output
MMX(64-bit)[10]	N/A	–	–
SSE2(64-bit)	`d = _mm_add_sd(a, b)`	`__m128d a, b;`	`__m128d d;`
SSE2(128-bit)	`d = _mm_add_pd(a, b)`	`__m128d a, b;`	`__m128d d;`
AVX(256-bit)	`d = _mm256_add_pd(a, b)`	`__m256d a, b;`	`__m256d d;`
NEON(64-bit)[11]	`d = vadd_f64(a, b)`	`float64x1_t a, b;`	`float64x1_t d;`
NEON(128-bit)	`d = vaddq_f64(a, b)`	`float64x2_t a, b;`	`float64x2_t d;`
AltiVec/VMX[12]	N/A	–	–
VSX	`d = vec_add(a, b)`	`vector double a, b;`	`vector double d;`

[10] MMX does not support double arithmetic
[11] double arithmetic is supported only by armv8 NEON
[12] AltiVec/VMX does not support double arithmetic

Assembly

SIMD	Code	Description	Notes
MMX(64-bit)	N/A	–	–
SSE2(64-bit)	`addsd xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit double registers, but the operation takes place only at the lower 64-bits
SSE2(128-bit)	`addpd xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit float registers
AVX(128-bit)	`addpd xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`addpd ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	N/A	–	–
NEON(armv7, 128-bit)	N/A	–	–
NEON(armv8, 64-bit)	`fadd Dd, Dn, Dm`	Adds Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`fadd Vd.2D, Vn.2D, Vm.2D`	Adds Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
AltiVec/VMX	N/A	–	–
VSX	`xsadddp vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are VSX registers holding double values, but the operation takes place only at the lower 64-bits
VSX	`xvadddp vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are VSX registers holding double values

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128d a = _mm_set_pd(1.0, 2.0, 3.0, 4.0);
__m128d b = _mm_set_pd(4.0, 3.0, 2.0, 1.0);
__m128d d = _mm_add_pd(a, b);
```

AVX (256-bit)

```
__m256d a = _mm256_set_pd(1.0, 2.0, 3.0, 4.0);
__m256d b = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
__m256d d = _mm256_add_pd(a, b);
```

NEON (128-bit)

```
float64x2_t a = {0, 1};
float64x2_t b = {3, 2};
float64x2_t d = vaddq_f64(a, b);
```

VSX

```
vector double a = {0, 1, 2, 3};
vector double b = {3, 2, 1, 0};
vector double d = vec_add(a, b);
```

4.2 Addition, saturated

Saturated addition in contrast to modulo addition, keeps the result in a fixed range between a minimum and maximum value. If it's greater than the maximum value, then it's "clamped" to the maximum value, if it's below the minimum then it's set to the minimum value. In case of 8-bit unsigned bytes (unsigned char), adding 10 to 255 using saturated addition would produce the result 255.

4.2.1 8-bit Integers

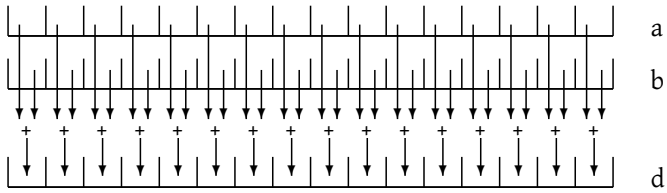

Figure 4.2: Addition of 16 8-bit integer elements in 128-bit vectors

C Intrinsics

SIMD	Code	Input	Output
8-bit unsigned			
MMX(64-bit)	`d = _mm_adds_pu8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_adds_epu8(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_add_epu8(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vadds_u8(a, b)`	`uint8x8_t a, b;`	`uint8x8_t d;`
NEON(128-bit)	`d = vaddsq_u8(a, b)`	`uint8x16_t a, b;`	`uint8x16_t d;`
AltiVec/VMX	`d = vec_adds(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
VSX[13]	`d = vec_adds(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
8-bit signed			
MMX(64-bit)	`d = _mm_adds_pi8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_adds_epi8(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_adds_epi8(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vqadd_s8(a, b)`	`int8x8_t a, b;`	`int8x8_t d;`
NEON(128-bit)	`d = vqaddq_s8(a, b)`	`int8x16_t a, b;`	`int8x16_t d;`
AltiVec/VMX	`d = vec_adds(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`
VSX[14]	`d = vec_adds(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`

[13] the VMX instruction will be used
[14] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
8-bit unsigned			
MMX(64-bit)	`paddusb mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddusb xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddusb xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddusb ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vqadd.i8 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqaddq.i8 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqadd Vd.8B, Vn.8B, Vm.8B`	Adds Vn.8B, Vm.8B, result in Vd.8B	d1,d2,d3 are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqadd Vd.16B, Vn.16B, Vm.16B`	Adds Vn.16B, Vm.16B, result in Vd.16B	Vn.16B, Vm.16B, Vd.16B are 128-bit NEON registers
AltiVec/VMX	`vaddubs vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vaddubs vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
8-bit signed			
MMX(64-bit)	`paddsb mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddsb xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddsb xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddsb ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vqadd.s8 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqaddq.s8 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqadd Vd.8B, Vn.8B, Vm.8B`	Adds Vn.8B, Vm.8B, result in Vd.8B	d1,d2,d3 are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqadd Vd.16B, Vn.16B, Vm.16B`	Adds Vn.16B, Vm.16B, result in Vd.16B	Vn.16B, Vm.16B, Vd.16B are 128-bit NEON registers
AltiVec/VMX	`vaddsbs vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vaddsbs vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epu8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
__m128i b = _mm_set_epu8(15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_adds_epu8(a, b);
```

NEON (128-bit)

```
int8x16_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
int8x16_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int8x16_t d = vqaddq_s8(a, b);
```

AltiVec/VMX

```
vector uint8_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
vector uint8_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
vector uint8_t d = vec_adds(a, b);
```

4.2.2 16-bit integers

TODO: insert diagram for 16-bit saturated vector addition

C Intrinsics

SIMD	Code	Input	Output
16-bit unsigned			
MMX(64-bit)	`d = _mm_adds_pu16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_adds_epu16(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_adds_epu16(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vqadd_u16(a, b)`	`uint16x4_t a, b;`	`uint16x4_t d;`
NEON(128-bit)	`d = vqaddq_u16(a, b)`	`uint16x8_t a, b;`	`uint16x8_t d;`
AltiVec/VMX	`d = vec_adds(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
VSX[15]	`d = vec_adds(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
16-bit signed			
MMX(64-bit)	`d = _mm_adds_pi16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_adds_epi16(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_adds_epi16(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vqadd_s16(a, b)`	`int16x4_t a, b;`	`int16x4_t d;`
NEON(128-bit)	`d = vqaddq_s16(a, b)`	`int16x8_t a, b;`	`int16x8_t d;`
AltiVec/VMX	`d = vec_adds(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`
VSX[16]	`d = vec_adds(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`

[15] the VMX instruction will be used
[16] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
16-bit unsigned			
MMX(64-bit)	`paddsw mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddsw xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddsw xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddsw ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vqadd.i16 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqaddq.i16 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqadd Vd.4H, Vn.4H, Vm.4H`	Adds Vn.4H, Vm.4H, result in Vd.4H	Vn.4H, Vm.4H, Vd.4H are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqadd Vd.8H, Vn.8H, Vm.8H`	Adds Vn.8H, Vm.8H, result in Vd.8H	Vn.8H, Vm.8H, Vd.8H are 128-bit NEON registers
AltiVec/VMX	`vadduhm vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vadduhm vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
16-bit signed			
MMX(64-bit)	`paddsw mm1, mm2`	Adds mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`paddsw xmm1, xmm2`	Adds xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpaddsw xmm1, xmm2, xmm3`	Adds xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpaddsw ymm1, ymm2, ymm3`	Adds ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(64-bit)	`vqadd.s16 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vqaddq.s16 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqadd Vd.4H, Vn.4H, Vm.4H`	Adds Vn.4H, Vm.4H, result in Vd.4H	Vn.4H, Vm.4H, Vd.4H are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqadd Vd.8H, Vn.8H, Vm.8H`	Adds Vn.8H, Vm.8H, result in Vd.8H	Vn.8H, Vm.8H, Vd.8H are 128-bit NEON registers
AltiVec/VMX	`vaddshm vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vaddshm vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi16(0, 1, 2, 3, 4, 5, 6, 7);
__m128i b = _mm_set_epi16(7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_adds_epi16(a, b);
```

NEON (128-bit)

```
int16x8_t a = {0, 1, 2, 3, 4, 5, 6, 7};
int16x8_t b = {7, 6, 5, 4, 3, 2, 1, 0};
int16x8_t d = vqaddq_s16(a, b);
```

AltiVec/VMX

```
vector uint16_t a = {0, 1, 2, 3, 4, 5, 6, 7};
vector uint16_t b = {7, 6, 5, 4, 3, 2, 1, 0};
vector uint16_t d = vec_adds(a, b);
```

4.2.3 32-bit integers

TODO: insert diagram for 32-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
32-bit unsigned			
MMX(64-bit)[17]	N/A	–	–
SSE2(128-bit)[18]	N/A	–	–
AVX(256-bit)[19]	N/A	–	–
NEON(64-bit)	`d = vqadd_u32(a, b)`	`uint32x2_t a, b;`	`uint32x2_t d;`
NEON(128-bit)	`d = vqaddq_u32(a, b)`	`uint32x4_t a, b;`	`uint32x4_t d;`
AltiVec/VMX	`d = vec_adds(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
VSX[20]	`d = vec_adds(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
32-bit signed			
MMX(64-bit)	N/A	–	–
SSE2(128-bit)	N/A	–	–
AVX(256-bit)	N/A	–	–
NEON(64-bit)	`d = vqadd_s32(a, b)`	`int32x2_t a, b;`	`int32x2_t d;`
NEON(128-bit)	`d = vqaddq_s32(a, b)`	`int32x4_t a, b;`	`int32x4_t d;`
AltiVec/VMX	`d = vec_adds(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`
VSX	`d = vec_adds(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`

[17] MMX has no support for saturated addition on 32-bit integers
[18] SSE has no support for saturated addition on 32-bit integers
[19] AVX has no support for saturated addition on 32-bit integers
[20] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
32-bit unsigned			
MMX(64-bit)	N/A	–	–
SSE2(128-bit)	N/A	–	–
AVX(128-bit)	N/A	–	–
AVX(256-bit)	N/A	–	–
NEON(64-bit)	`vqadd.i32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vqaddq.i32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqadd Dd, Dn, Dm`	Adds Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqadd Vd.2D, Vn.2D, Vm.2D`	Adds Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
AltiVec/VMX	`vadduws vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vadduws vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
32-bit signed			
MMX(64-bit)	N/A	–	–
SSE2(128-bit)	N/A	–	–
AVX(128-bit)	N/A	–	–
AVX(256-bit)	N/A	–	–
NEON(64-bit)	`vqadd.s32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vqaddq.s32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqadd Dd, Dn, Dm`	Adds Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqadd Vd.2D, Vn.2D, Vm.2D`	Adds Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
AltiVec/VMX	`vaddsws vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vaddsws vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

NEON (128-bit)

```
uint32x4_t a = {0, 1, 2, 3};
uint32x4_t b = {3, 2, 1, 0};
uint32x4_t d = vaddq_u32(a, b);
```

AltiVec/VMX

```
vector uint32_t a = {0, 1, 2, 3};
vector uint32_t b = {3, 2, 1, 0};
vector uint32_t d = vec_adds(a, b);
```

4.2.4 64-bit integers

At the moment the only architecture that supports 64-bit saturated addition is NEON, in both ARMv7 and ARMv8 implementations.

TODO: insert diagram for 64-bit vector saturated addition

C Intrinsics

SIMD	Code	Input	Output
64-bit unsigned			
NEON(armv7, 64-bit)	`d = vqadd_u64(a, b)`	`uint64x1_t a, b;`	`uint64x1_t d;`
NEON(armv7, 128-bit)	`d = vqaddq_u64(a, b)`	`uint64x2_t a, b;`	`uint64x2_t d;`
64-bit signed			
NEON(64-bit)	`d = vqadd_s64(a, b)`	`int64x1_t a, b;`	`int64x1_t d;`
NEON(128-bit)	`d = vqaddq_s64(a, b)`	`int64x2_t a, b;`	`int64x2_t d;`

Assembly

SIMD	Code	Description	Notes
64-bit unsigned			
NEON(armv7, 64-bit)	`vqadd.i64 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqaddq.i64 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqadd Dd, Dn, Dm`	Adds Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqadd Vd.2D, Vn.2D, Vm.2D`	Adds Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
64-bit signed			
NEON(64-bit)	`vqadd.s64 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vqaddq.s64 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqadd Dd, Dn, Dm`	Adds Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqadd Vd.2D, Vn.2D, Vm.2D`	Adds Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

NEON (128-bit)

```
uint64x2_t a = {0, 1};
uint64x2_t b = {3, 2};
uint64x2_t d = vaddq_u64(a, b);
```

4.3 Subtraction, modulo/wrap-around

FIXME: Modulo/wrap-around addition is the most common amongst CPUs, where in case of an overflow, the value is wrapped around to the minimum value. In case of 8-bit unsigned bytes (unsigned char), adding 1 to 255 using modulo/wrap-around addition would produce the result 0.

4.3.1 8-bit Integers

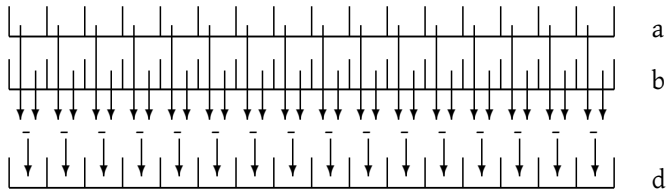

Figure 4.3: Addition of 16 8-bit integer elements in 128-bit vectors

C Intrinsics

SIMD	Code	Input	Output
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_sub_pi8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_sub_epi8(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_sub_epi8(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vsub_u8(a, b)`	`uint8x8_t a, b;`	`uint8x8_t d;`
NEON(128-bit)	`d = vsubq_u8(a, b)`	`uint8x16_t a, b;`	`uint8x16_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
VSX[21]	`d = vec_sub(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
8-bit signed			
NEON(64-bit)	`d = vsub_s8(a, b)`	`int8x8_t a, b;`	`int8x8_t d;`
NEON(128-bit)	`d = vsubq_s8(a, b)`	`int8x16_t a, b;`	`int8x16_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`
VSX[22]	`d = vec_sub(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`

[21] the VMX instruction will be used
[22] the VMX instruction will be used

Notes

Assembly

SIMD	Code	Description	Notes
8-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`psubb mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubb xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubb xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubb ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(64-bit)	`vsub.i8 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.i8 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sub Vd.8B, Vn.8B, Vm.8B`	Subtracts Vn.8B, Vm.8B, result in Vd.8B	d1,d2,d3 are 64-bit NEON registers
NEON(armv8, 128-bit)	`sub Vd.16B, Vn.16B, Vm.16B`	Subtracts Vn.16B, Vm.16B, result in Vd.16B	Vn.16B, Vm.16B, Vd.16B are 128-bit NEON registers
AltiVec/VMX	`vsububm vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are VMX 128-bit integer registers
VSX	`vsububm vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction
8-bit signed			
NEON(64-bit)	`vsub.s8 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.s8 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
__m128i b = _mm_set_epi8(15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_sub_epi8(a, b);
```

NEON (128-bit)

```
uint8x16_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
uint8x16_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
uint8x16_t d = vsubq_u8(a, b);
```

AltiVec/VMX

```
vector uint8_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
vector uint8_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
vector uint8_t d = vec_sub(a, b);
```

4.3.2 16-bit integers

TODO: insert diagram for 16-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
16-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_sub_pi16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_sub_epi16(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_sub_epi16(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vsub_u16(a, b)`	`uint16x4_t a, b;`	`uint16x4_t d;`
NEON(128-bit)	`d = vsubq_u16(a, b)`	`uint16x8_t a, b;`	`uint16x8_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
VSX[23]	`d = vec_sub(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
16-bit signed			
NEON(64-bit)	`d = vsub_s16(a, b)`	`int16x4_t a, b;`	`int16x4_t d;`
NEON(128-bit)	`d = vsubq_s16(a, b)`	`int16x8_t a, b;`	`int16x8_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`
VSX[24]	`d = vec_sub(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`

[23] the VMX instruction will be used
[24] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
16-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`psubw mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubw xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubw xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubw ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(64-bit)	`vsub.i16 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.i16 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sub Vd.4H, Vn.4H, Vm.4H`	Subtracts Vn.4H, Vm.4H, result in Vd.4H	Vn.4H, Vm.4H, Vd.4H are 64-bit NEON registers
NEON(armv8, 128-bit)	`sub Vd.8H, Vn.8H, Vm.8H`	Subtracts Vn.8H, Vm.8H, result in Vd.8H	Vn.8H, Vm.8H, Vd.8H are 128-bit NEON registers
AltiVec/VMX	`vsubuhm vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubuhm vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction
16-bit signed			
NEON(64-bit)	`vsub.s16 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.s16 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi16(0, 1, 2, 3, 4, 5, 6, 7);
__m128i b = _mm_set_epi16(7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_sub_epi16(a, b);
```

NEON (128-bit)

```
uint16x8_t a = {0, 1, 2, 3, 4, 5, 6, 7};
uint16x8_t b = {7, 6, 5, 4, 3, 2, 1, 0};
uint16x8_t d = vsubq_u16(a, b);
```

AltiVec/VMX

```
vector uint16_t a = {0, 1, 2, 3, 4, 5, 6, 7};
vector uint16_t b = {7, 6, 5, 4, 3, 2, 1, 0};
vector uint16_t d = vec_sub(a, b);
```

4.3.3 32-bit integers

TODO: insert diagram for 32-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
32-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_sub_pi32(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_sub_epi32(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_sub_epi32(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vsub_u32(a, b)`	`uint32x2_t a, b;`	`uint32x2_t d;`
NEON(128-bit)	`d = vsubq_u32(a, b)`	`uint32x4_t a, b;`	`uint32x4_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
VSX[25]	`d = vec_sub(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
32-bit signed			
NEON(64-bit)	`d = vsub_s32(a, b)`	`int32x2_t a, b;`	`int32x2_t d;`
NEON(128-bit)	`d = vsubq_s32(a, b)`	`int32x4_t a, b;`	`int32x4_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`
VSX[26]	`d = vec_sub(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`

[25] the VMX instruction will be used
[26] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
32-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`psubd mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubd xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubd xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubd ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(64-bit)	`vsub.i32 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.i32 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sub Vd.2S, Vn.2S, Vm.2S`	Subtracts Vn.2S, Vm.2S, result in Vd.2S	Vn.2S, Vm.2S, Vd.2S are 64-bit NEON registers
NEON(armv8, 128-bit)	`sub Vd.4S, Vn.4S, Vm.4S`	Subtracts Vn.4S, Vm.4S, result in Vd.4S	Vn.4S, Vm.4S, Vd.4S are 128-bit NEON registers
AltiVec/VMX	`vsubuwm vD, vA, vB`	Subtracts vB from vA, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubuwm vD, vA, vB`	Subtracts vB from vA, result in vD	VMX instruction
32-bit signed			
NEON(64-bit)	`vsub.s32 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.s32 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi32(0, 1, 2, 3);
__m128i b = _mm_set_epi32(3, 2, 1, 0);
__m128i d = _mm_sub_epi32(a, b);
```

NEON (128-bit)

```
uint32x4_t a = {0, 1, 2, 3};
uint32x4_t b = {3, 2, 1, 0};
uint32x4_t d = vsubq_u32(a, b);
```

AltiVec/VMX

```
vector uint32_t a = {0, 1, 2, 3};
vector uint32_t b = {3, 2, 1, 0};
vector uint32_t d = vec_sub(a, b);
```

4.3.4 64-bit integers

TODO: insert diagram for 64-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
64-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`d = _mm_sub_si64(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_sub_epi64(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_sub_epi64(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vsub_u64(a, b)`	`uint64x1_t a, b;`	`uint64x1_t d;`
NEON(128-bit)	`d = vsubq_u64(a, b)`	`uint64x2_t a, b;`	`uint64x2_t d;`
AltiVec/VMX[27]	N/A	–	–
VSX	`d = vec_sub(a, b)`	`vector uint64_t a, b;`	`vector uint64_t d;`
64-bit signed			
NEON(64-bit)	`d = vsub_u64(a, b)`	`uint64x1_t a, b;`	`uint64x1_t d;`
NEON(128-bit)	`d = vsubq_u64(a, b)`	`uint64x2_t a, b;`	`uint64x2_t d;`
VSX	`d = vec_sub(a, b)`	`vector int64_t a, b;`	`vector int64_t d;`

Assembly

SIMD	Code	Description	Notes
64-bit unsigned/signed (unless specified otherwise)			
MMX(64-bit)	`psubq mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubq xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubq xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubq ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vsub.i64 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vsubq.i64 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sub Dd, Dn, Dm`	Subtracts Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`sub Vd.2D, Vn.2D, Vm.2D`	Subtracts Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
AltiVec/VMX	N/A	–	–
VSX	`vsubuqm vD, vA, vB`	Subtracts vA, vB, result in vD	VSX extension
64-bit signed			
NEON(armv7, 64-bit)	`vsub.s64 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vsubq.s64 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers

[27] AltiVec/VMX does not support 64-bit integer arithmetic

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi64(0, 1);
__m128i b = _mm_set_epi64(3, 2);
__m128i d = _mm_sub_epi64(a, b);
```

AVX (256-bit)

```
__m256i a = _mm256_set_epi64(0, 1, 2, 3);
__m256i b = _mm256_set_epi64(3, 2, 1, 0);
__m256i d = _mm256_sub_epi64(a, b);
```

VSX

```
vector uint64_t a = {0, 1};
vector uint64_t b = {1, 0};
vector uint64_t d = vec_sub(a, b);
```

4.3.5 32-bit floats

TODO: insert diagram for 32-bit float vector subtraction

C Intrinsics

SIMD	Code	Input	Output
MMX(64-bit)[28]	N/A	–	–
SSE(64-bit)	`d = _mm_sub_ss(a, b)`	`__m128 a, b;`	`__m128 d;`
SSE(128-bit)	`d = _mm_sub_ps(a, b)`	`__m128 a, b;`	`__m128 d;`
AVX(256-bit)	`d = _mm256_sub_ps(a, b)`	`__m256 a, b;`	`__m256 d;`
NEON(64-bit)	`d = vsub_f32(a, b)`	`float32x2_t a, b;`	`float32x2_t d;`
NEON(128-bit)	`d = vsubq_f32(a, b)`	`float32x4_t a, b;`	`float32x4_t d;`
AltiVec/VMX	`d = vec_sub(a, b)`	`vector float a, b;`	`vector float d;`
VSX[29]	`d = vec_sub(a, b)`	`vector float a, b;`	`vector float d;`

[28] MMX does not support float arithmetic
[29] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
MMX(64-bit)	N/A	–	–
SSE(64-bit)	`subss xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit double registers, but the operation takes place only at the lower 64-bits
SSE(128-bit)	`subps xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit float registers
AVX(128-bit)	`subps xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`subps ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(64-bit)	`vsub.f32 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vsubq.f32 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`fsub Vd.2S, Vn.2S, Vm.2S`	Subtracts Vn.2S, Vm.2S, result in Vd.2S	Vn.2S, Vm.2S, Vd.2S are 64-bit NEON registers
NEON(armv8, 128-bit)	`fsub Vd.4S, Vn.4S, Vm.4S`	Subtracts Vn.4S, Vm.4S, result in Vd.4S	Vn.4S, Vm.4S, Vd.4S are 128-bit NEON registers
AltiVec/VMX	`vsubfp vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubfp vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction
VSX	`xssubsp vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are VSX registers holding double values, but the operation takes place only at the lower 64-bits (FIXME: better description)

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128 a = _mm_set_ps(1.0, 2.0, 3.0, 4.0);
__m128 b = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
__m128 d = _mm_sub_ps(a, b);
```

AVX (256-bit)

```
__m256 a = _mm256_set_ps(1.0, 2.0, 3.0, 4.0);
__m256 b = _mm256_set_ps(4.0, 3.0, 2.0, 1.0);
__m256 d = _mm256_sub_ps(a, b);
```

NEON (128-bit)

```
float32x4_t a = {0, 1, 2, 3};
float32x4_t b = {3, 2, 1, 0};
float32x4_t d = vsubq_f32(a, b);
```

AltiVec/VMX

```
vector float a = {0, 1, 2, 3};
vector float b = {3, 2, 1, 0};
vector float d = vec_sub(a, b);
```

4.3.6 64-bit doubles

TODO: insert diagram for 64-bit float vector subtraction

C Intrinsics

SIMD	Code	Input	Output
MMX(64-bit)[30]	N/A	–	–
SSE2(64-bit)	`d = _mm_sub_sd(a, b)`	`__m128d a, b;`	`__m128d d;`
SSE2(128-bit)	`d = _mm_sub_pd(a, b)`	`__m128d a, b;`	`__m128d d;`
AVX(256-bit)	`d = _mm256_sub_pd(a, b)`	`__m256d a, b;`	`__m256d d;`
NEON(64-bit)[31]	`d = vadd_f64(a, b)`	`float64x1_t a, b;`	`float64x1_t d;`
NEON(128-bit)	`d = vaddq_f64(a, b)`	`float64x2_t a, b;`	`float64x2_t d;`
AltiVec/VMX[32]	N/A	–	–
VSX	`d = vec_sub(a, b)`	`vector double a, b;`	`vector double d;`

[30] MMX does not support double arithmetic
[31] double arithmetic is supported only by armv8 NEON
[32] AltiVec/VMX does not support double arithmetic

Assembly

SIMD	Code	Description	Notes
MMX(64-bit)	N/A	–	–
SSE2(64-bit)	`subsd xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit double registers, but the operation takes place only at the lower 64-bits
SSE2(128-bit)	`subpd xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit float registers
AVX(128-bit)	`subpd xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`subpd ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	N/A	–	–
NEON(armv7, 128-bit)	N/A	–	–
NEON(armv8, 64-bit)	`fadd Dd, Dn, Dm`	Subtracts Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`fadd Vd.2D, Vn.2D, Vm.2D`	Subtracts Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
AltiVec/VMX	N/A	–	–
VSX	`xssubdp vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are VSX registers holding double values, but the operation takes place only at the lower 64-bits
VSX	`xvsubdp vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are VSX registers holding double values

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128d a = _mm_set_pd(1.0, 2.0, 3.0, 4.0);
__m128d b = _mm_set_pd(4.0, 3.0, 2.0, 1.0);
__m128d d = _mm_sub_pd(a, b);
```

AVX (256-bit)

```
__m256d a = _mm256_set_pd(1.0, 2.0, 3.0, 4.0);
__m256d b = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
__m256d d = _mm256_sub_pd(a, b);
```

NEON (128-bit)

```
float64x2_t a = {0, 1};
float64x2_t b = {3, 2};
float64x2_t d = vsubq_f64(a, b);
```

VSX

```
vector double a = {0, 1, 2, 3};
vector double b = {3, 2, 1, 0};
vector double d = vec_sub(a, b);
```

4.4 Subtraction, saturated

Saturated subtraction in contrast to modulo addition, keeps the result in a fixed range between a minimum and maximum value. If it's greater than the maximum value, then it's "clamped" to the maximum value, if it's below the minimum then it's set to the minimum value. In case of 8-bit unsigned bytes (unsigned char), adding 10 to 255 using saturated addition would produce the result 255.

4.4.1 8-bit Integers

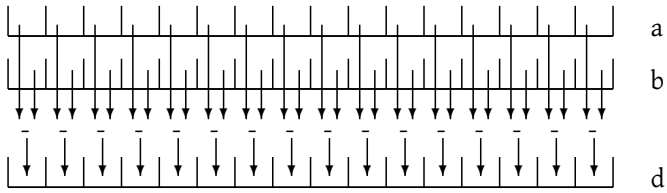

Figure 4.4: Addition of 16 8-bit integer elements in 128-bit vectors

C Intrinsics

SIMD	Code	Input	Output
8-bit unsigned			
MMX(64-bit)	`d = _mm_subs_pu8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_subs_epu8(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_subs_epu8(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vsubs_u8(a, b)`	`uint8x8_t a, b;`	`uint8x8_t d;`
NEON(128-bit)	`d = vsubsq_u8(a, b)`	`uint8x16_t a, b;`	`uint8x16_t d;`
AltiVec/VMX	`d = vec_subs(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
VSX[33]	`d = vec_subs(a, b)`	`vector uint8_t a, b;`	`vector uint8_t d;`
8-bit signed			
MMX(64-bit)	`d = _mm_subs_pi8(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_subs_epi8(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_subs_epi8(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vqsub_s8(a, b)`	`int8x8_t a, b;`	`int8x8_t d;`
NEON(128-bit)	`d = vqsubq_s8(a, b)`	`int8x16_t a, b;`	`int8x16_t d;`
AltiVec/VMX	`d = vec_subs(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`
VSX[34]	`d = vec_subs(a, b)`	`vector int8_t a, b;`	`vector int8_t d;`

[33] the VMX instruction will be used
[34] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
8-bit unsigned			
MMX(64-bit)	`psubusb mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubusb xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubusb xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubusb ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vqsub.i8 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqsubq.i8 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqsub Vd.8B, Vn.8B, Vm.8B`	Subtracts Vn.8B, Vm.8B, result in Vd.8B	d1,d2,d3 are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqsub Vd.16B, Vn.16B, Vm.16B`	Subtracts Vn.16B, Vm.16B, result in Vd.16B	Vn.16B, Vm.16B, Vd.16B are 128-bit NEON registers
AltiVec/VMX	`vsububs vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsububs vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction
8-bit signed			
MMX(64-bit)	`psubsb mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubsb xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubsb xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubsb ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(64-bit)	`vqsub.s8 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vqsubq.s8 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqsub Vd.8B, Vn.8B, Vm.8B`	Subtracts Vn.8B, Vm.8B, result in Vd.8B	d1,d2,d3 are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqsub Vd.16B, Vn.16B, Vm.16B`	Subtracts Vn.16B, Vm.16B, result in Vd.16B	Vn.16B, Vm.16B, Vd.16B are 128-bit NEON registers
AltiVec/VMX	`vsubsbs vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubsbs vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epu8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15);
__m128i b = _mm_set_epu8(15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_subs_epu8(a, b);
```

NEON (128-bit)

```
int8x16_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
int8x16_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int8x16_t d = vqsubq_s8(a, b);
```

AltiVec/VMX

```
vector uint8_t a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
vector uint8_t b = {15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
vector uint8_t d = vec_subs(a, b);
```

4.4.2 16-bit integers

TODO: insert diagram for 16-bit saturated vector addition

C Intrinsics

SIMD	Code	Input	Output
16-bit unsigned			
MMX(64-bit)	`d = _mm_subs_pu16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_subs_epu16(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_subs_epu16(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vqsub_u16(a, b)`	`uint16x4_t a, b;`	`uint16x4_t d;`
NEON(128-bit)	`d = vqsubq_u16(a, b)`	`uint16x8_t a, b;`	`uint16x8_t d;`
AltiVec/VMX	`d = vec_subs(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
VSX[35]	`d = vec_adds(a, b)`	`vector uint16_t a, b;`	`vector uint16_t d;`
16-bit signed			
MMX(64-bit)	`d = _mm_subs_pi16(a, b)`	`__m64 a, b;`	`__m64 d;`
SSE2(128-bit)	`d = _mm_subs_epi16(a, b)`	`__m128i a, b;`	`__m128i d;`
AVX(256-bit)	`d = _mm256_subs_epi16(a, b)`	`__m256i a, b;`	`__m256i d;`
NEON(64-bit)	`d = vqsub_s16(a, b)`	`int16x4_t a, b;`	`int16x4_t d;`
NEON(128-bit)	`d = vqsubq_s16(a, b)`	`int16x8_t a, b;`	`int16x8_t d;`
AltiVec/VMX	`d = vec_subs(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`
VSX[36]	`d = vec_subs(a, b)`	`vector int16_t a, b;`	`vector int16_t d;`

[35] the VMX instruction will be used
[36] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
16-bit unsigned			
MMX(64-bit)	`psubsw mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubsw xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubsw xmm1, xmm2, xmm3`	Subtractss xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubsw ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vqsub.i16 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqsubq.i16 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqsub Vd.4H, Vn.4H, Vm.4H`	Subtracts Vn.4H, Vm.4H, result in Vd.4H	Vn.4H, Vm.4H, Vd.4H are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqsub Vd.8H, Vn.8H, Vm.8H`	Subtracts Vn.8H, Vm.8H, result in Vd.8H	Vn.8H, Vm.8H, Vd.8H are 128-bit NEON registers
AltiVec/VMX	`vsubuhm vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubuhm vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction
16-bit signed			
MMX(64-bit)	`psubsw mm1, mm2`	Subtracts mm1, mm2, result in mm1	mm1, mm2 are 64-bit MMX registers
SSE2(128-bit)	`psubsw xmm1, xmm2`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vpsubsw xmm1, xmm2, xmm3`	Subtracts xmm2, xmm3, result xmm1	xmm1, xmm2, xmm3 are AVX 128-bit registers
AVX(256-bit)	`vpsubsw ymm1, ymm2, ymm3`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
NEON(armv7, 64-bit)	`vqsub.s16 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqsubq.s16 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqsub Vd.4H, Vn.4H, Vm.4H`	Subtracts Vn.4H, Vm.4H, result in Vd.4H	Vn.4H, Vm.4H, Vd.4H are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqsub Vd.8H, Vn.8H, Vm.8H`	Subtracts Vn.8H, Vm.8H, result in Vd.8H	Vn.8H, Vm.8H, Vd.8H are 128-bit NEON registers
AltiVec/VMX	`vsubshm vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubshm vD, vA, vB`	Subtracts vA, vB, result in vD	VMX instruction

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

SSE (128-bit)

```
__m128i a = _mm_set_epi16(0, 1, 2, 3, 4, 5, 6, 7);
__m128i b = _mm_set_epi16(7, 6, 5, 4, 3, 2, 1, 0);
__m128i d = _mm_subs_epi16(a, b);
```

NEON (128-bit)

```
int16x8_t a = {0, 1, 2, 3, 4, 5, 6, 7};
int16x8_t b = {7, 6, 5, 4, 3, 2, 1, 0};
int16x8_t d = vqsubq_s16(a, b);
```

AltiVec/VMX

```
vector uint16_t a = {0, 1, 2, 3, 4, 5, 6, 7};
vector uint16_t b = {7, 6, 5, 4, 3, 2, 1, 0};
vector uint16_t d = vec_subs(a, b);
```

4.4.3 32-bit integers

TODO: insert diagram for 32-bit integer saturated subtraction

C Intrinsics

SIMD	Code	Input	Output
32-bit unsigned			
MMX(64-bit)[37]	N/A	–	–
SSE2(128-bit)[38]	N/A	–	–
AVX(256-bit)[39]	N/A	–	–
NEON(64-bit)	`d = vqsub_u32(a, b)`	`uint32x2_t a, b;`	`uint32x2_t d;`
NEON(128-bit)	`d = vqsubq_u32(a, b)`	`uint32x4_t a, b;`	`uint32x4_t d;`
AltiVec/VMX	`d = vec_subs(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
VSX[40]	`d = vec_subs(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
32-bit signed			
MMX(64-bit)	N/A	–	–
SSE2(128-bit)	N/A	–	–
AVX(256-bit)	N/A	–	–
NEON(64-bit)	`d = vqsub_s32(a, b)`	`int32x2_t a, b;`	`int32x2_t d;`
NEON(128-bit)	`d = vqsubq_s32(a, b)`	`int32x4_t a, b;`	`int32x4_t d;`
AltiVec/VMX	`d = vec_subs(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`
VSX	`d = vec_subs(a, b)`	`vector int32_t a, b;`	`vector int32_t d;`

[37] MMX has no support for saturated addition on 32-bit integers
[38] SSE has no support for saturated addition on 32-bit integers
[39] AVX has no support for saturated addition on 32-bit integers
[40] the VMX instruction will be used

Assembly

SIMD	Code	Description	Notes
32-bit unsigned			
MMX(64-bit)	N/A	–	–
SSE2(128-bit)	N/A	–	–
AVX(128-bit)	N/A	–	–
AVX(256-bit)	N/A	–	–
NEON(armv7, 64-bit)	`vqsub.i32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqsubq.i32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqsub Vd.2S, Vn.2S, Vm.2S`	Subtracts Vn.2S, Vm.2S, result in Vd.2S	Vn.2S, Vm.2S, Vd.2S are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqsub Vd.4S, Vn.4S, Vm.4S`	Subtracts Vn.4S, Vm.4S, result in Vd.4S	Vn.4S, Vm.4S, Vd.4S are 128-bit NEON registers
AltiVec/VMX	`vsubuws vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubuws vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction
32-bit signed			
MMX(64-bit)	N/A	–	–
SSE2(128-bit)	N/A	–	–
AVX(128-bit)	N/A	–	–
AVX(256-bit)	N/A	–	–
NEON(armv7, 64-bit)	`vqsub.s32 d1, d2, d3`	Adds d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqsubq.s32 q1, q2, q3`	Adds q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqsub Vd.2S, Vn.2S, Vm.2S`	Subtracts Vn.2S, Vm.2S, result in Vd.2S	Vn.2S, Vm.2S, Vd.2S are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqsub Vd.4S, Vn.4S, Vm.4S`	Subtracts Vn.4S, Vm.4S, result in Vd.4S	Vn.4S, Vm.4S, Vd.4S are 128-bit NEON registers
AltiVec/VMX	`vsubsws vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubsws vD, vA, vB`	Adds vA, vB, result in vD	VMX instruction

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

NEON (128-bit)

```
uint32x4_t a = {0, 1, 2, 3};
```

```
uint32x4_t b = {3, 2, 1, 0};
uint32x4_t d = vqsubq_u32(a, b);
```

AltiVec/VMX

```
vector uint32_t a = {0, 1, 2, 3};
vector uint32_t b = {3, 2, 1, 0};
vector uint32_t d = vec_subs(a, b);
```

4.4.4 64-bit integers

At the moment the only architecture that supports 64-bit saturated subtraction is NEON, in both ARMv7 and ARMv8 implementations.

TODO: insert diagram for 64-bit vector saturated subtraction

C Intrinsics

SIMD	Code	Input	Output
64-bit unsigned			
NEON(armv7, 64-bit)	`d = vqsub_u64(a, b)`	`uint64x1_t a, b;`	`uint64x1_t d;`
NEON(armv7, 128-bit)	`d = vqsubq_u64(a, b)`	`uint64x2_t a, b;`	`uint64x2_t d;`
64-bit signed			
NEON(64-bit)	`d = vqsub_s64(a, b)`	`int64x1_t a, b;`	`int64x1_t d;`
NEON(128-bit)	`d = vqsubq_s64(a, b)`	`int64x2_t a, b;`	`int64x2_t d;`

Assembly

SIMD	Code	Description	Notes
64-bit unsigned			
NEON(armv7, 64-bit)	`vqsub.i64 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(armv7, 128-bit)	`vqsubq.i64 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`uqsub Dd, Dn, Dm`	Subtracts Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`uqsub Vd.2D, Vn.2D, Vm.2D`	Subtracts Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers
64-bit signed			
NEON(64-bit)	`vqsub.s64 d1, d2, d3`	Subtracts d2, d3, result in d1	d1,d2,d3 are 64-bit NEON registers
NEON(128-bit)	`vqsubq.s64 q1, q2, q3`	Subtracts q2, q3, result in q1	q1,q2,q3 are 128-bit NEON registers
NEON(armv8, 64-bit)	`sqsub Dd, Dn, Dm`	Subtracts Dn, Dm, result in Dd	Dn, Dm, Dd are 64-bit NEON registers
NEON(armv8, 128-bit)	`sqsub Vd.2D, Vn.2D, Vm.2D`	Subtracts Vn.2D, Vm.2D, result in Vd.2D	Vn.2D, Vm.2D, Vd.2D are 128-bit NEON registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

NEON (128-bit)

```
uint64x2_t a = {0, 1};
uint64x2_t b = {3, 2};
uint64x2_t d = vsubq_u64(a, b);
```

4.5 Special addition/subtraction instructions

4.5.1 Addition/subtraction with carry (32-bit integers, unsigned)

AltiVec/VSX also supports addition/subtraction with carry, but only for 32-bit/64-bit unsigned integers.

TODO: insert diagram for 32-bit vector addition/subtraction with carry

C Intrinsics

SIMD	Code	Input	Output
AltiVec/VMX/VSX	`d = vec_addc(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`
AltiVec/VMX/VSX	`d = vec_subc(a, b)`	`vector uint32_t a, b;`	`vector uint32_t d;`

Assembly

SIMD	Code	Description	Notes
AltiVec/VMX/VSX	`vaddcuw vD, vA, vB`	Adds vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
AltiVec/VMX/VSX	`vsubcuw vD, vA, vB`	Subtracts vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

AltiVec/VMX/VSX

```
vector uint32_t a = {0, 1, 2, 3};
vector uint32_t b = {3, 2, 1, 0};
vector uint32_t d = vec_addc(a, b);
vector uint32_t e = vec_subc(a, b);
```

4.5.2 Addition/subtraction with carry (64-bit integers, unsigned)

On par with AltiVec/VMX, 64-bit integer addition/subtraction with carry is supported only by its extended version VSX.

TODO: insert diagram for 64-bit vector addition

C Intrinsics

SIMD	Code	Input	Output
VSX	`d = vec_addc(a, b)`	`vector uint64_t a, b;`	`vector uint64_t d;`
VSX	`d = vec_subc(a, b)`	`vector uint64_t a, b;`	`vector uint64_t d;`

Assembly

SIMD	Code	Description	Notes
VSX	`vaddcuq vD, vA, vB`	Adds with carry vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers
VSX	`vsubcuq vD, vA, vB`	Subtracts with carry vA, vB, result in vD	vA, vB, vD are normal VMX 128-bit registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

VSX

```
vector uint64_t a = {0, 1};
vector uint64_t b = {3, 2};
vector uint64_t d = vec_addc(a, b);
vector uint64_t e = vec_subc(a, b);
```

4.5.3 Odd/Even Addition/subtraction (32-bit floats/64-bit doubles)

Odd/Even Addition/Subtraction is an SSE3-specific instruction (with the respective C intrinsic) that adds odd-numbered elements and subtracts even-numbered elements in the vector.

TODO: insert diagram for 32-bit float vector odd/even addition/subtraction

C Intrinsics

SIMD	Code	Input	Output
SSE3(128-bit)	`d = _mm_addsub_ps(a, b)`	`__m128 a, b;`	`__m128 d;`
SSE3(128-bit)	`d = _mm_addsub_pd(a, b)`	`__m128d a, b;`	`__m128d d;`

Assembly

SIMD	Code	Description	Notes
SSE3(128-bit)	`addsubps xmm1, xmm2/m128`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
SSE3(128-bit)	`addsubpd xmm1, xmm2/m128`	Subtracts xmm1, xmm2, result in xmm1	xmm1, xmm2 are SSE 128-bit registers
AVX(128-bit)	`vaddsubps xmm1, xmm2, xmm3/m128`	Subtractss xmm2, xmm3, result xmm1	xmm1, xmm2 are AVX 128-bit registers, xmm3 can be an AVX
AVX(256-bit)	`vaddsubps ymm1, ymm2, ymm3/m128`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers
AVX(128-bit)	`vaddsubpd xmm1, xmm2, xmm3/m128`	Subtractss xmm2, xmm3, result xmm1	xmm1, xmm2 are AVX 128-bit registers, xmm3 can be an AVX
AVX(256-bit)	`vaddsubpd ymm1, ymm2, ymm3/m128`	Subtracts ymm2, ymm3, result in ymm1	ymm1, ymm2, ymm3 are AVX 256-bit registers

Notes

Examples

Here, we present some coding examples using the C intrinsics for some of the SIMD engines.

VSX

```
vector uint64_t a = {0, 1};
vector uint64_t b = {3, 2};
vector uint64_t d = vec_addc(a, b);
```

4.6 In-vector sum reduction

Many times one must do an in-vector sum reduction, that is, an addition of all elements in a vector, and the normal addition instructions won't do, and more sophisticated instructions are called for. Most SIMD engines provide for at least one of such instructions for at least one datatype, but a 1:1 mapping is not possible. However, since the current document is a Functionality Reference Guide (with emphasis on Functionality) we will provide not only the instruction where available, but also ways to duplicate the functionality in other SIMD engines where a single instruction is missing.

4.6.1 Integer Sum across Partial 1/4 saturated

TODO: insert diagram for 32-bit vector partial 1/4 sum reduction

Index