

Load/Store	Arithmetic	In-register movement	Data	Other
store[u]	add	blend[v]		cmp
load[u]	mul	shuffle		and
maskstore	div	permute		or
maskload	rcp	permute2f128		...
stream	dp	insertf128		
...	hadd	extractf128		
	...	movehdup		
		moveldup		
		unpackhi		
		unpacklo		
		castps256_ps128		
		castps128_ps256		
		...		

```
void _mm256_store_ps (float * mem_addr, __m256 a)
MEM[mem_addr+255:mem_addr] := a[255:0]
```

- MEM must be aligned to 32byte

```
void _mm256_maskstore_ps ( float * mem_addr,
                           __m256i mask,
                           __m256 a)
FOR j := 0 to 7
  i := j*32
  IF mask[i+31]
    MEM[mem_addr+i+31:mem_addr+i] := a[i+31:i]
  FI
ENDFOR
```

```
void _mm256_storeu_ps (float * mem_addr, __m256 a)
MEM[mem_addr+255:mem_addr] := a[255:0]
```

- MEM does not need to be aligned

```
void _mm256_maskstore_ps ( float * mem_addr,
                          __m256i mask,
                          __m256 a)
FOR j := 0 to 7
  i := j*32
  IF mask[i+31]
    MEM[mem_addr+i+31:mem_addr+i] := a[i+31:i]
  FI
ENDFOR
```

```
__m256 mm256_load_ps (float const * mem_addr)
```

```
dst[127:0] := MEM[mem_addr+127:mem_addr]
```

- MEM must be aligned to 32byte

```
__m128 mm_maskload_ps (float const * mem_addr, __m128i mask)
```

```
FOR j := 0 to 3
```

```
    i := j*32
```

```
    IF mask[i+31]
```

```
        dst[i+31:i] := MEM[mem_addr+i+31:mem_addr+i]
```

```
    ELSE
```

```
        dst[i+31:i] := 0
```

```
    FI
```

```
ENDFOR
```

```
__m256 mm256_loadu_ps (float const * mem_addr)  
dst[127:0] := MEM[mem_addr+127:mem_addr]
```

- MEM does not need to be aligned

```
__m128 mm_maskload_ps (float const * mem_addr, __m128i mask)  
FOR j := 0 to 3  
  i := j*32  
  IF mask[i+31]  
    dst[i+31:i] := MEM[mem_addr+i+31:mem_addr+i]  
  ELSE  
    dst[i+31:i] := 0  
  FI  
ENDFOR
```

```
_m256 mm256_rcp_ps (_m256 a)
```

```
FOR j := 0 to 7
```

```
  i := j*32
```

```
  dst[i+31:i] := APPROXIMATE(1.0/a[i+31:i])
```

```
ENDFOR
```

- Higher throughput than ordinary division

```
_m256 mm256_hadd_ps (_m256 a, _m256 b)
```

```
dst[31:0] := a[63:32] + a[31:0]
```

```
dst[63:32] := a[127:96] + a[95:64]
```

```
dst[95:64] := b[63:32] + b[31:0]
```

```
dst[127:96] := b[127:96] + b[95:64]
```

```
dst[159:128] := a[191:160] + a[159:128]
```

```
dst[191:160] := a[255:224] + a[223:192]
```

```
dst[223:192] := b[191:160] + b[159:128]
```

```
dst[255:224] := b[255:224] + b[223:192]
```

```
_m256 _mm256_dp_ps (_m256 a, _m256 b, const int imm)
```

```
DP(a[127:0], b[127:0], imm[7:0]) {  
    FOR j := 0 to 3  
        i := j*32  
        IF imm[4+j]  
            temp[i+31:i] := a[i+31:i] * b[i+31:i]  
        ELSE  
            temp[i+31:i] := 0  
        FI  
    ENDFOR  
  
    sum[31:0] := (temp[127:96] + temp[95:64]) + (temp[63:32] + temp[31:0])  
  
    FOR j := 0 to 3  
        i := j*32  
        IF imm[j]  
            tmpdst[i+31:i] := sum[31:0]  
        ELSE  
            tmpdst[i+31:i] := 0  
        FI  
    ENDFOR  
    RETURN tmpdst[127:0]  
}  
  
dst[127:0] := DP(a[127:0], b[127:0], imm[7:0])  
dst[255:128] := DP(a[255:128], b[255:128], imm[7:0])
```

• Dot-Product

```
__m256 _mm256_blend_ps (__m256 a, __m256 b, const int imm)
```

```
FOR j := 0 to 7  
  i := j*32  
  IF imm[j]  
    dst[i+31:i] := b[i+31:i]  
  ELSE  
    dst[i+31:i] := a[i+31:i]  
  FI  
ENDFOR
```



```
_m256 _mm256_permute2f128_ps (_m256 a, _m256 b, int imm)
```

```
SELECT4(src1, src2, control){
```

```
    CASE(control[1:0])
```

```
    0: tmp[127:0] := src1[127:0]
```

```
    1: tmp[127:0] := src1[255:128]
```

```
    2: tmp[127:0] := src2[127:0]
```

```
    3: tmp[127:0] := src2[255:128]
```

```
    ESAC
```

```
    IF control[3]
```

```
        tmp[127:0] := 0
```

```
    FI
```

```
    RETURN tmp[127:0]
```

```
}
```

```
dst[127:0] := SELECT4(a[255:0], b[255:0], imm[3:0])
```

```
dst[255:128] := SELECT4(a[255:0], b[255:0], imm[7:4])
```

```
__m256 mm256_insertf128_ps (__m256 a, __m128 b, int imm)
```

```
dst[255:0] := a[255:0]
```

```
CASE (imm[1:0]) of
```

```
0: dst[127:0] := b[127:0]
```

```
1: dst[255:128] := b[127:0]
```

```
ESAC
```

```
__m128 mm256_extractf128_ps (__m256 a, const int imm)
```

```
CASE imm of
```

```
0: dst[127:0] := a[127:0]
```

```
1: dst[127:0] := a[255:128]
```

```
ESAC
```

```
__m256 _mm256_movehdup_ps (__m256 a)
```

```
dst[31:0] := a[63:32]
```

```
dst[63:32] := a[63:32]
```

```
dst[95:64] := a[127:96]
```

```
dst[127:96] := a[127:96]
```

```
dst[159:128] := a[191:160]
```

```
dst[191:160] := a[191:160]
```

```
dst[223:192] := a[255:224]
```

```
dst[255:224] := a[255:224]
```

```
__m256 mm256_moveldup_ps (__m256 a)
```

```
dst[31:0] := a[31:0]
```

```
dst[63:32] := a[31:0]
```

```
dst[95:64] := a[95:64]
```

```
dst[127:96] := a[95:64]
```

```
dst[159:128] := a[159:128]
```

```
dst[191:160] := a[159:128]
```

```
dst[223:192] := a[223:192]
```

```
dst[255:224] := a[223:192]
```

```
_m256 mm256_unpackhi_ps (_m256 a, _m256 b)
```

```
INTERLEAVE_HIGH_DWORDS(src1[127:0], src2[127:0]){  
    dst[31:0] := src1[95:64]  
    dst[63:32] := src2[95:64]  
    dst[95:64] := src1[127:96]  
    dst[127:96] := src2[127:96]  
    RETURN dst[127:0]  
}
```

```
dst[127:0] := INTERLEAVE_HIGH_DWORDS(a[127:0], b[127:0])  
dst[255:128] := INTERLEAVE_HIGH_DWORDS(a[255:128], b[255:128])
```

```
_mm256_mm256_unpacklo_ps (_mm256 a, _mm256 b)
```

```
INTERLEAVE_DWORDS(src1[127:0], src2[127:0]){
```

```
    dst[31:0] := src1[31:0]
```

```
    dst[63:32] := src2[31:0]
```

```
    dst[95:64] := src1[63:32]
```

```
    dst[127:96] := src2[63:32]
```

```
    RETURN dst[127:0]
```

```
}
```

```
dst[127:0] := INTERLEAVE_DWORDS(a[127:0], b[127:0])
```

```
dst[255:128] := INTERLEAVE_DWORDS(a[255:128], b[255:128])
```

```
dst[MAX:256] := 0
```

```
_m256 _mm256_permute_ps (_m256 a, int imm)
```

```
SELECT4(src, control){  
    CASE(control[1:0])  
    0: tmp[31:0] := src[31:0]  
    1: tmp[31:0] := src[63:32]  
    2: tmp[31:0] := src[95:64]  
    3: tmp[31:0] := src[127:96]  
    ESAC  
    RETURN tmp[31:0]  
}  
  
dst[31:0] := SELECT4(a[127:0], imm[1:0])  
dst[63:32] := SELECT4(a[127:0], imm[3:2])  
dst[95:64] := SELECT4(a[127:0], imm[5:4])  
dst[127:96] := SELECT4(a[127:0], imm[7:6])  
dst[159:128] := SELECT4(a[255:128], imm[1:0])  
dst[191:160] := SELECT4(a[255:128], imm[3:2])  
dst[223:192] := SELECT4(a[255:128], imm[5:4])  
dst[255:224] := SELECT4(a[255:128], imm[7:6])
```

- Also allows to duplicate entries
- Separation between lower and upper 128bit

```
__m256 mm256_shuffle_ps (__m256 a, __m256 b, const int imm)
```

```
SELECT4(src, control){  
    CASE(control[1:0])  
    0: tmp[31:0] := src[31:0]  
    1: tmp[31:0] := src[63:32]  
    2: tmp[31:0] := src[95:64]  
    3: tmp[31:0] := src[127:96]  
    ESAC  
    RETURN tmp[31:0]  
}  
  
dst[31:0] := SELECT4(a[127:0], imm[1:0])  
dst[63:32] := SELECT4(a[127:0], imm[3:2])  
dst[95:64] := SELECT4(b[127:0], imm[5:4])  
dst[127:96] := SELECT4(b[127:0], imm[7:6])  
dst[159:128] := SELECT4(a[255:128], imm[1:0])  
dst[191:160] := SELECT4(a[255:128], imm[3:2])  
dst[223:192] := SELECT4(b[255:128], imm[5:4])  
dst[255:224] := SELECT4(b[255:128], imm[7:6])
```



```
__m256 mm256_cmp_ps (__m256 a, __m256 b, const int imm)
FOR j := 0 to 7
  i := j*32
  dst[i+31:i] := ( a[i+31:i] OP b[i+31:i] ) ? 0xFFFFFFFF : 0
ENDFOR
```

```
__m256 _mm256_and_ps (__m256 a, __m256 b)  
FOR j := 0 to 7  
  i := j*32  
  dst[i+31:i] := (a[i+31:i] AND b[i+31:i])  
ENDFOR
```

```
__m256 _mm256_or_ps (__m256 a, __m256 b)  
FOR j := 0 to 7  
  i := j*32  
  dst[i+31:i] := a[i+31:i] BITWISE OR b[i+31:i]  
ENDFOR
```

- Alternative for blending