SSE architecture

A Detailed Overview

Part 1

February 2008
Sverre Jarp
CERN IT Department

Two natural parts

- Part One:
  - Floating-point support
    - Single/Double
- Part Two
  - Integer support
    - Bytes/Words/DWords/QWords

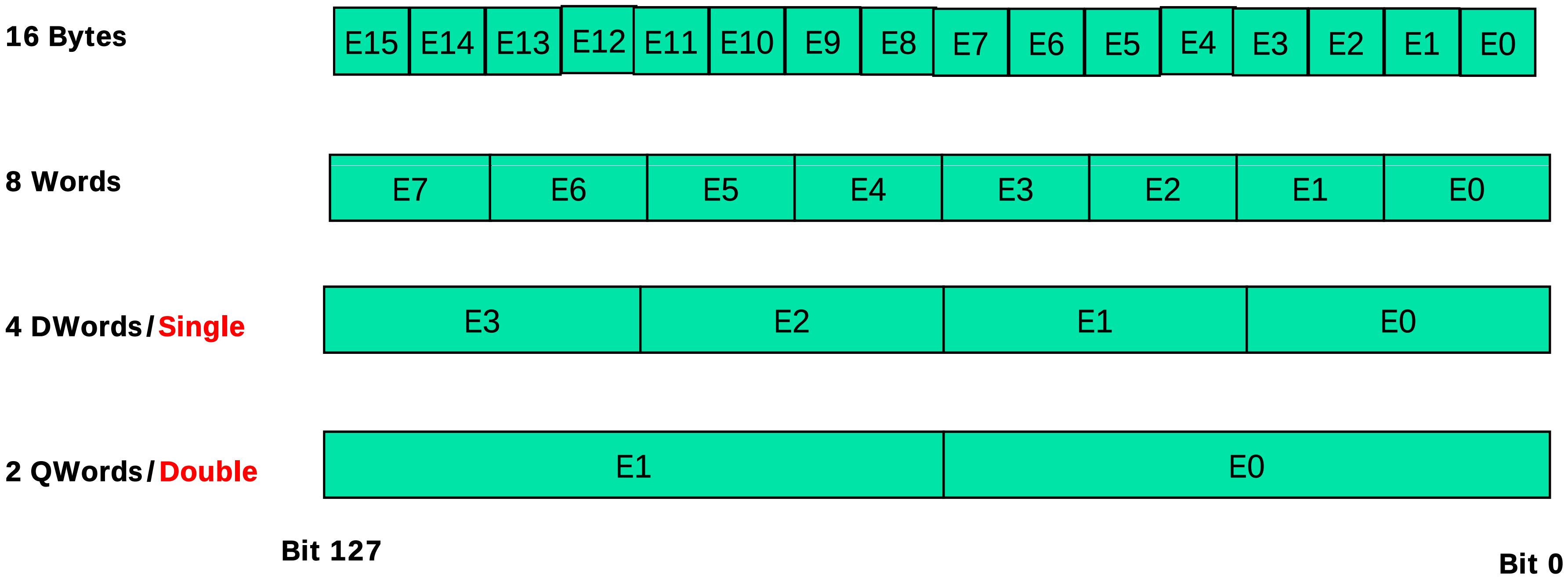
Part 1:
FLOATING-POINT

Aims

- Group instructions to allow a good understanding of
  - Which operations can be done on what data
- E.g. Can I do a multiply-and-add of double precision numbers with a single instruction?
  - d = a * b + c
- Show examples of practical use

XMM Registers

- 16 registers with 128 bits each
  - (only 8 registers in 32-bit mode)



16 Bytes: E15 E14 E13 E12 E11 E10 E9 E8 E7 E6 E5 E4 E3 E2 E1 E0

8 Words: E7 E6 E5 E4 E3 E2 E1 E0

4 DWords/**Single**: E3 E2 E1 E0

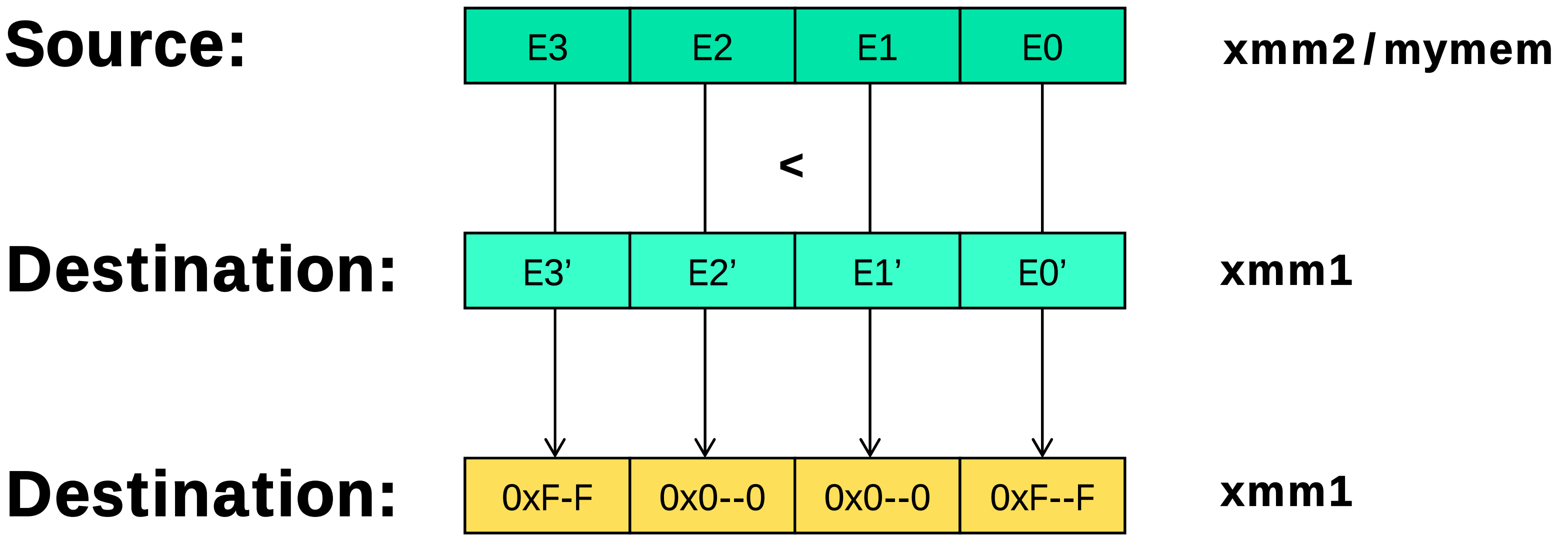
2 QWords/**Double**: E1 E0

Bit 127

Bit 0

Typical instruction format

- Unary/Binary:
  - Mnemonic   Destination, Source [,Immediate]
  - Mnemonic: Verb + Data Types
    - Example:
    - CMPPS xmm1, xmm2/mymem, LT



Source: | E3 | E2 | E1 | E0 | xmm2/mymem

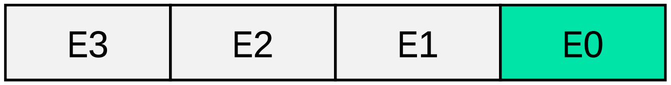
<

Destination: | E3’ | E2’ | E1’ | E0’ | xmm1

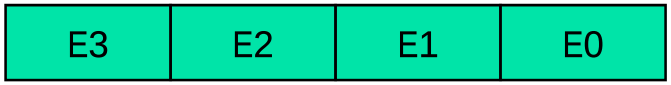
Destination: | 0xF-F | 0x0--0 | 0x0--0 | 0xF--F | xmm1

Four data flavours

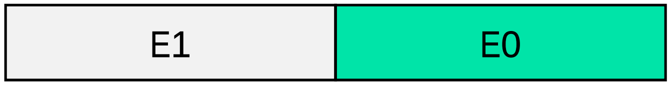
- Single precision
  - Scalar Single (SS)



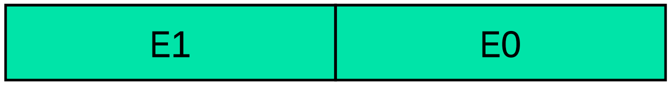
  - Packed Single (PS)



- Double precision
  - Scalar Double (SD)



  - Packed Double (PD)

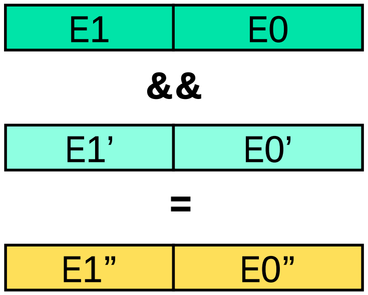


- Scalar instructions replace x87 in Intel-64!

FLP logical operations

	Mnemonic	SS	SD	PS	PD
And	AND	--	--	0f 54	66 0f 54
And not	ANDN	--	--	0f 55	66 0f 55
Or	OR	--	--	0f 56	66 0f 56
Xor	XOR	--	--	0f 57	66 0f 57

- For example:
  - andpd



E1 E0

&&

E1’ E0’

=

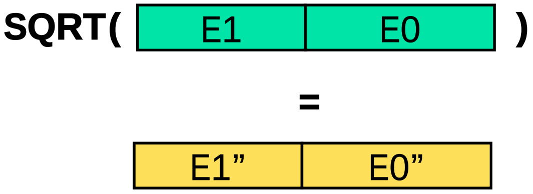
E1” E0”

FLP standard arithmetic (1)

	Mnemonic	SS	SD	PS	PD
Square root	SQRT	f3 0f 51	f2 0f 51	0f 51	66 0f 51
Normal add	ADD	f3 0f 58	f2 0f 58	0f 58	66 0f 58
Multiplication	MUL	f3 0f 59	f2 0f 59	0f 59	66 0f 59
Normal subtract	SUB	f3 0f 5c	f2 0f 5c	0f 5c	66 0f 5c
Division	DIV	f3 0f 5e	f2 0f 5e	0f 5e	66 0f 5e
Horizontal add	HADD	--	--	f2 0f 7c	66 0f 7c
Horizontal subtract	HSUB	--	--	f2 0f 7d	66 0f 7d
Add and subtract	ADDSUB	--	--	f2 0f d0	66 0f d0
Dot Product	DP	--	--	66 0f 3a 41	66 0f 3a 40
Reciprocal approximation	RCP	f3 0f 53	--	0f 53	--
Reciprocal SQRT approx.	RSQRT	f3 0f 52	--	0f 52	--

FLP standard arithmetic (2)

- Two examples:
  - SQRTPD dest, source
    - Put SQRT of source in destination

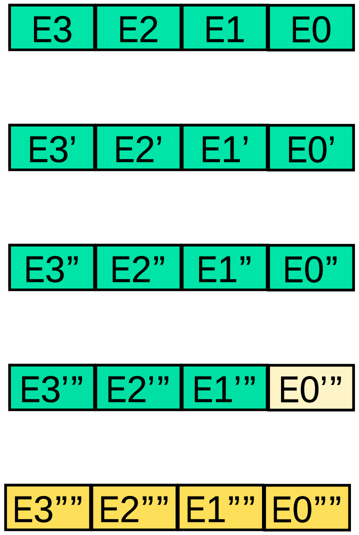


SQRT(E1 | E0)

=

E1” | E0”

  - DOTPS dest, source, imm8
    - Perform 4 dot products or deliver zero
      - According to IMM8[7:4]
    - Add horizontally
    - Broadcast result
      - According to IMM8[3:0]



E3 | E2 | E1 | E0

E3’ | E2’ | E1’ | E0’

E3” | E2” | E1” | E0”

E3’” | E2’” | E1’” | E0’”

E3”” | E2”” | E1”” | E0””

FLP Round

- Round FLP results to an Integral Value:
  - E.g. ROUNDPS xmm2, xmm3/mem, IMM8
    - IMM8:
      - Rounding modes:
        - Nearest (0), Down (1), Up (2), Truncate (3)
      - Rounding Select
      - Precision Mask

	Mnemonic	SS	SD	PS	PD
Round	ROUND	66 0f 3a 0a	66 0f 3a 0b	66 0f 3a 08	66 0f 3a 09

FLP Minimum/Maximum

- Return the lower/higher value after comparison:
  - E.g. MAXPS xmm1, xmm2/mem

	Mnemonic	SS	SD	PS	PD
Minimum	MIN	f3 0f 5d	f2 0f 5d	0f 5d	66 0f 5d
Maximum	MAX	f3 0f 5f	f2 0f 5f	0f 5f	66 0f 5f

FLP comparisons (1)

- Example:
  - CMPPS xmm1, xmm2, LT

	E3	E2	E1	E0	**xmm2**
		LT			
Compare:	E3’	E2’	E1’	E0’	**xmm1**
		=			
Generated masks:	0xF-F	0x0--0	0x0--0	0xF--F	**xmm1**

FLP comparisons (2)

	Imm.	Mnemonic	SS	SD	PS	PD
Compare equal	0	CMP EQ	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare less than	1	CMP LT	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare less or equal	2	CMP LE	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare unordered	3	CMP UNORD	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare not equal	4	CMP NEQ	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare not less than	5	CMP NLT	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare not less or equal	6	CMP NLE	f3 0f c2	f2 0f c2	0f c2	66 0f c2
Compare ordered	7	CMP ORD	f3 0f c2	f2 0f c2	0f c2	66 0f c2

FLP comparisons (3)

- Compare Scalar and set EFLAGS
  - Example: COMISD xmm1, xmm2/mymem
- Sets ZF, PF, and CF in EFLAGS register
  - Allowing conditional JMP to follow

	Imm.	Mnemonic	SS	SD	PS	PD
Compare ordered and set EFLAGS	0	COMI	0f 2f	66 0f 2f	- -	- -

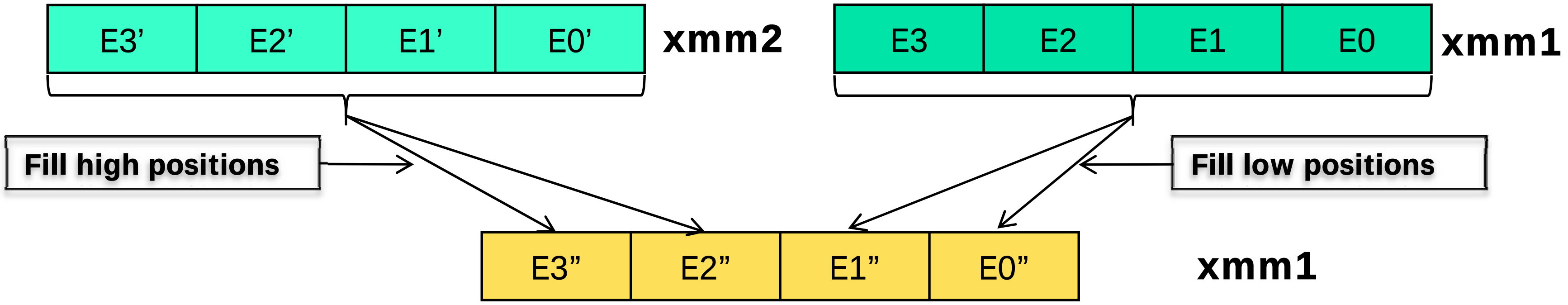
Shuffle

- SHUFxx xmm1, xmm2/mem, IMM8

	Mnemonic	SS	SD	PS	PD
Shuffle	SHUF	--	--	0f c6	66 0f c6

- Example: SHUFPS

Wanted in chunk:	3	2	1	0	Total (hex)
Quaternary:	0-3	0-3	0-3	0-3	0x0-0xff



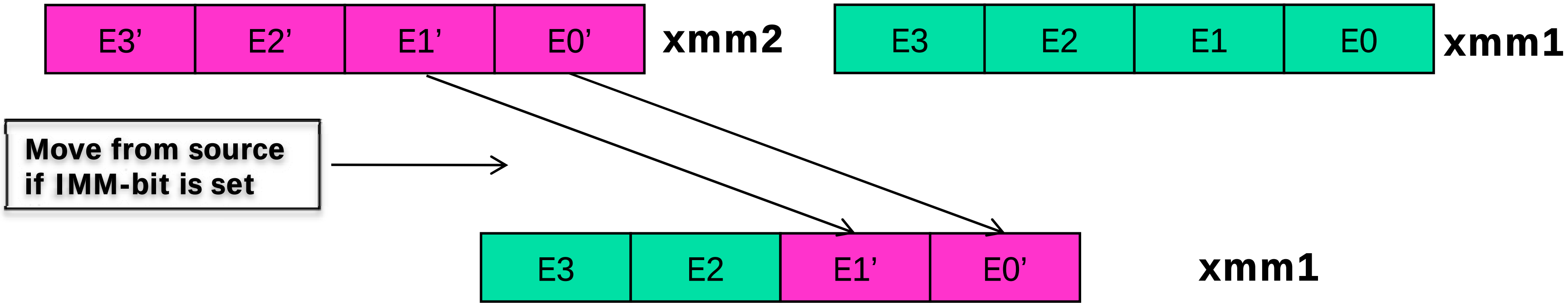
Blend (Move conditionally)

- BLENDxx xmm1, xmm2/mem, IMM8

	Mnemonic	SS	SD	PS	PD
Move	BLEND	--	--	66 0f 3a 0c	66 0f 3a 0d

- Example: BLENDPS

Move source:	3	2	1	0	Total (hex)
Binary:	0	0	1	1	0x3



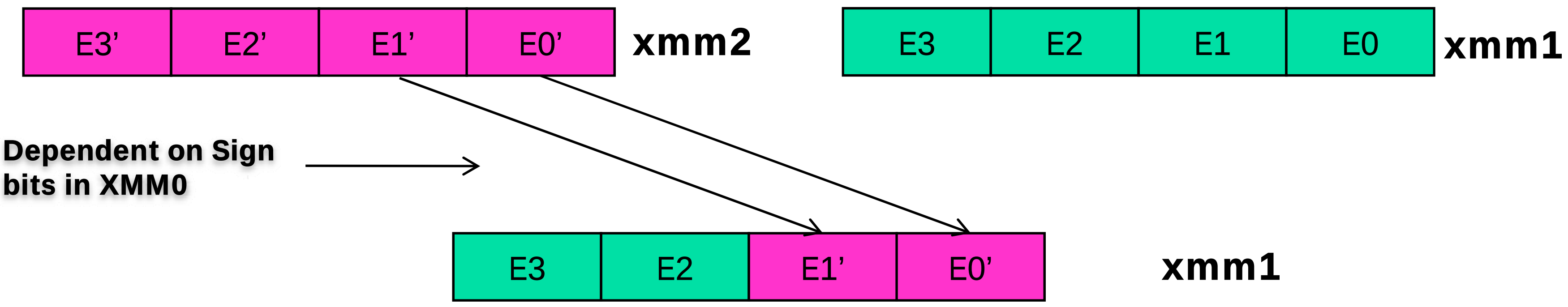
Move from source if IMM-bit is set

Blend-V (Move conditionally)

- BLENDVxx xmm1, xmm2/mem, <XMM0>

	Mnemonic	SS	SD	PS	PD
Move	BLEND	--	--	66 0f 38 14	66 0f 38 15

- Example: BLENDVPS



Dependent on Sign bits in XMM0

FLP move operations (1)

- Different op.codes for memory source (load) and memory target (store)
- Any difference between MOVSD and MOVLPD ?

	Mnemonic	SS	SD	PS	PD
Move Aligned	MOVA	--	--	0f 28, 0f 29	66 0f 28, 66 0f 29
Move Unaligned	MOVU	--	--	0f 10, 0f 11	66 0f 10, 66 0f 11
Move (scalar)	MOV	f3 0f 10, f3 0f 11	f2 0f 10, f2 of 11	--	--
Move Low	MOVL	--	--	0f 12, 0f 13	66 0f 12, 66 0f 13
Move High	MOVH	--	--	0f 16, 0f 17	66 0f 16, 66 0f 17
Store w/Non-temporal hint	MOVNT	f3 0f 2b	f2 0f 2b	0f 2b	66 0f 2b

Other move instructions in appendix

Autovectorization w/icc (1)

- Source code:

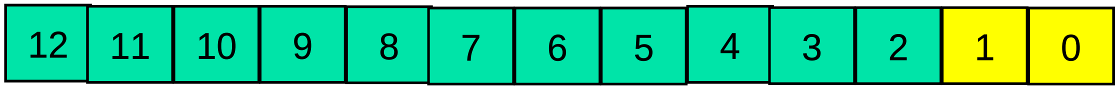
```
#ifdef FLOAT
bool test(float* d)
#else
bool test(double* d)
#endif
#ifdef SPACING
        { for (int i =0; i<620; i+=4)
#else
        { for (int i =0; i<620; i++)
#endif
                {
#ifdef COMPARE
                if (d[i] > 0.75)
#endif
                d[i] = (d[i+4] + d[i])/2.0;
                }
      return 1;
}
```

SUCCESS !

- Compilation and object code:

```
 icc -O2 -S -unroll0 mysse.cxx
mysse.cxx(9): (col. 11) remark:
             LOOP WAS VECTORIZED.
```

```
..B1.5:
        movaps     32(%rdx), %xmm1
        addpd      (%rdx), %xmm1
        mulpd      %xmm0, %xmm1
        movaps     %xmm1, (%rdx)
        addq       $16, %rdx
        cmpq       %rax, %rcx
        jl         ..B1.5         # Prob 99%
```



2 * double

Autovectorization w/icc (2)

- Source code:

```
#ifdef FLOAT
bool test(float* d)
#else
bool test(double* d)
#endif
#ifdef SPACING
        { for (int i =0; i<620; i+=4)
#else
        { for (int i =0; i<620; i++)
#endif
                {
#ifdef COMPARE
                if (d[i] > 0.75)
#endif
                d[i] = (d[i+4] + d[i])/2.0;
                }
      return 1;
}
```

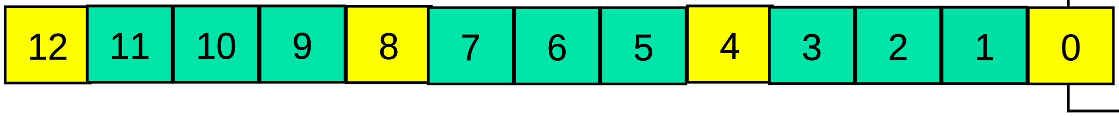
SUCCESS !

- Compilation and object code:

```
 icc -O2 -S -unroll0 –DSPACING mysse.cxx
mysse.cxx(7): (col. 11) remark:
              LOOP WAS VECTORIZED.
```

```
..B1.2:
        movsd     32(%rdx), %xmm2
        movsd     (%rdx), %xmm1
        movl      $4, %ecx
        movhpd    32(%rdx,%rcx,8), %xmm2
        movhpd    (%rdx,%rcx,8), %xmm1
        addpd     %xmm1, %xmm2
        mulpd     %xmm0, %xmm2
        movlpd    %xmm2, (%rdx)
        movhpd    %xmm2, (%rdx,%rcx,8)

        addq      $64, %rdx
        addl      $2, %eax
        cmpl      $154, %eax
        jb        ..B1.2        # Pro
```



12	11	10	9	8	7	6	5	4	3	2	1	0

Autovectorization w/icc (3)

- Source code:

```
#ifdef FLOAT
bool test(float* d)
#else
bool test(double* d)
#endif
#ifdef SPACING
        { for (int i =0; i<620; i+=4)
#else
        { for (int i =0; i<620; i++)
#endif
                {
#ifdef COMPARE
                if (d[i] > 0.75)
#endif
                d[i] = (d[i+4] + d[i])/2.0;
                }
      return 1;
}
```

- Compilation and object code:

IMPOSSIBILITY ?

```
icc -O2 -S -unroll0 –DCOMPARE mysse.cxx
```

```
..B1.2:
        movsd     (%rax,%rdi), %xmm1
        comisd    %xmm0, %xmm1
        jbe       ..B1.4

..B1.3:
        movsd     32(%rax,%rdi), %xmm2
        addsd     %xmm1, %xmm2
        mulsd     _2il0floatpacket.2(%rip), %xmm2
        movsd     %xmm2, (%rax,%rdi)

..B1.4:
        addq      $8, %rax
        cmpq      $4960, %rax
        jl        ..B1.2              # Prob 99%
```

12	11	10	9	8	7	6	5	4	?	2	1	0

Autovectorization w/icc (4)

- Source code:

```
#ifdef FLOAT
bool test(float* d)
#else
bool test(double* d)
#endif
#ifdef SPACING
        { for (int i =0; i<620; i+=4)
#else
        { for (int i =0; i<620; i++)
#endif
                {
#ifdef COMPARE
                if (d[i] > 0.75)
#endif
                d[i] = (d[i+4] + d[i])/2.0;
#ifdef COMPARE2
                else d[i] = (d[i+4] + d[i])*2.0;
#endif
                }
      return 1;
}
```

SANITY RESTORED!

- Compilation and object code:

```
  icc -O2 -S -unroll0 –DCOMPARE –DCOMPARE2
                       mysse.cxx
```

```
..B1.7:
        movaps _2il0floatpacket.1(%rip), %xmm2
        movaps _2il0floatpacket.2(%rip), %xmm3
        movaps    (%rdx), %xmm1
        cmpltpd   %xmm1, %xmm2
        andps     %xmm2, %xmm3
        andnps    %xmm0, %xmm2
        orps      %xmm2, %xmm3
        movaps    32(%rdx), %xmm4
        addpd     %xmm1, %xmm4
        mulpd     %xmm3, %xmm4
        movaps    %xmm4, (%rdx)
        addq      $16, %rdx
        cmpq      %rax, %rdx
        jl        ..B1.7         # Prob 99%
```

12	11	10	9	8	7	6	5	4	3	2	1	0

Explicit programming

- Use special class libraries + intrinsics
- Classes:
  - F64vec2
  - F32vec4
- Intrinsics:
  - _mm_add_pd
  - _mm_mul_pd
  - _mm_cmpgt_pd

See the next talk for examples

Conclusions

- SSE has grown into an extremely rich ISA extension
  - Too rich?
  - It is (somewhat) limited by the x86 legacy
- SSE may speed-up appropriate FLP loops
  - 4x (in single-precision mode)
  - 2x (in double-precision mode)
- Main decision to be taken:
  - Rely on auto-vectorization ?
  - “Hand-code” using intrinsics ?

BACKUP

Special moves

- Move to/from general registers

	Mnemonic	SS	SD	PS	PD
Extract	EXTRACT	--	--	66 0f 3a 17	--
Insert	INSERT	--	--	66 0f 3a 21	--
Extract Sign Mask	MOVMSK	--	--	0f 50	66 0f 50

FLP move operations (2)

- Move with duplication

	Mnemonic	(SS)	(SD)		
Move SP Low and Duplicate	MOVSLDUP	f3 0f 12	--	--	--
Move DP (Low) and Duplicate	MOVDDUP	--	f2 0f 12	--	--
Move SP High and Duplicate	MOVSHDUP	f3 0f 16	--	--	--