An initial encounter

Sverre Jarp

CERN openlab Minor Review Meeting – 16 June 2009

S.Jarp

Disclaimer

- This is a small digest of what is publicly available on the Web
- In particular:
  - "A First Look at the LRBni" by Michael Abrash, Dr. Dobb's Journal (Apr 01, 2009)
- But,
  - The presentation is principally limited to the instruction set

Today's agenda

- What is Larrabee, anyway ?
- Architecture
  - Chip, Processor, Vector
- Instruction overview
  - Vector load, store
  - Vector masks
  - Vector arithmetic, logical, shift
  - Others
- A simple code snippet

S.Jarp

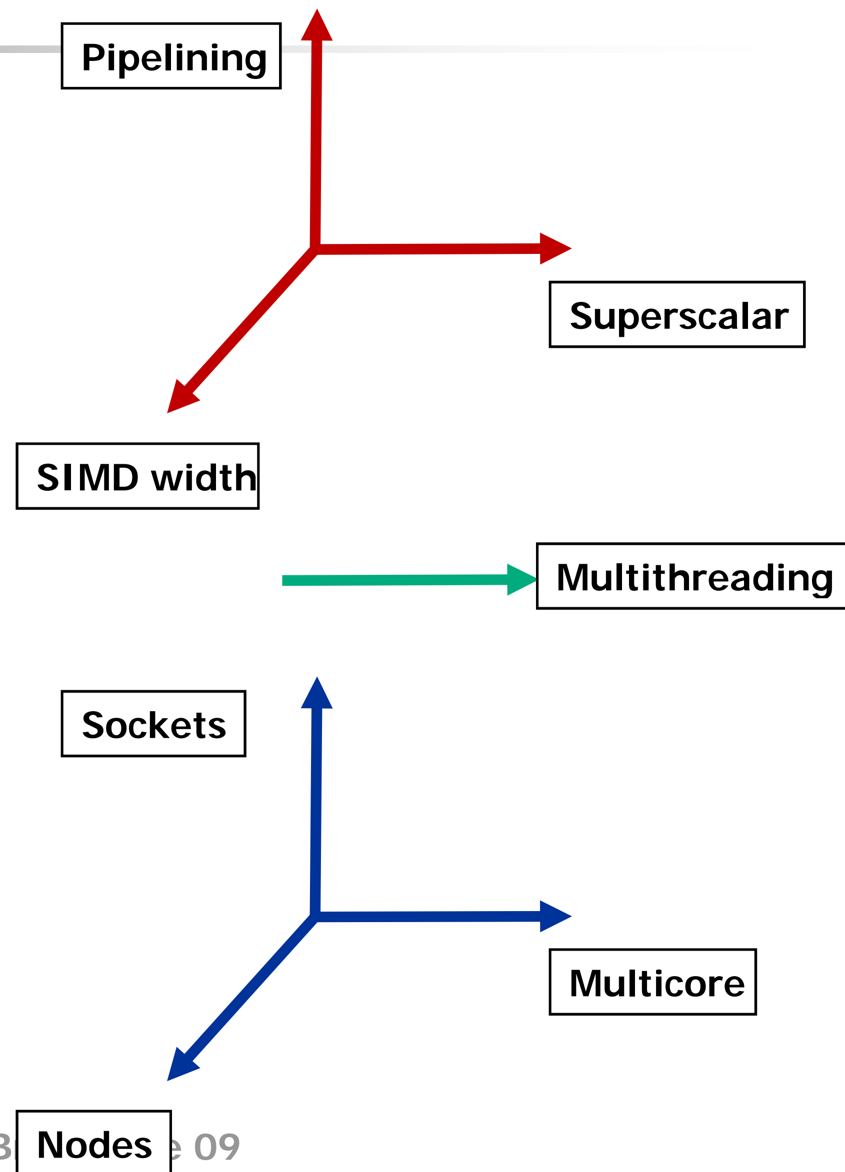
What is Larrabee?

- It is an architecture
  - Actually, it is a “RISC-like” architectural extension of x86
- Implementation:
  - Per core:
    - Multiple threads (currently 4)
    - Vector unit with lots of new instructions
    - In-order execution
  - Many cores
    - Interconnected via a coherent bus (or “ring”)
- “The architecture will first be used in GPUs, and could be used in CPUs as well”

S.Jarp

Sverre's 7 dimensions

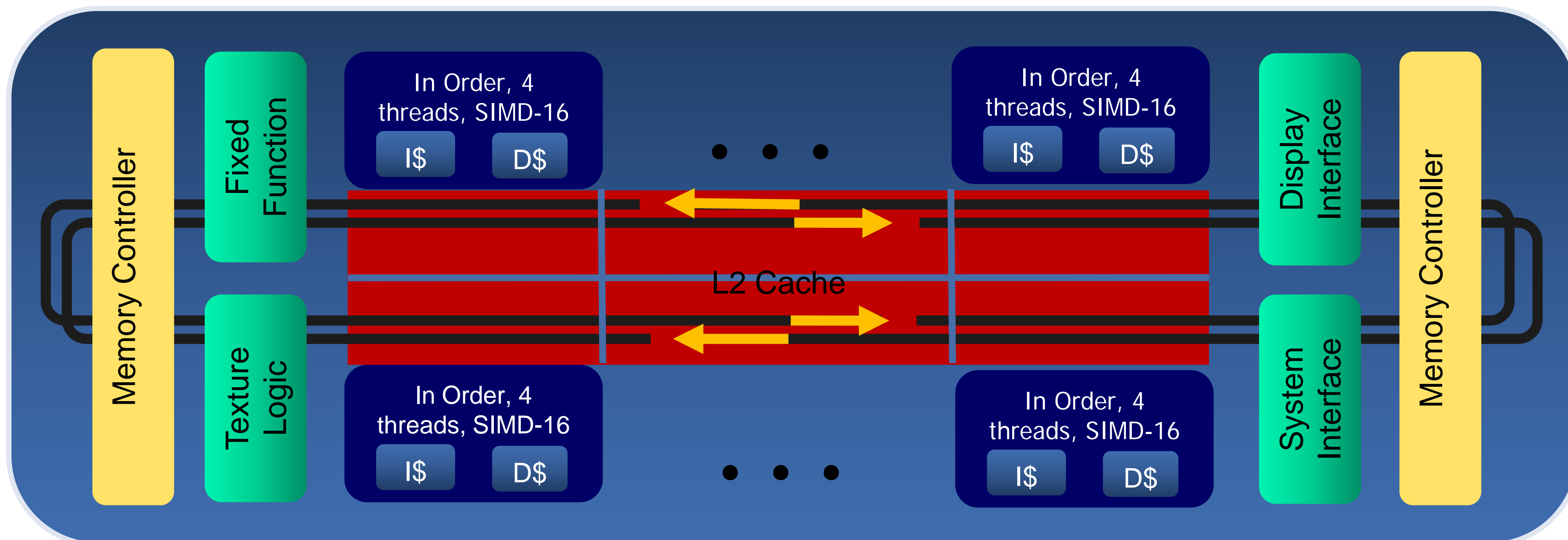
- Several are rather different in Larrabee
  - Compared to Xeon:
    - Superscalar (-)
    - SIMD width (++)
    - Multithreading (+)
    - Multicore (++)



Schematic chip overview

S.Jarp

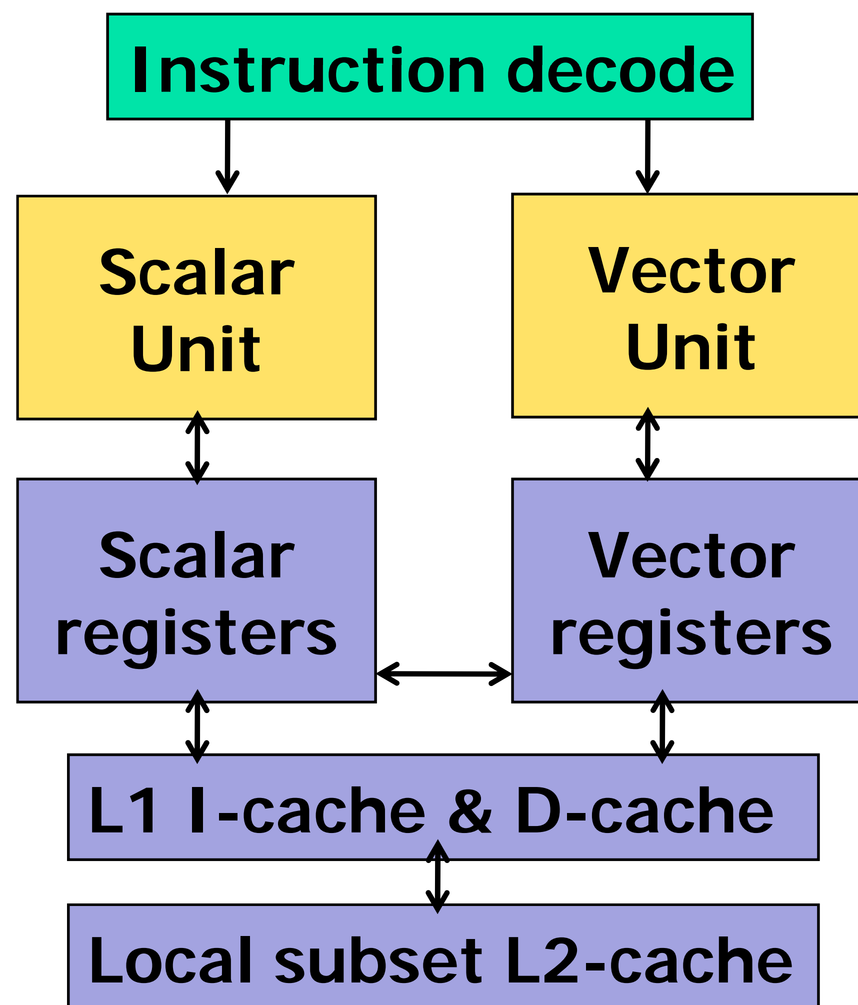
- Conceptually, it looks like this:



Processor diagram

S.Jarp

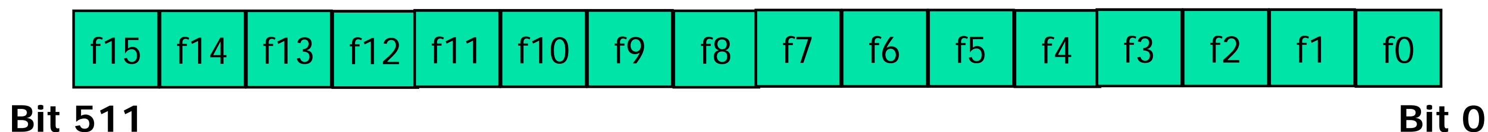
- Scalar and vector units are separate
  - Communication via registers (and flags)
- L1-cache
  - 32 KB each
- L2-cache
  - 256 KB each subset



S.Jarp

Vector-related registers

- 32 vector registers with 512 bits
  - v0 – v31
- Each can hold:
  - 16 floats, 16 int32s, 8 doubles, 8 int64s, etc.



- 8 mask registers with 16 bits
  - k0 – k7

Instruction classes

- Six major groupings:
  1) Vector arithmetic, logical and shift
  2) Vector compare
  3) Vector mask
  4) Vector load/store
  5) Misc. vector
  6) Misc. scalar

- Convention for mnemonics:
  - **v**xxx**pt** (**v**ector *instruction* **p**acked **t**ype)
  - **k**xxx (mas**k** *instruction*)

Standard instruction format

- Typically “vop v1{k1}, v2, (v3/src) “
  - Ternary (3 sources)
  - Target same as first source
  - Third source (but only this one):
    - Also addresses memory, as in: [rbx + rcx*4]
  - Mask register:
    - Predicates updates of target
    - Maintains “state”

Data element types

- Quick overview:

Size	Ld/St	Signed	Unsigned	FP
32	**d**word	**i**nt32	**u**int32	**s**ingle
64	**q**word			**d**ouble

{disu}

- So, an instruction may operate on all or only on some types:
  - v**min**p{disu}
  - v**sll**pi

S.Jarp

Arithmetic, Logical and Shift

- Summarized in the backup section
- Normal collection of
  - Logical (and, or, xor, not, etc.)
  - Shift (Shift left logical, shift right arithmetic, etc.)
  - Lots of convert variants
  - Add, subtract, multiply
  - Min, max
  - Scale, round, etc.
- Some more exotic ones

Vsllpi (shift i32 vector left logical)

- vsllpi v1, v2, v3
  - Shift each i32 element in v2 left logical according to values in v3.
  - Store in v1

v2:	5	14	13	12	11	10	5	6	2	5	7	9	14	4	4	4
<<																
v3:	2	1	0	3	0	0	3	2	1	0	3	1	0	2	1	0
=																
v1:	20	28	13	96	11	10	40	24	4	5	56	18	14	16	8	4

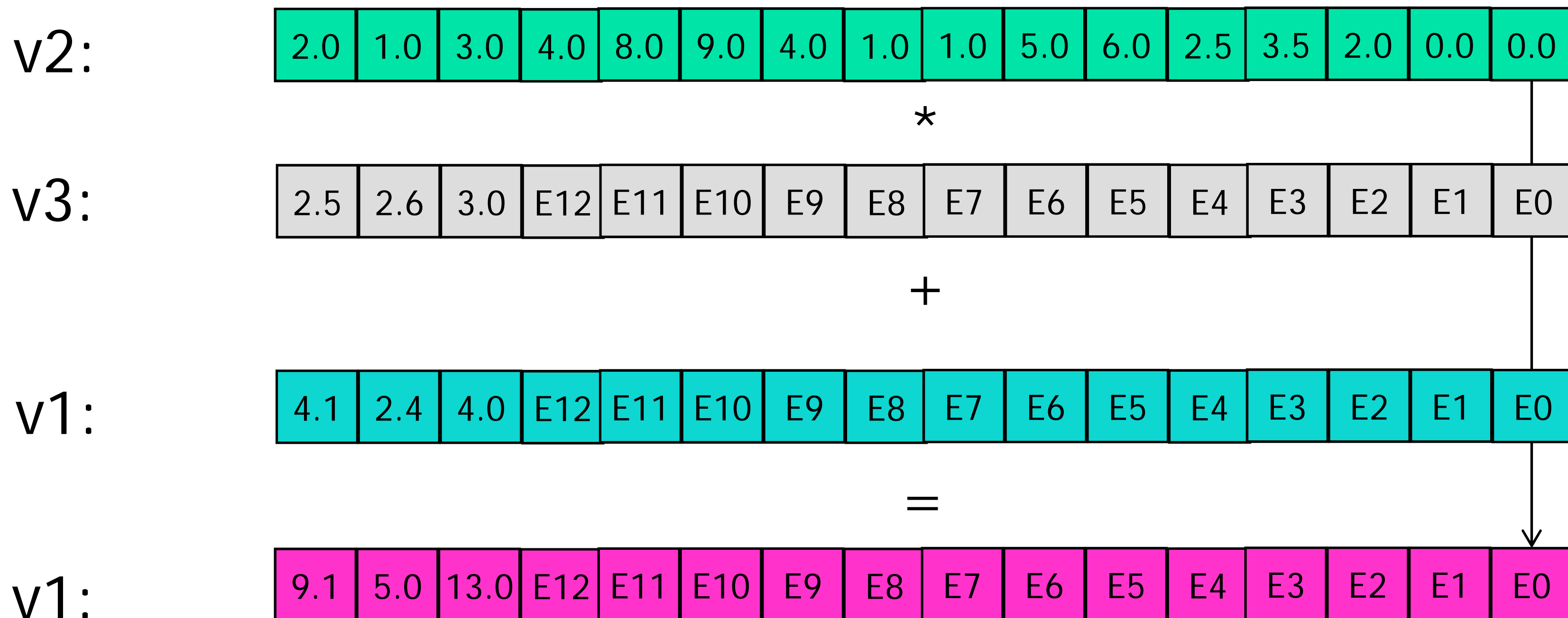
Standard Math Instructions

- 24 fused multiply-add/subtract instructions of the format:
  - "vm fff nnn p t" ← White space for readability
    - fff (function): add, addn (negate), sub, subr (reverse)
    - nnn (sequence of operands): 132, 213, 231
    - t: {ds}

- For instance:
  - vmadd231ps

Multiply Add (231ps)

- vmadd231ps v1, v2, v3
  - Multiply f32 v2 and f32 v3, add the result to f32 v1.



S.Jarp

Fused multiply-Add

- Combines one multiplication and one addition
  - Without any loss of accuracy

- Good for cases like:
  - $X^3*A+X^2*B+X*C+D =$
  - $X*(X*(X*A+B)+C)+D$

Vector compare instructions

- Syntax:
  - "vcmpp{disu} k2{k1}, v1, v2, cc"
  - cc:
    - eq, neq
    - lt, nlt
    - le, nle
    - ord, unord

- Results are always stored in a mask register

Mask instructions

- kand
  - Bitwise logical-and
- kandn
  - Bitwise logical-and-not
- kandnr
  - And-not reverse
- knot
  - Bitwise logical-not
- kor
  - Bitwise logical-or
- kxnor
  - Bitwise logical-xnor
- kxor
  - Bitwise logical-xor

- kortest
  - Set ZF if OR results in all ‘0’, CF if all ‘1’
- kmov
  - Move vector masks
- kmovlhb
  - Move low to high byte
- kswapb
  - Swap and merge high byte portion

S.Jarp

Vector Load/Store instructions

- vgatherd
  - Gather vector (32-bit elements)
- vgatherpfd
  - Prefetch vector (in gather form)
- vload{dq}
  - Load vector
- vexpand{dq}
  - Load unaligned and expand to vector
- vcompress{dq}
  - Compress and store unaligned from vector
- vscatterd
  - Scatter vector
- vscatterpfd
  - Prefetch vector (in scatter from)
- vstore{dq}
  - Store vector

← "d" for "dword", not "double"

Two common data scenarios

- Work with SOAs
  - Structures of Arrays or simply Arrays
- Work with AOSs
  - Arrays of Structures

S.Jarp

Structures Of Arrays

- Typical sequence:
  - Load all data
    - vloadd or vexpandd
  - Work
  - Perform tests
  - Mask out irrelevant elements
  - More work
  - Store modified elements
    - vstored or vcompressd
- Masking (predication) ensures algorithmic optimization, possibly also vexpand

Arrays of Structures

- Typical sequence:
  - Gather all data (vgatherd)
  - Work
  - Perform tests
  - Mask out irrelevant elements
  - More work
  - Store modified elements (vscatterd)

- Gather and masking (predication) should ensure even better algorithmic optimization

Simple example (Checksumming)

- From Dr. Dobbs:
  - v0 accumulates the results
  - v2 keeps addressing offsets

```
        vxorpi        v0, v0, v0
;
ChecksumLoop:
        vgatherd      v1{k0}, [rbx + v2]
        vaddpi        v0, v0, v1
        vaddpi        v2,v2, [Mem_Structure_Sizes]
        dec           rcx
        jnz           ChecksumLoop
```

S.Jarp

Simple example (Navigation)

- Physics example:
  - Is the particle inside a box or not?

```
if (abs(point[0] - origin[0]) > xhalfsz) return FALSE;
if (abs(point[1] - origin[1]) > yhalfsz) return FALSE;
if (abs(point[2] - origin[2]) > zhalfsz) return FALSE;
return true;
```

- Can now be handled inside one vector register
  - This will also be true for AVX
    - Next year's extension to Xeon

S.Jarp

Andrzej's usage comparisons

- Do you want (computing) power, flexibility, or ease of use?

ASM

Power
Flexibility
Ease of use

Intrinsics

Power
Flexibility
Ease of use

Autovectorization

Power
Flexibility
Ease of use

High level

Power
Flexibility
Ease of use

Conclusions

- Larrabee exploits Moore's law in several dimensions:
  - Long vectors
    - Coupled with sophisticated instruction set
  - Four threads
  - Large (double-digit) core count
- Consequently, applications need to expose:
  - Data parallelism
  - Task parallelism

Further reading

- On the Web:
  - "A First Look at the LRBni" by Michael Abrash, Dr. Dobb's Journal (Apr 01, 2009)
    - http://www.ddj.com/architect/216402188
  - "Larrabee: A Many-Core x86 Architecture for Visual Computing" by L.Seiler et al (Siggraph, Aug. 2008)
    - http://software.intel.com/file/18198/
  - Collection of articles from Intel, including:
  - "Game Physics Performance on the Larrabee Architecture", by A.Bader et al.
  - "Rasterization on Larrabee" by Michael Abrash
    - http://software.intel.com/en-us/visual-computing/

BACKUP

CERN openlab

S.Jarp

Arithmetic, Logical and Shift (1)

- v**adc**pi
  - Add vectors with carry (in and out)
- v**addn**p{ds}
  - Add and negate vectors
- v**add**p{dis}
  - Add vectors
- v**addsetc**pi
  - Add vectors and set mask to carry
- v**addsets**p{is}
  - Add vectors and set mask to sign
- v**andn**p{iq}
  - Bitwise logical-and-not vectors
- v**and**p{iq}
  - Bitwise logical-and vectors
- v**clampz**p{is}
  - Clamp vector between value and zero
- v**cvtpd2**p{isu}
  - Convert vector of double
- v**cvtpi2**p{ds}
  - Convert vector of int32
- v**cvtps2**p{diu}
  - Convert vector of double
- v**cvtps2srgb8**
  - Convert single to sRGB8
- v**cvtpu2**p{ds}
  - Convert vector of uint32

Arithmetic, Logical and Shift (2)

- v**maxabs**ps
  - Absolute maximum of singles
- v**max**p{disu}
  - Maximum
- v**min**p{disu}
  - Minimum
- v**mulh**p{iu}
  - Multiply and store high
- v**mull**pi
  - Multiply and store low
- v**mul**p{ds}
  - Multiply

- v**or**p{iq}
  - Bitwise logical-or
- v**round**ps
  - Round vector
- v**sbb**pi
  - Subtract with borrow (in and out)
- v**sbbr**pi
  - Reverse subtract with borrow (in and out)
- v**scale**ps
  - Scale vector

S.Jarp

Arithmetic, Logical and Shift (3)

- v**sll**pi
  - Shift left logical
- v**sra**pi
  - Shift right arithmetic
- v**srl**pi
  - Shift right logical
- v**sub**p{dis}
  - Subtract
- v**subr**p{dis}
  - Reverse subtract
- v**subrsetb**pi
  - Reverse subtract and set borrow
- v**subsetb**pi
  - Subtract and set borrow
- v**xor**p{iq}
  - Bitwise logical-xor

Special madd/msub cases

- vmadd231p{dis}
  - Variant with int32
- vmadd233p{is}
  - v1 = (v2 * ExtractScaleElement(v3)) + ExtractOffsetElement(v3)
- vmsubr23c1p{ds}
  - v1 = 1.0 – (v2 * v3)