Introduction to IA-32 and IA-64: Architectures, Tools and Libraries

G. Hager (RRZE), H. Bast (Intel)

Outline

- IA-32 Architecture
  - Architectural basics
  - Optimization with SIMD operations

- Why Multi-Core?

- Intel IA64 (Itanium) Architecture

IA-32

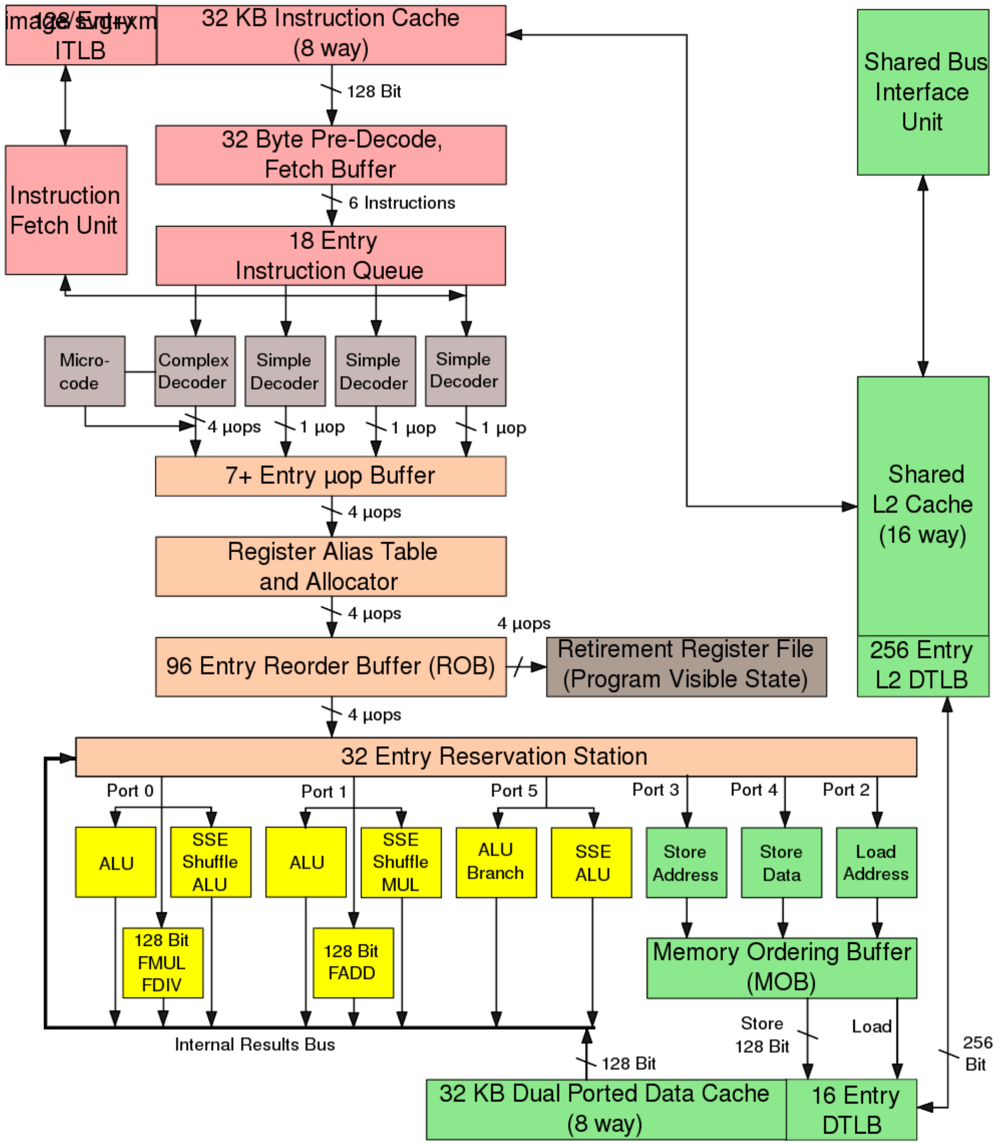
IA-32 Architecture Basics – a Little History

- IA-32 has roots dating back to the early 80s
  - Intel's first 16-bit CPU: 8086 with 8087 math coprocessor (x86 is born)
  - Even the latest Pentium 4 is still binary compatible with 8086
  - loads of advances over the last 20 years:
    - addressing range (1MB → 16MB → 64 GB)
    - protected mode (80286)
    - 32 bit GPRs and usable protected mode (80386)
    - on-chip caches (80486)
    - SIMD extensions and superscalarity (Pentium)
    - CISC-to-RISC translation and out-of-order superscalar processing (Pentium Pro)
    - floating-point SIMD with SSE and SSE2 (Pentium III/4)
- Competitive High Performance Computing was only possible starting with Pentium III
  - IA32 processors (Intel Core 2, AMD K10) are today rivaling all other established processor architectures

IA-32 Architecture Basics

- IA-32 has a CISC instruction set
  - „Complex Instruction Set Computing“
  - Operations like „load value from memory, add to register and store result in register“ are possible in one single machine instruction
  - Huge number of assembler instructions
  - Very compact code possible
  - Hard to interpret for CPU hardware, difficulties with out-of-order processing
- Since Pentium Pro, CISC is translated to RISC (called µOps here) on the fly
  - „Reduced Instruction Set Computing“
  - Very simple instructions like „load value from memory to register“ or „add two registers and store result in another“
  - RISC instructions are held in a reorder buffer for later out-of-order processing

IA-32 Architecture Basics: Intel Core 2 block diagram (one core)



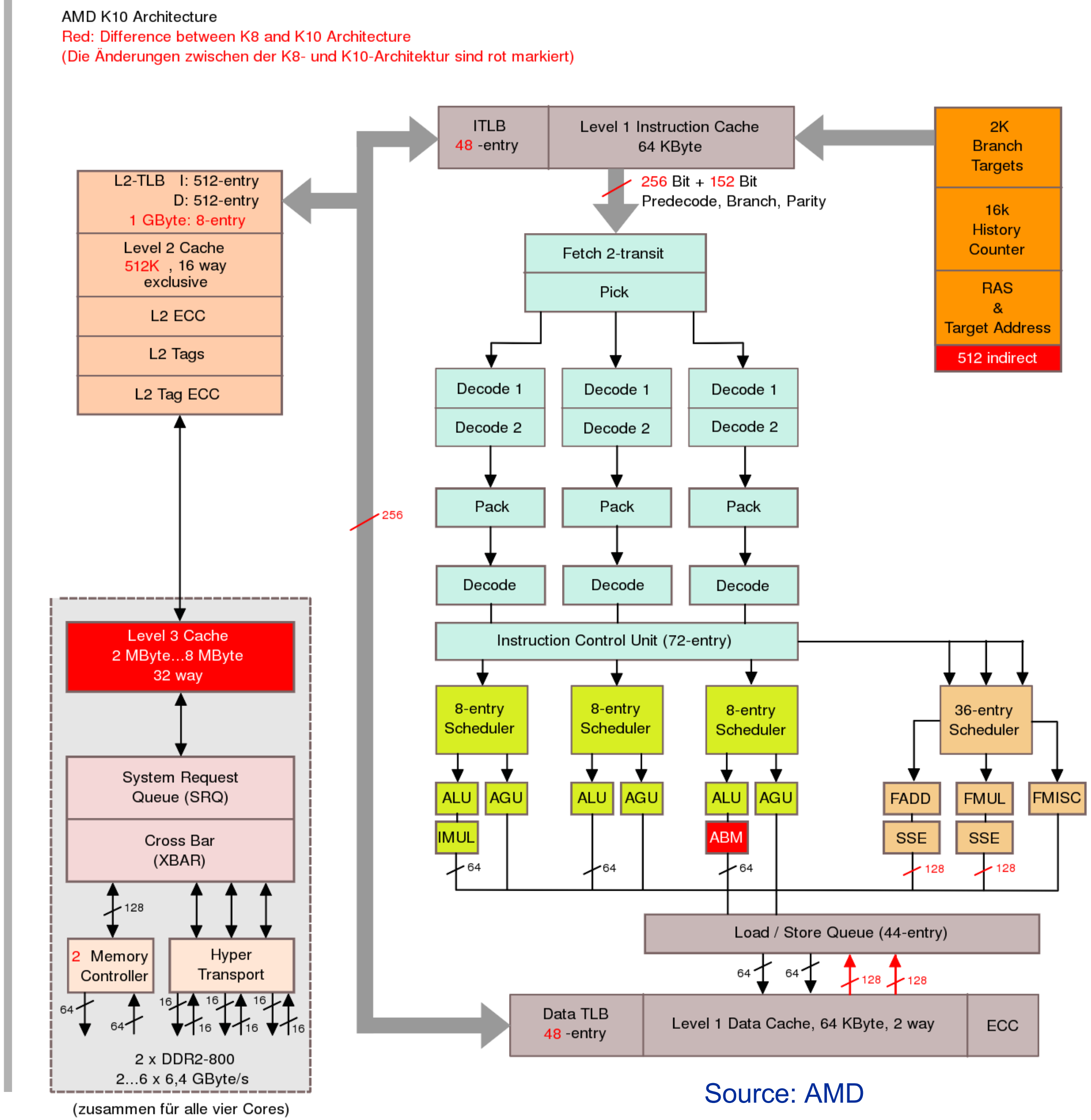
Intel Core 2 Architecture

Source: Intel Corp.

Intel Core 2 architecture

- Successor of Netburst
- Reduced pipeline length
- Improved instruction issue
- Double FP performance
- Instruction level parallelism: 4µops issue/cycle
- Desktop variants (45 nm)
  - Core2Duo „Wolfdale“
  - Core2Quad „Yorkfield“
  - Top bin: 3 GHz

IA-32 Architecture Basics:
AMD K10 block diagram (one core)



Source: AMD

AMD K10 architecture

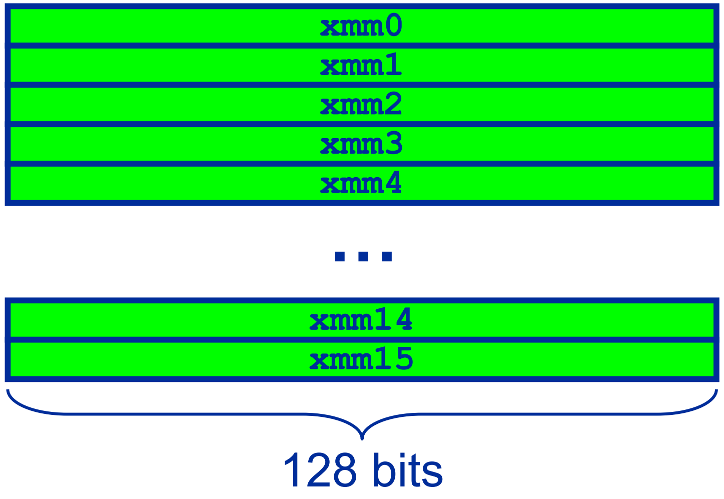
- To be used in first native Quad-Core („Barcelona“)
- L3 cache shared by all 4 cores
- There are only a few „Barcelona“ systems in field…(one at RRZE)
- Top bin: 2.0 GHz

IA-32 Architecture Basics

- This seems to be quite an ordinary architecture. It features
  - caches (instruction/data)
  - register files
  - functional units for integer & FP operations
  - a memory bus
- What‘s special about the Core 2 Duo / K10 ?
  - high clock frequency (currently 3.0 / 2.0 GHz)
  - special SIMD (Single Instruction Multiple Data) functional units and registers enable “vector computing for the masses”
  - Architectural enhancements
    - Low cache latencies
    - Opcode fusion
    - AMD: on-chip memory controllers
    - Large caches

IA-32 Architecture Basics: Floating Point Operations and SIMD

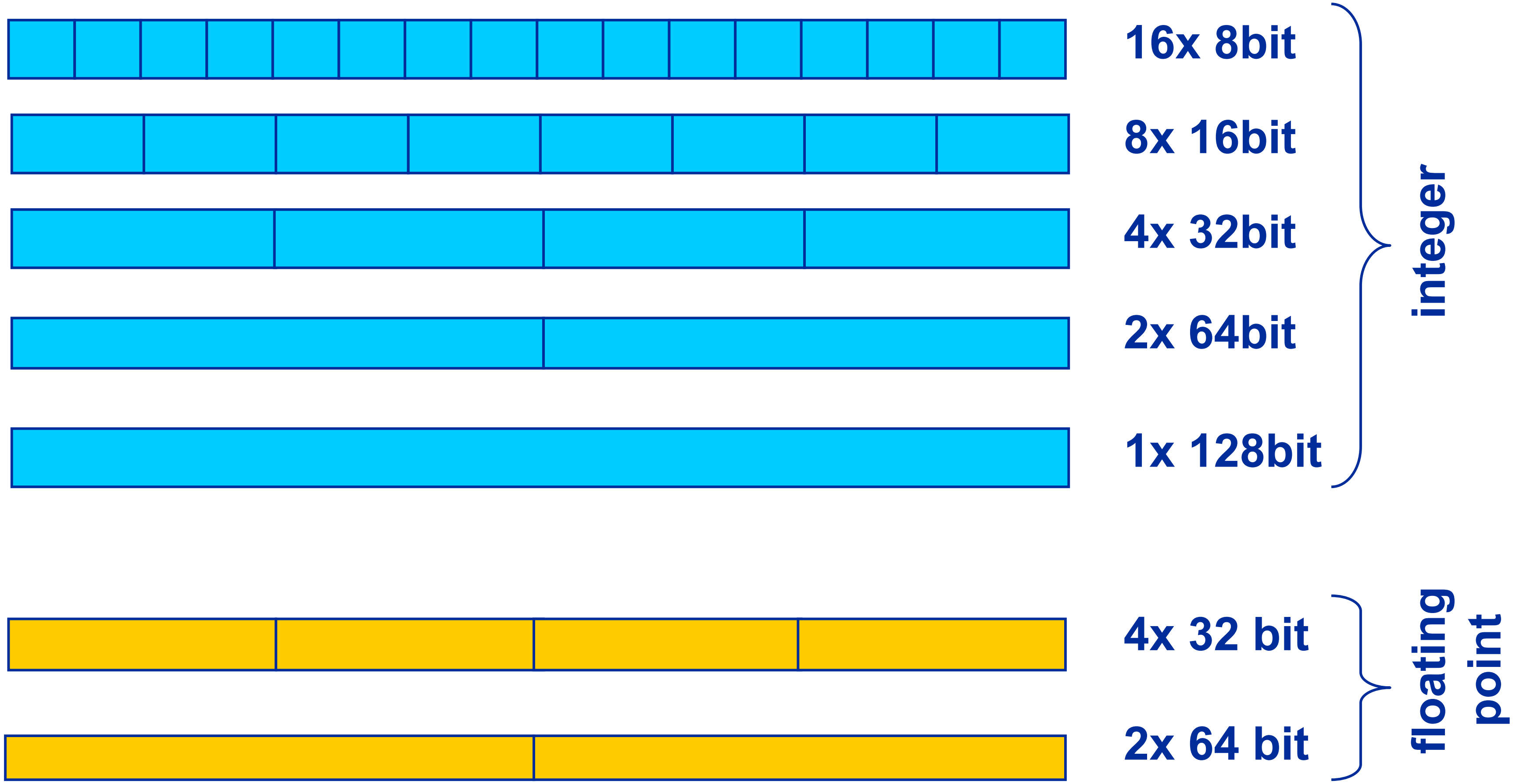
- First SIMD implementation: Pentium MMX
  - SIMD registers shared with FP stack
  - Switch between SIMD and FP mode was expensive overhead
  - Nobody should use this any more
- „Sensible SIMD“ came about with SSE (Pentium III) and SSE2 (Pentium 4) – Streaming SIMD Extensions
  - Register Model:



- Each register can be partitioned into several integer or FP data types
  - 8 to 128-bit integers
  - single (SSE) or double precision (SSE2) floating point
- SIMD instructions can operate on the lowest or all partitions of a register at once

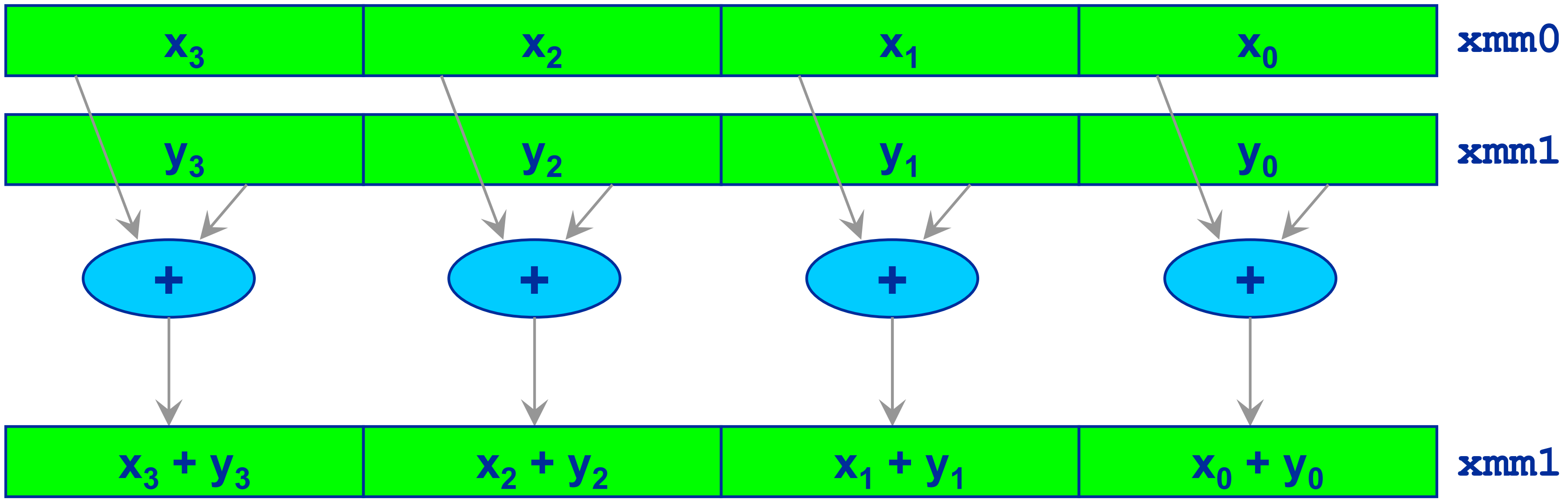
IA-32 Architecture Basics: SIMD Operations

- Possible data types in an SSE register



IA-32 Architecture Basics: Floating Point Operations and SIMD

- Example: Single precision FP packed vector addition



x_3 x_2 x_1 x_0 xmm0

y_3 y_2 y_1 y_0 xmm1

+ \+ \+ \+

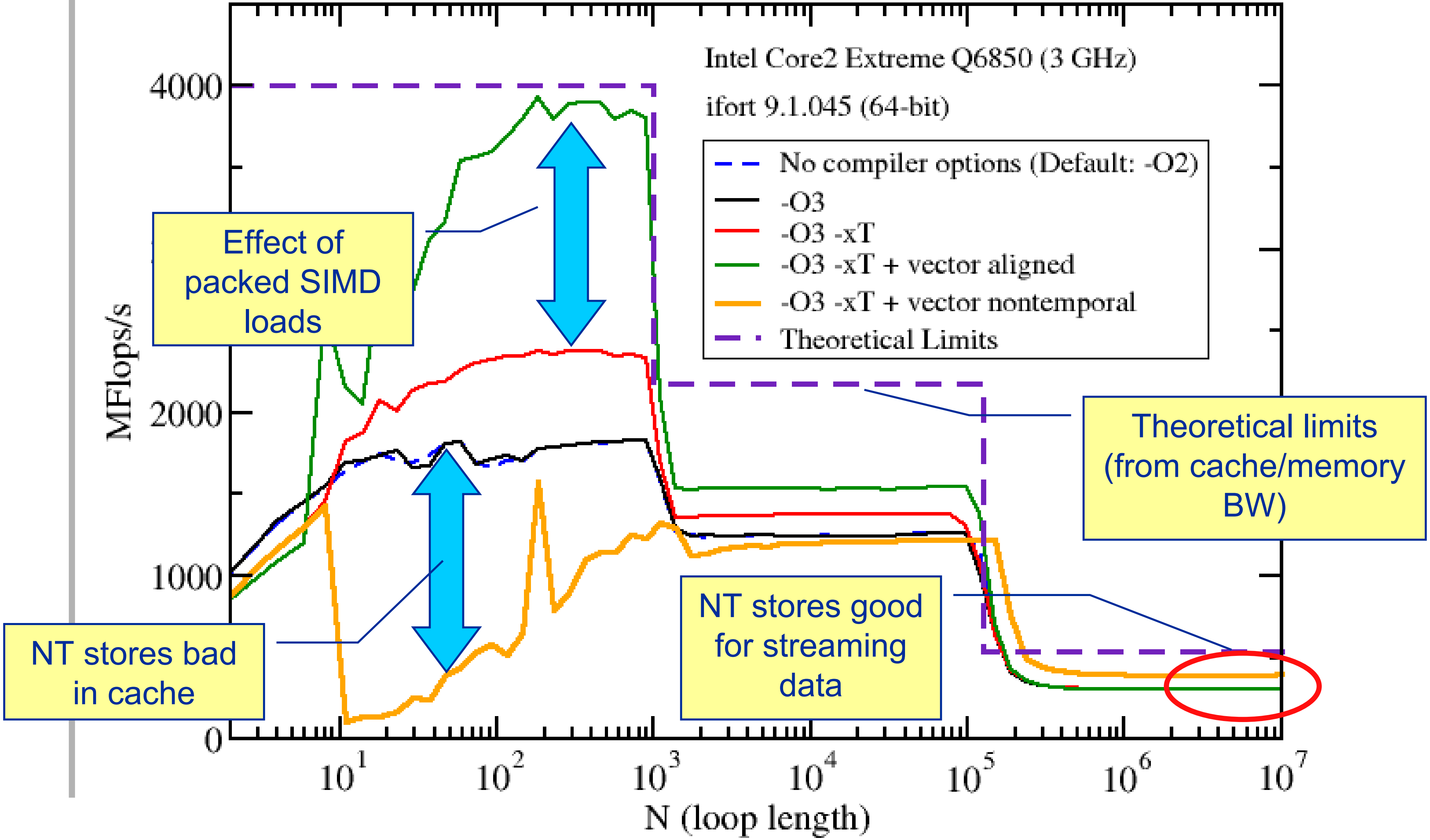
$x_3 + y_3$ $x_2 + y_2$ $x_1 + y_1$ $x_0 + y_0$ xmm1

- x_i and y_i are single precision FP numbers (4 per SSE register)
- Four single precision FP additions are done in one single instruction
- Core 2: 4-cycle latency & 1-cycle throughput for double precision SSE2 MULT & ADD leading to a peak performance of 4 (DP) FLOPs/cycle

IA-32 Architecture Basics: Floating Point Operations and SIMD

- **In addition to packed SIMD operations, scalar operations are also possible**
  - operate on lowest item only!
  - alternative for non-vectorizable codes (still better than x87)
- **Observe alignment constraints**
  - Loads/stores exist in aligned (fast) and unaligned (slower) variants
  - AMD K10: no penalty for unaligned loads on 8-byte boundaries
- **Significant performance boost achievable by using SSE(2)!**
  - hardly any gain for out of cache performance expected (but see below)
  - see later for compiler options concerning SSE
- **Another SSE feature: Non-temporal stores**
  - Bypass all cache levels; must be aligned
  - AMD K10: Scalar NT store available!
  - Good for “streaming” applications with no temporal locality

IA-32 Architecture Basics
Triads Performance: REPEAT[A(:) = B(:) + C(:) * D(:)]



Effect of packed SIMD loads

Theoretical limits (from cache/memory BW)

NT stores good for streaming data

NT stores bad in cache

IA-32 Architecture Basics: Programming With SIMD Extensions

- When given correct options, compiler will automatically try to vectorize simple loops
  - Rules for vectorizability similar as for SMP parallelization or "real" vector machines
  - See compiler documentation
- SIMD can also be used directly by the programmer if compiler fails
- Several alternatives:
  - **Assembly language**
    For experts only
  - **Compiler Intrinsics**
    Map closely to assembler instructions, programmer is relieved of working with registers directly.
  - **C++ class data types and operators**
    High-level interface to SIMD operations. Easy to use, but restricted to C++.

IA-32 Architecture Basics:
Programming With SIMD Extensions

- Special C++ data types map to SSE registers
- FP types:

`F32vec4`	4 single-precision FP numbers
`F32vec1`	1 single-precision FP number
`F64vec2`	2 double-precision FP numbers

- Integer types: `Is32vec4`, `Is64vec2`, etc.
- C++ `operator+` and `operator*` are overloaded to accomodate operations on those types
  - programmer must take care of remainder loops manually
- Alignment issues arise when using SSE data types
  - compiler intrinsics and command line options control alignment
  - uncontrolled unaligned access to SSE data will induce runtime exceptions!

IA-32 Architecture Basics: Programming With SIMD Extensions

- A simple example: vectorized array summation
- Original code (compiler-vectorizable):

```
double asum(double *x, int n) {
    int i;
    double s = 0.0;
    for(i=0; i<n; i++)
        s += x[i];
    return s;
}
```

- Hand-vectorized code:

```
#include <dvec.h>
double asum_simd(double *x, int n) {
    int i; double s = 0.0;
    F64vec2 *vbuf = (F64vec2*)x;
    F64vec2 accum(0.,0.);
    for(i=0; i<n/2; i++)
        accum += vbuf[i];
    for(i=n&(-2); i<n; i++)
        s += x[i];
    return accum[0] + accum[1] + s;
}
```

compiler-vectorized version is still faster than hand-vectorized; WHY?

→ exercise!

remainder loop

summation across SSE register

IA-32 Architecture Basics: Programming With SIMD Extensions

- Alignment issues
  - alignment of arrays in SSE calculations should be on 16-byte boundaries
  - other alternatives: use explicit unaligned load operations (not covered here)
  - How is manual alignment accomplished?
- 2 alternatives
  - manual alignment of structures and arrays with

    ```
    __declspec(align(16)) <declaration>;
    ```

  - dynamic allocation of aligned memory (align=alignment boundary)

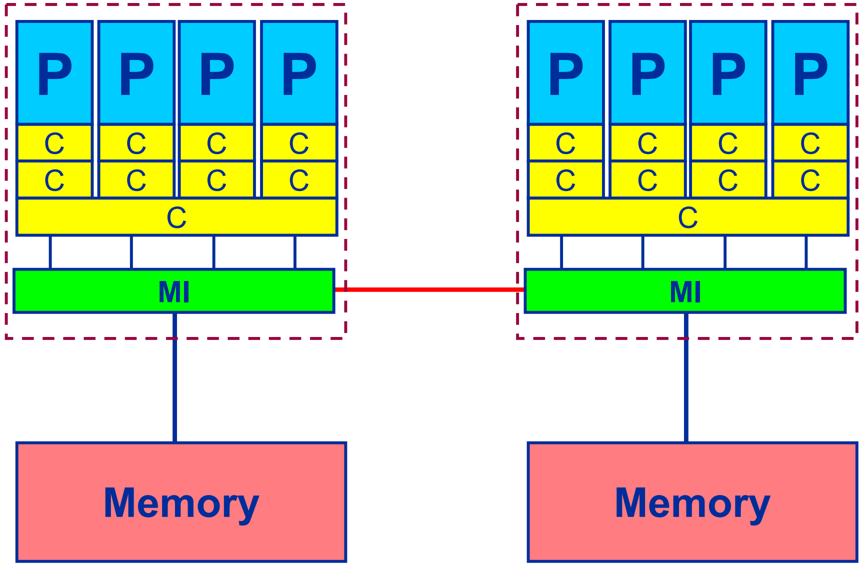
    ```
    void* _mm_malloc (int size, int align);
    void _mm_free (void *p);
    ```

    … or just use the standard `memalign()`

Shared-Memory System Architectures using Intel64/AMD64 Processors

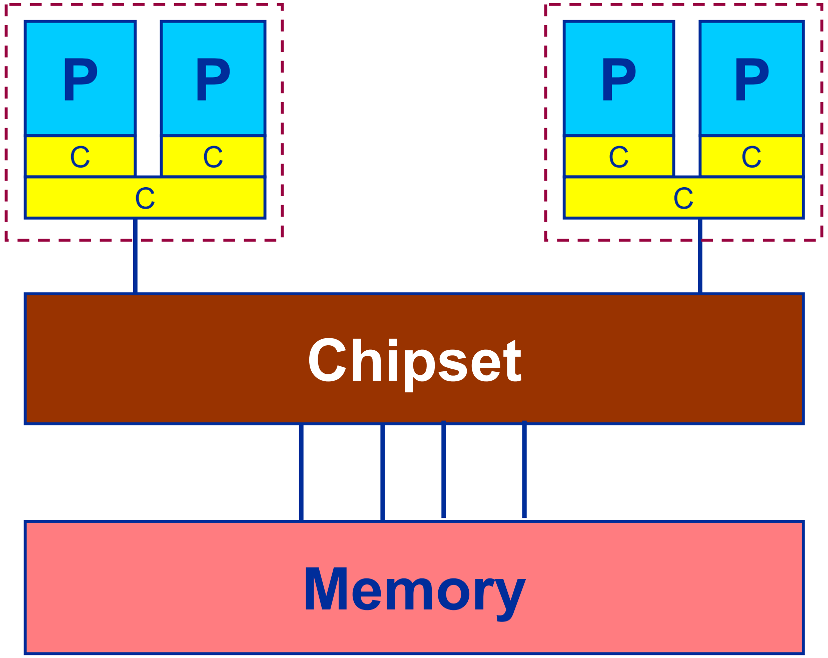
AMD Opteron (K10) Systems

- Typical cluster configuration: 2-socket node
  - 8 cores
  - 2 memory locality domains
  - Connected via HyperTransport
  - Shared L3 caches per socket
- ccNUMA
  - Local vs. non-local memory access
  - Potential congestion
  - Thread/process pinning is essential
- For details on ccNUMA programming, see the talk on Thursday



Intel “Woodcrest” (Ducl-Core Core2) Systems

- Typical cluster configuration: 2-socket system
  - 4 cores
  - FSB1333 (10.7 GB/s)
  - Theoretical bandwidth of 21.3 GB/s per system
  - Application-level bandwidth of < 45% of peak
  - Overall memory bandwidth does not scale perfectly from 1→2 sockets
- Flat memory model, but:
- Shared L2 per dual core
  - Makes thread placement an issue in spite of flat memory



- End of 2008: Intel will “go ccNUMA” as well!

Why Multi-Core?

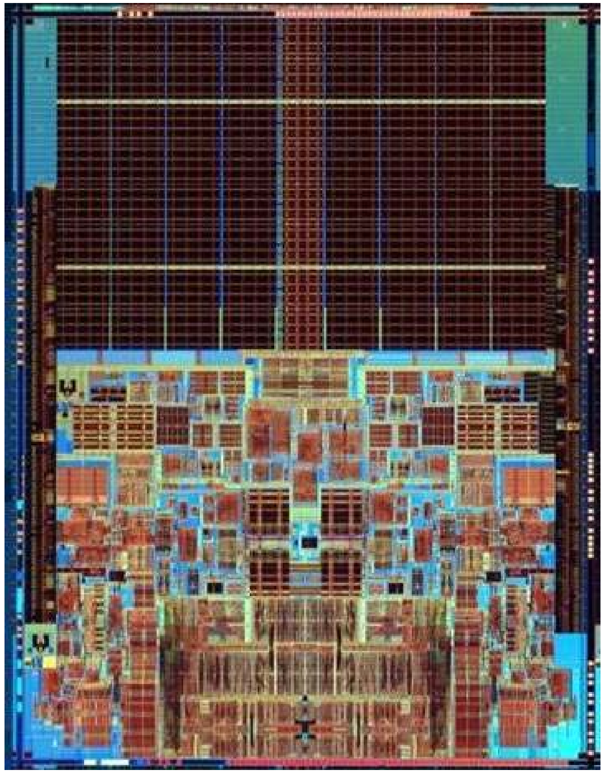
Multi-core processors

The party is over!

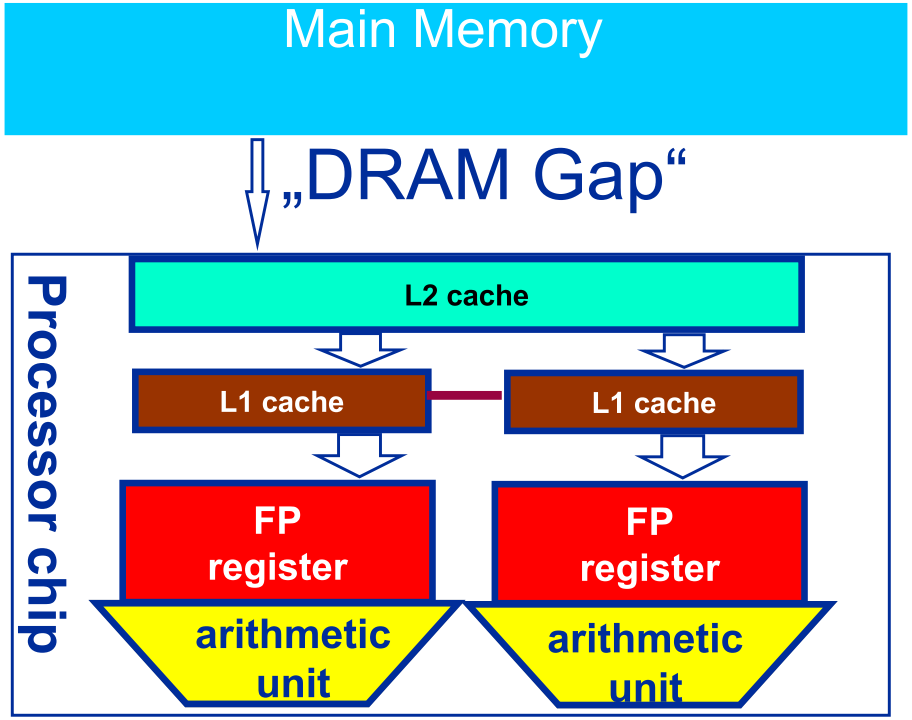
- Problem: Moore's law is still valid but increasing clock speed hits a technical wall (heat)
- Solution: Reduce clock speed of processor but put 2 (or more) processors (cores) on a single silicon die

Clock speed of single core will decrease in future!

(Xeon/Netburst: max. 3.73 GHz -> Xeon/Core: max. 3.0 GHz)

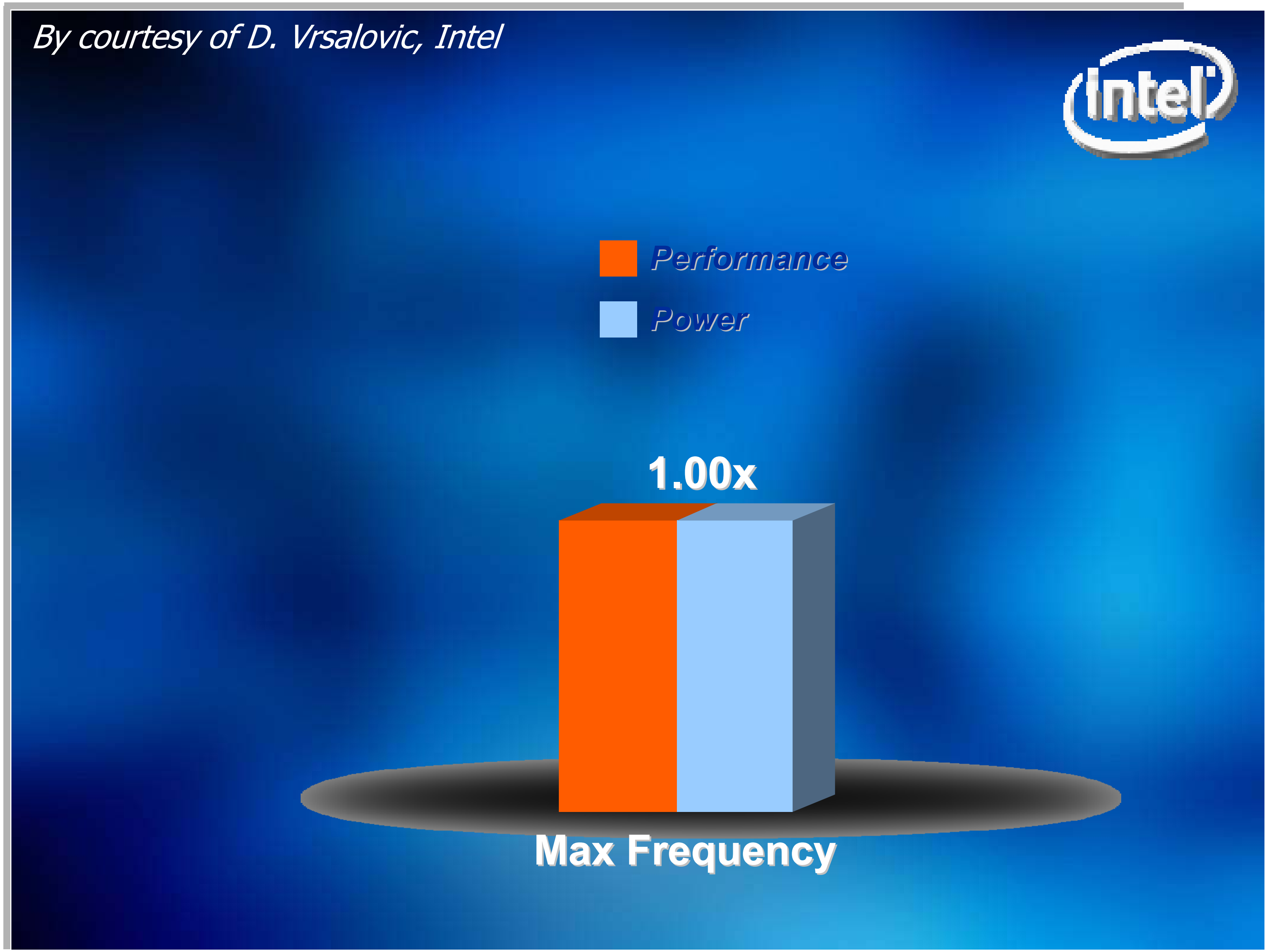


Intel Xeon / Core2 ("Woodcrest")



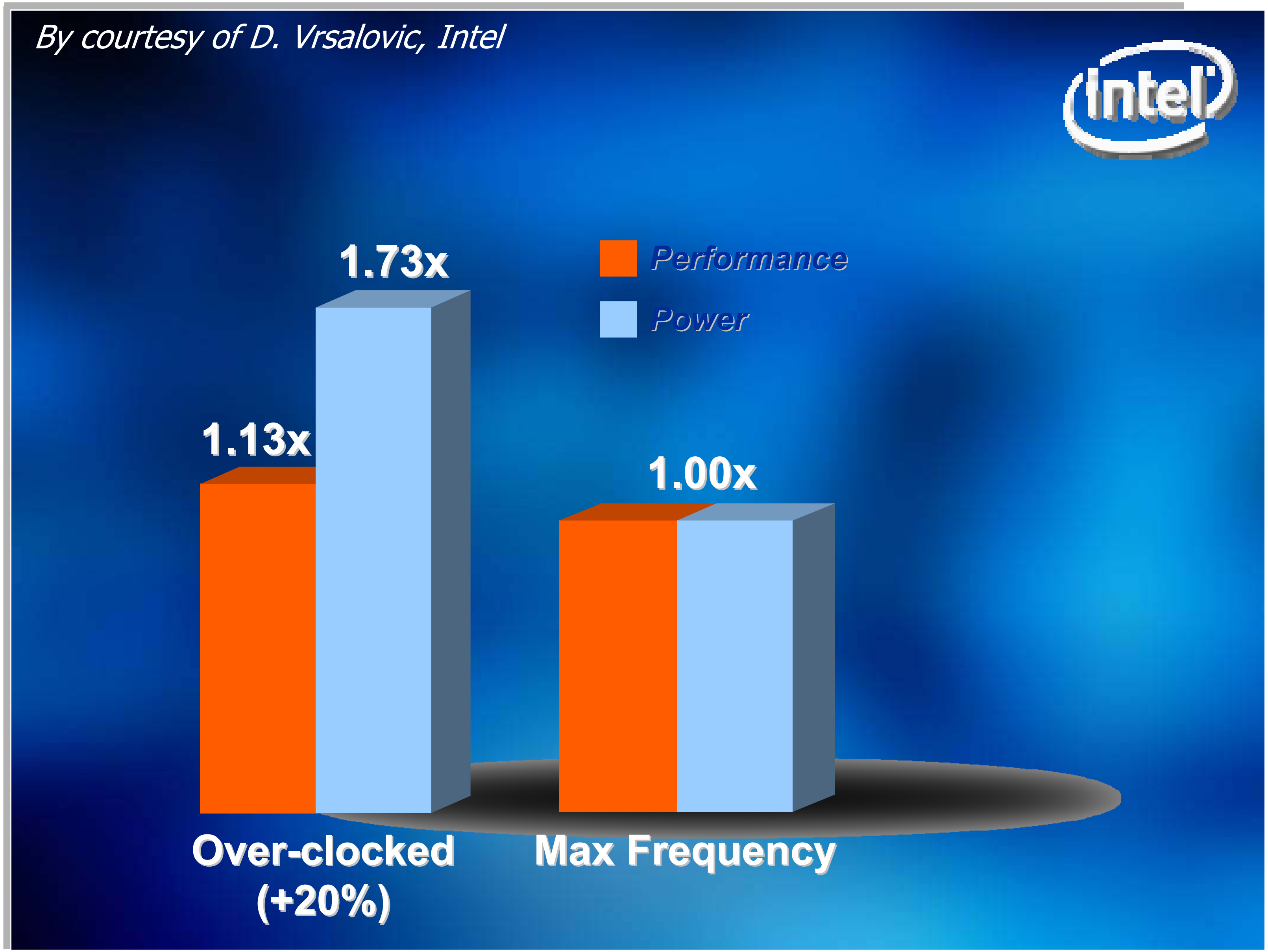
Multi-core processors

The party is over!



Multi-core processors

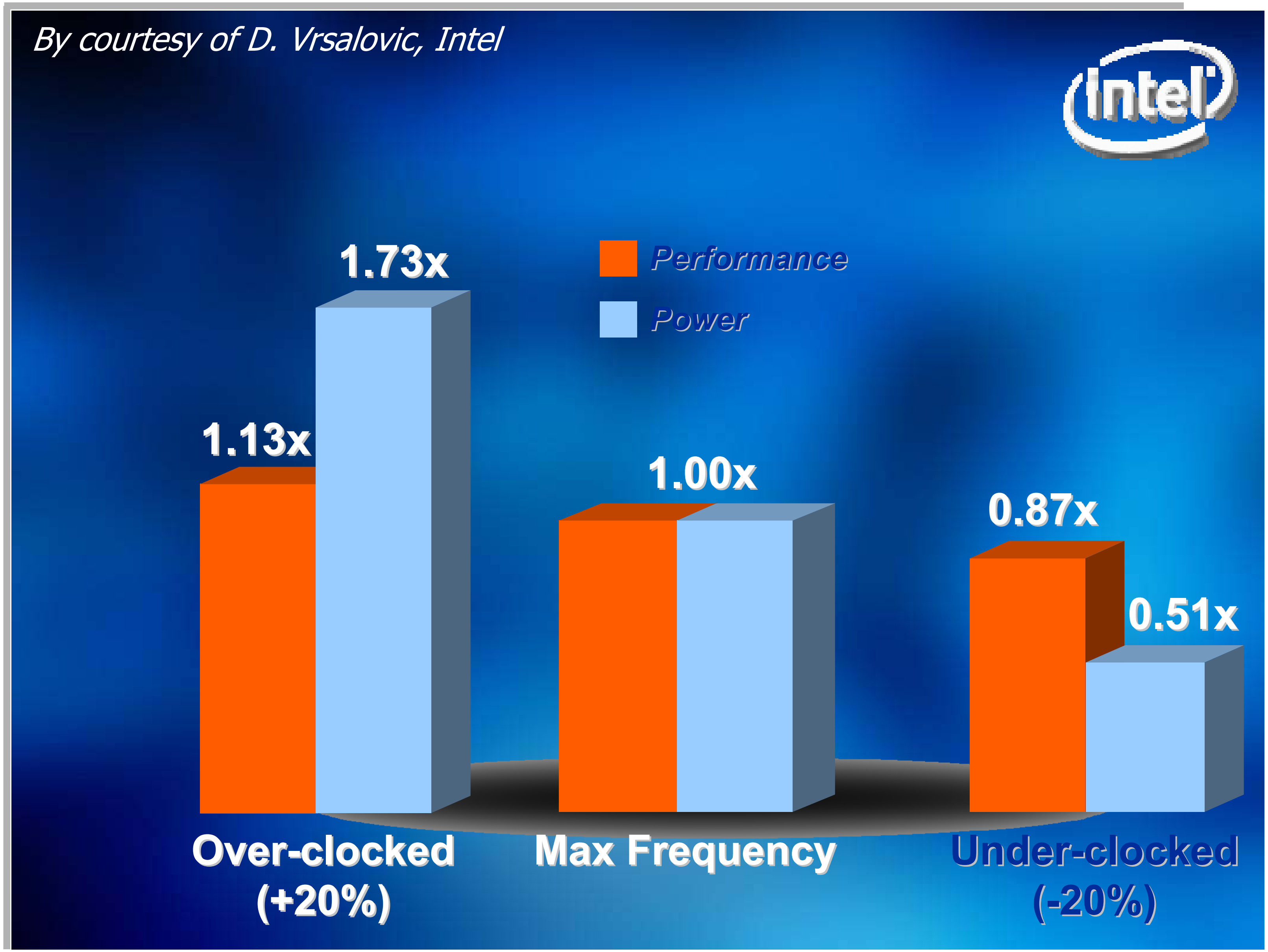
The party is over!



By courtesy of D. Vrsalovic, Intel

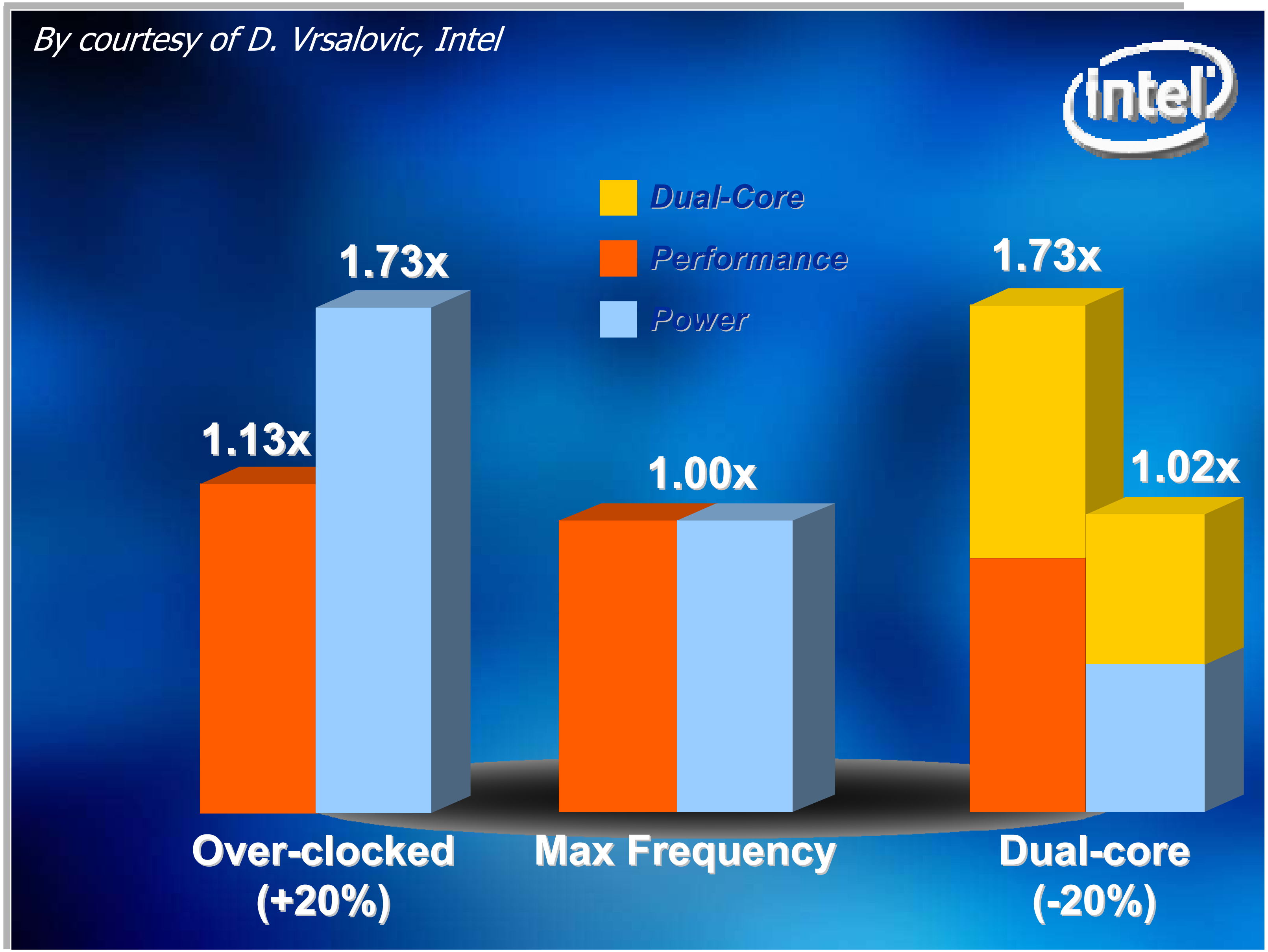
Multi-core processors

The party is over!



Multi-core processors

The party is over!



Power dissipation in VLSI Circuits

- In CMOS VLSIs, power dissipation is proportional to clock frequency:

$$W \propto f_c$$

- Moreover, it is proportional to supply voltage squared:

$$W \propto V_{cc}^2$$

- For reasons of noise immunity, supply voltage has to grow linearly with frequency, so:

$$\boxed{W \propto f_c^3}$$

- **Frequency reduction** is the key to saving power with modern microprocessors
  - all other factors, e.g. manufacturing technology, unchanged
- This seems to contradict the verdict of ever-growing chip performance

Multi-Core Processors

- Question: What fraction of performance must be sacrificed per core in order to benefit from *m* cores?
- Prerequisite: Overall power dissipation should be unchanged

$$W + \Delta W = (1+\varepsilon_f)^3 W$$

$$(1+\varepsilon_f)^3 m = 1$$

$$\boxed{\varepsilon_f = m^{-1/3} - 1}$$

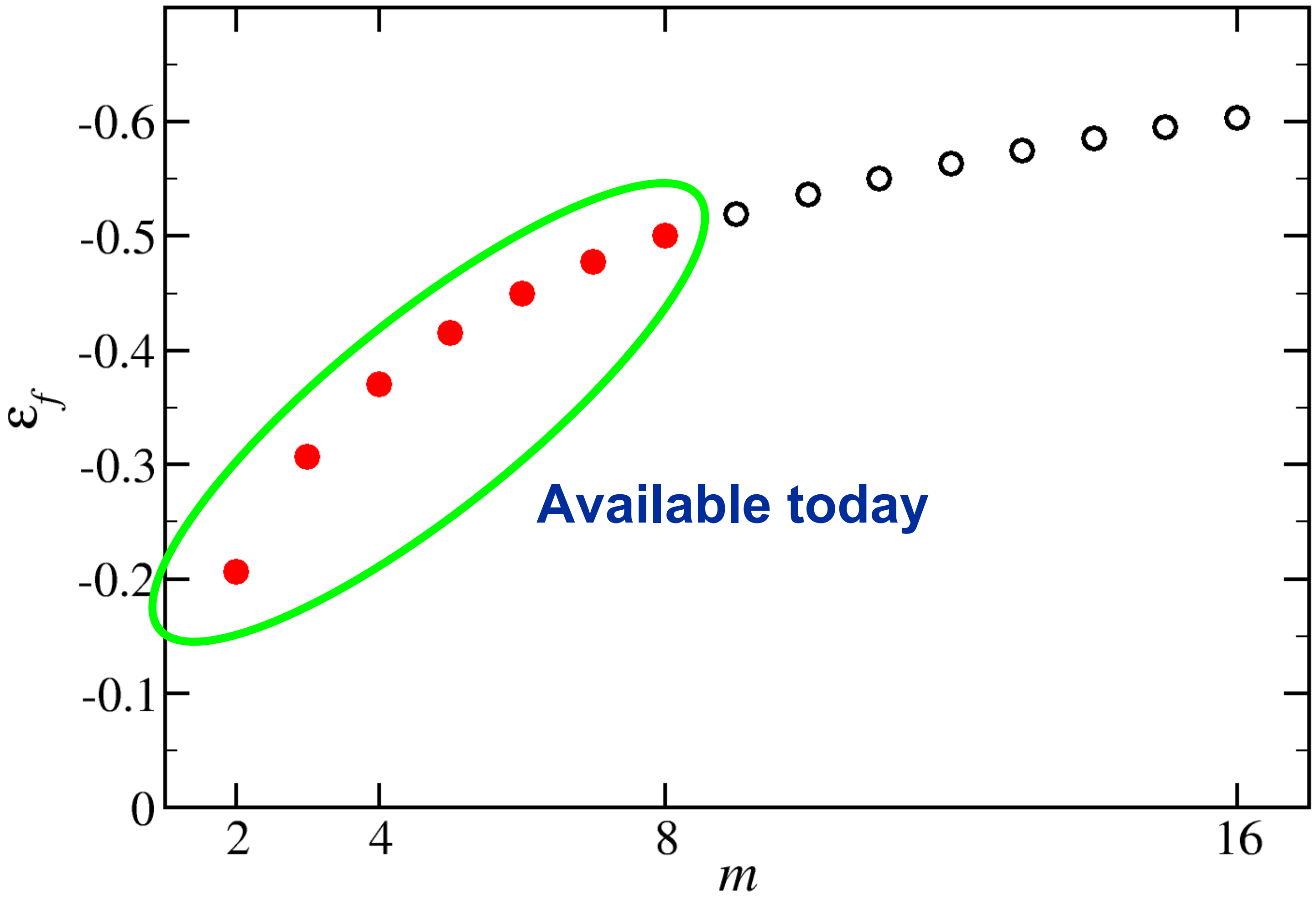
W	power dissipation
p	performance (1 core)
p_m	performance (*m* cores)
ε_f	rel. frequency change $\Delta f_c/f_c$
ε_p	rel. performance change $\Delta p/p$
m	number of cores

$$p_m = (1+\varepsilon_p)\,pm$$

$$\boxed{p_m \geq p \;\Rightarrow\; \varepsilon_p \geq \frac{1}{m} - 1}$$

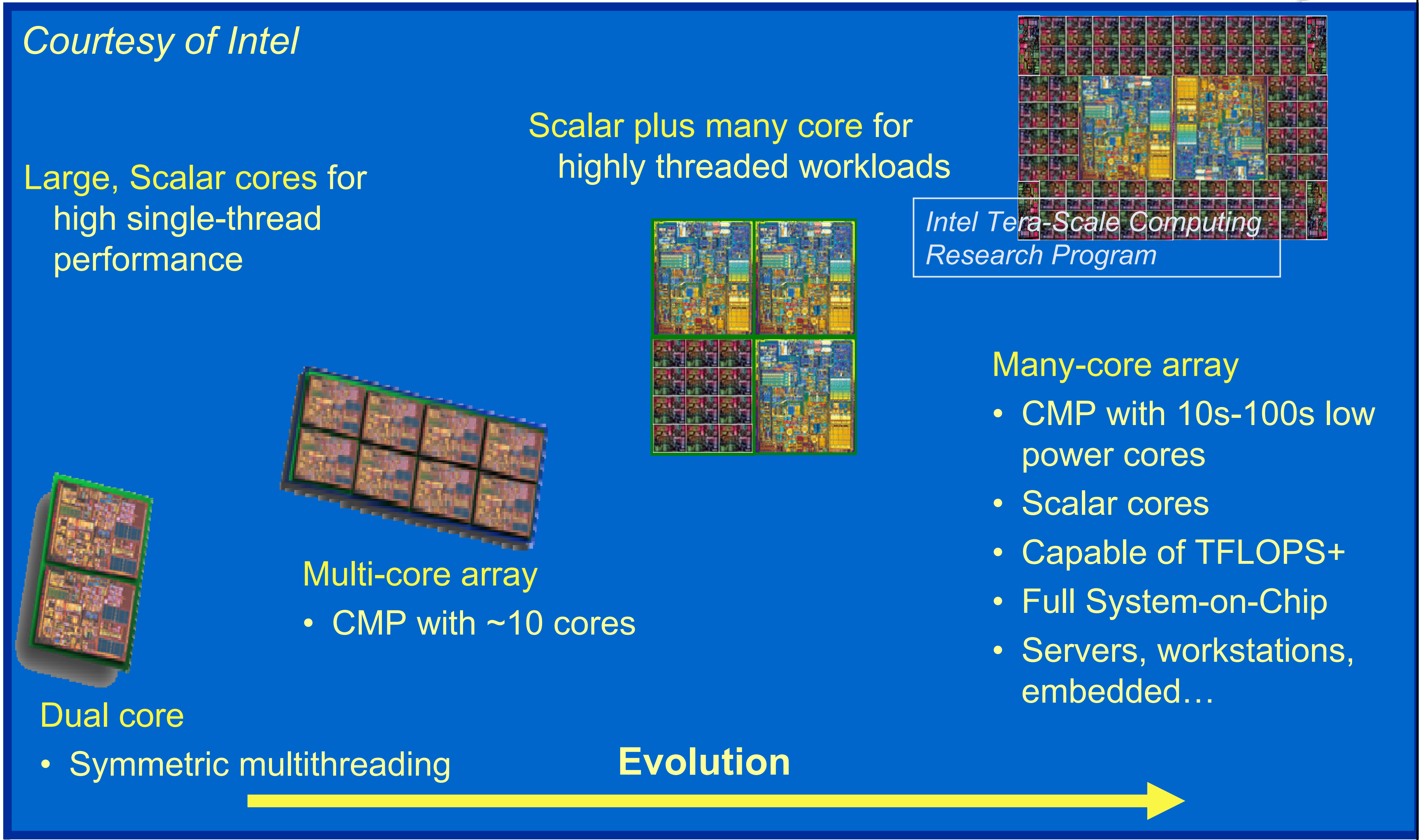
Why Multi-Core?

- Required relative frequency reduction vs. core count



- Remember the assumptions that go into this model!

Multi-core processors
A challenging future ahead?



Courtesy of Intel

Large, Scalar cores for high single-thread performance

Dual core
- Symmetric multithreading

Multi-core array
- CMP with ~10 cores

Scalar plus many core for highly threaded workloads

Intel Tera-Scale Computing Research Program

Many-core array
- CMP with 10s-100s low power cores
- Scalar cores
- Capable of TFLOPS+
- Full System-on-Chip
- Servers, workstations, embedded…

Evolution

Parallelization will be mandatory for everyone in the future !

IA-64: Intel's Itanium Architecture

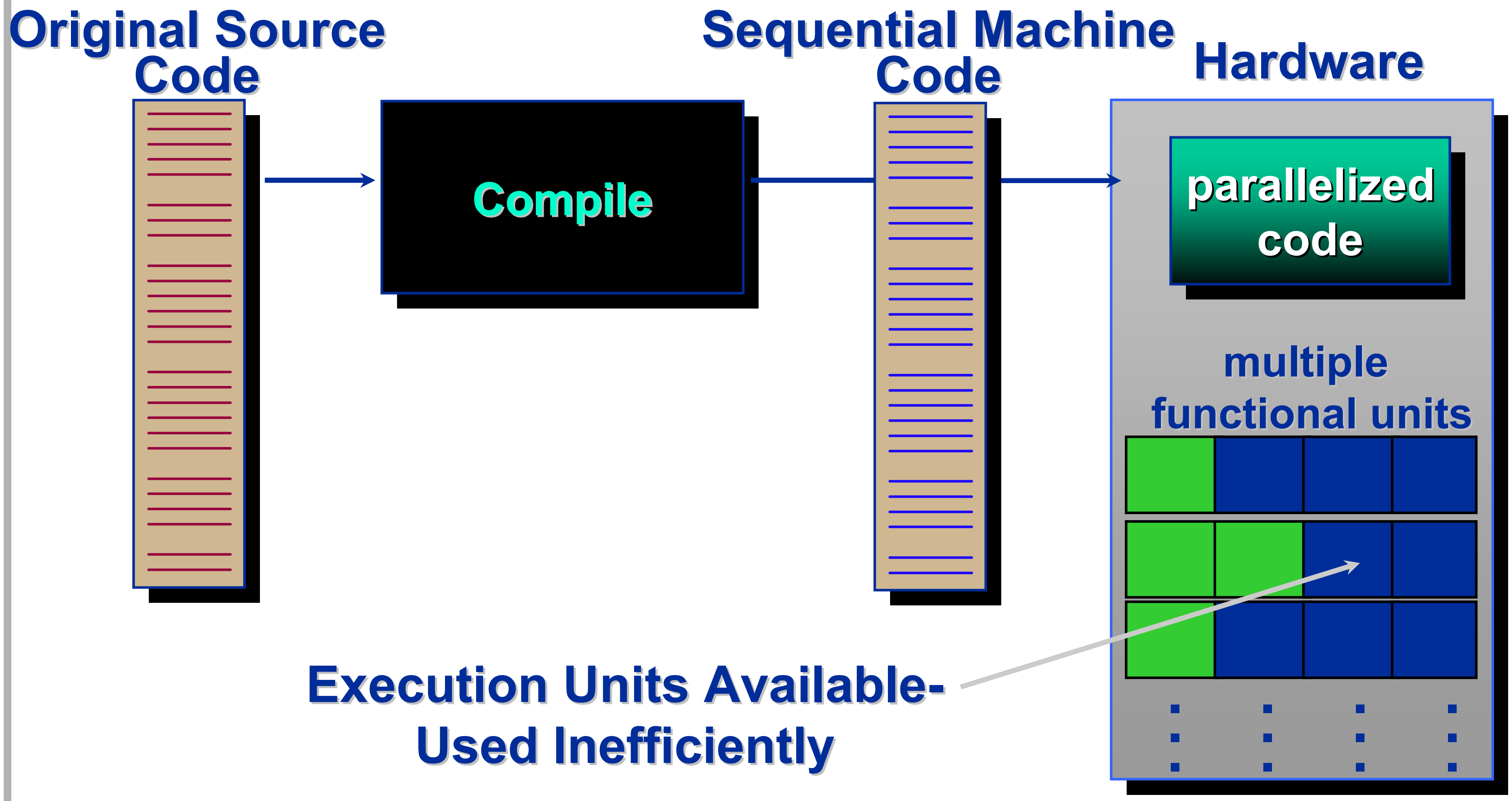
H. Bast (Intel), G. Hager (RRZE)

Itanium® Processor Architecture

Selected Features

- 64-bit Addressing Flat Memory Model
- Instruction Level Parallelism (6-way)
- Large Register Files
- Automatic Register Stack Engine
- Predication
- Software Pipelining Support
- Register Rotation
- Loop Control Hardware
- Sophisticated Branch Architecture
- Control & Data Speculation
- Powerful 64-bit Integer Architecture
- Advanced 82-bit Floating Point Architecture
- Multimedia Support (MMX™ Technology)

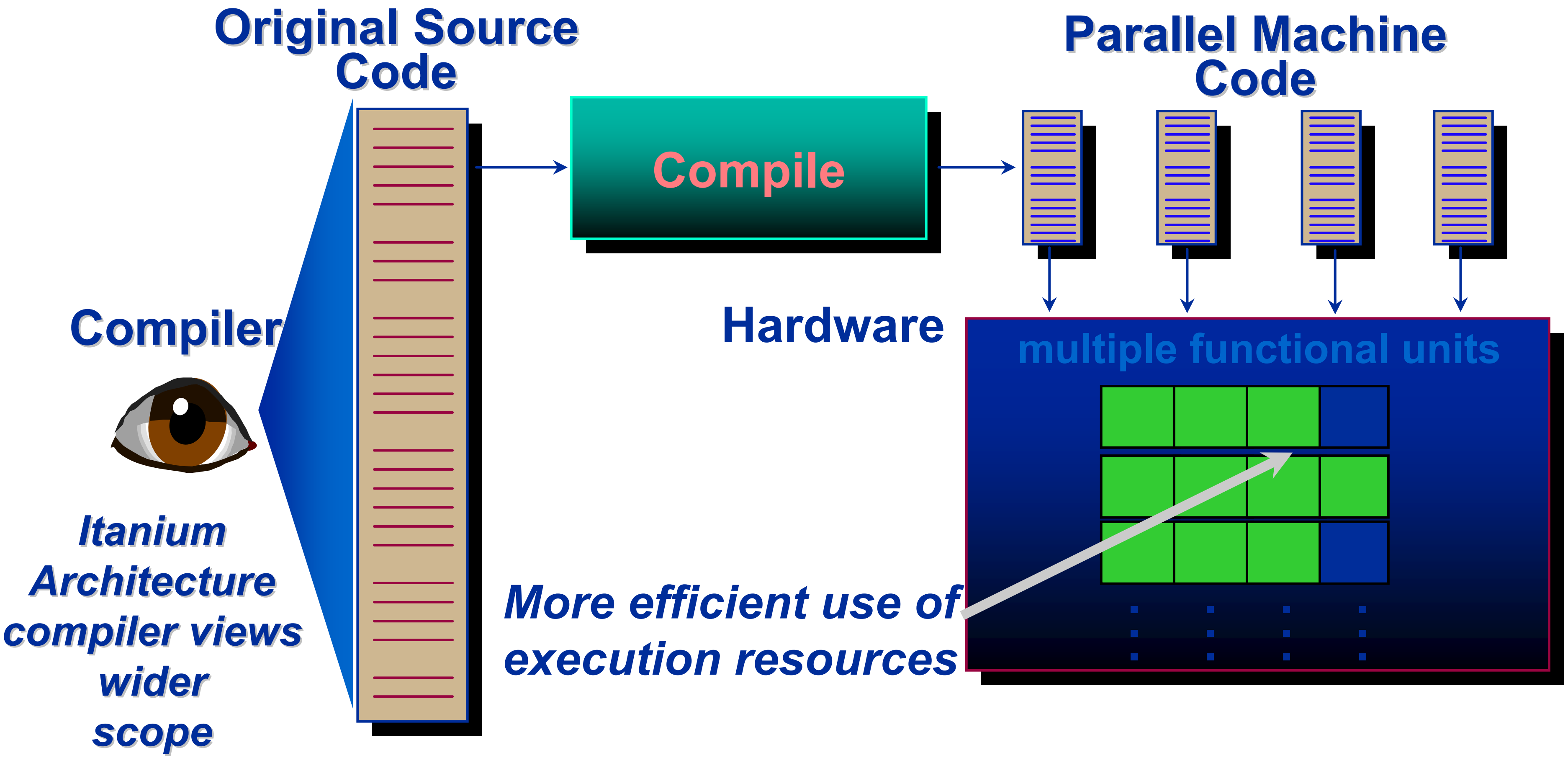
Traditional Architectures:
Limited Parallelism



Consequence: Current processors are often 60% idle

Intel® Itanium® Architecture:

Explicit Parallelism



Original Source Code

Compile

Parallel Machine Code

Compiler

Hardware

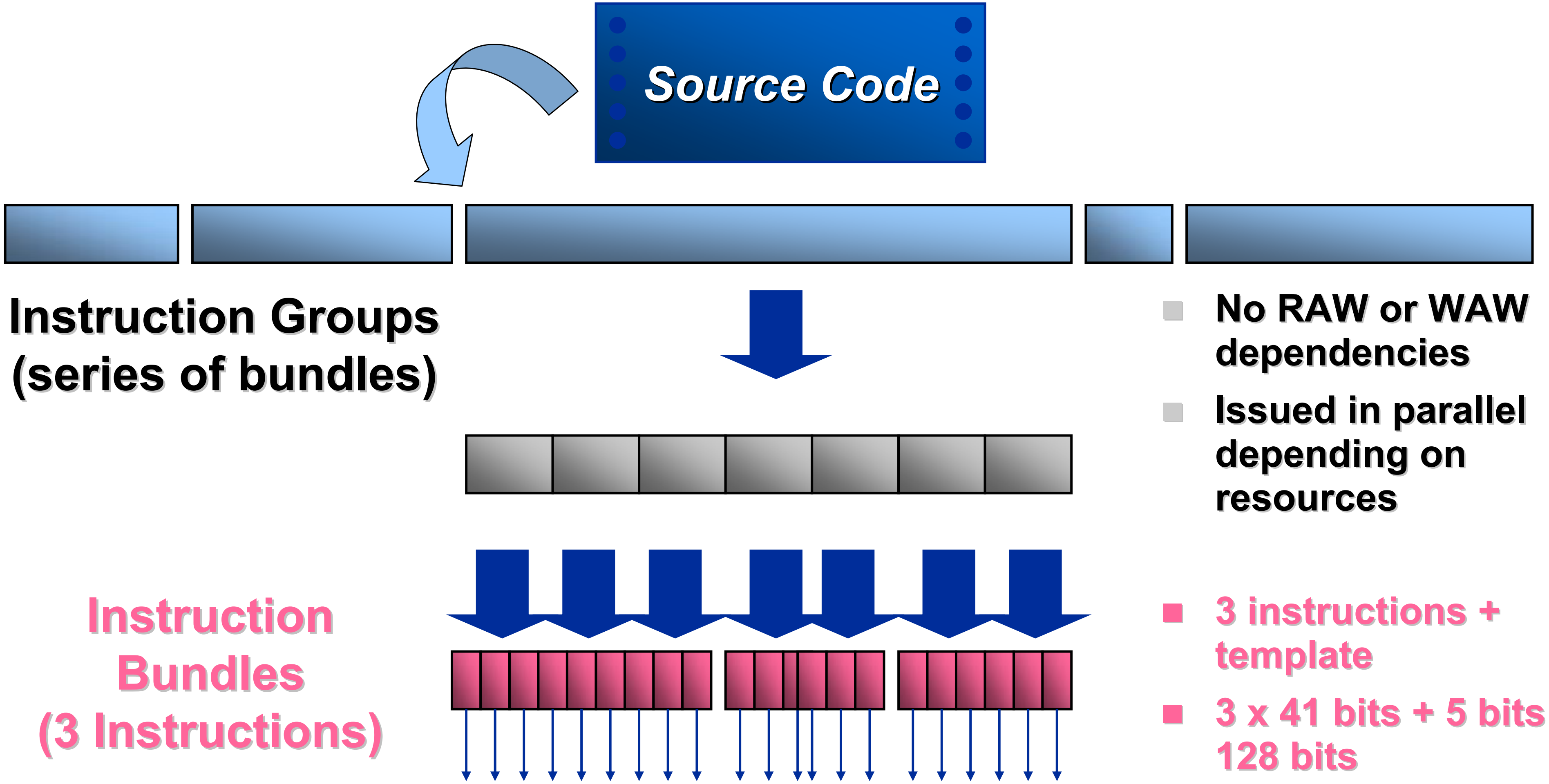
multiple functional units

Itanium Architecture compiler views wider scope

More efficient use of execution resources

Enhances Parallel Execution

EPIC Instruction Parallelism



Instruction Groups (series of bundles)

- No RAW or WAW dependencies
- Issued in parallel depending on resources

Instruction Bundles (3 Instructions)

- 3 instructions + template
- 3 x 41 bits + 5 bits = 128 bits

Up to 6 instructions executed per clock

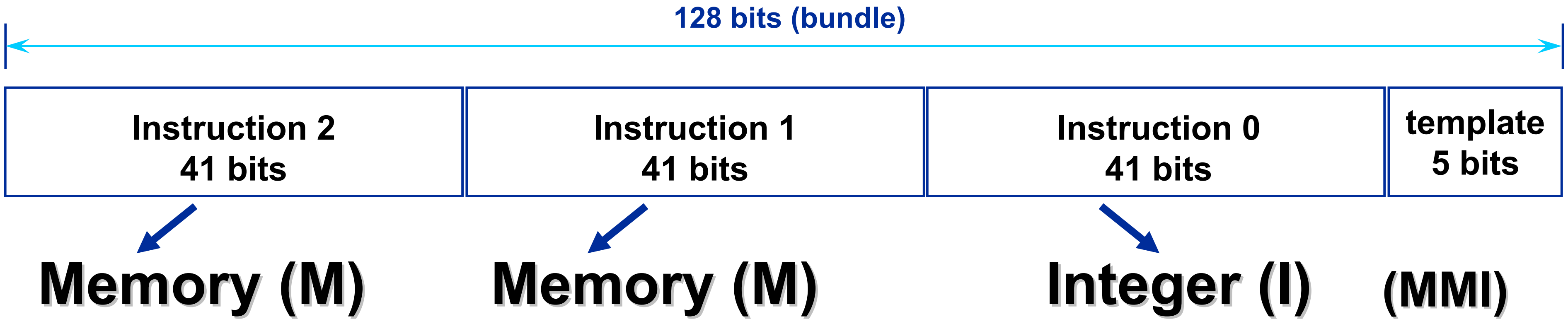
Instruction Level Parallelism

- Instruction Groups
  - No mutual dependencies
  - Delimited by ‘stops’ in assembly code
  - Instructions in groups issued in parallel, depending on available resources.

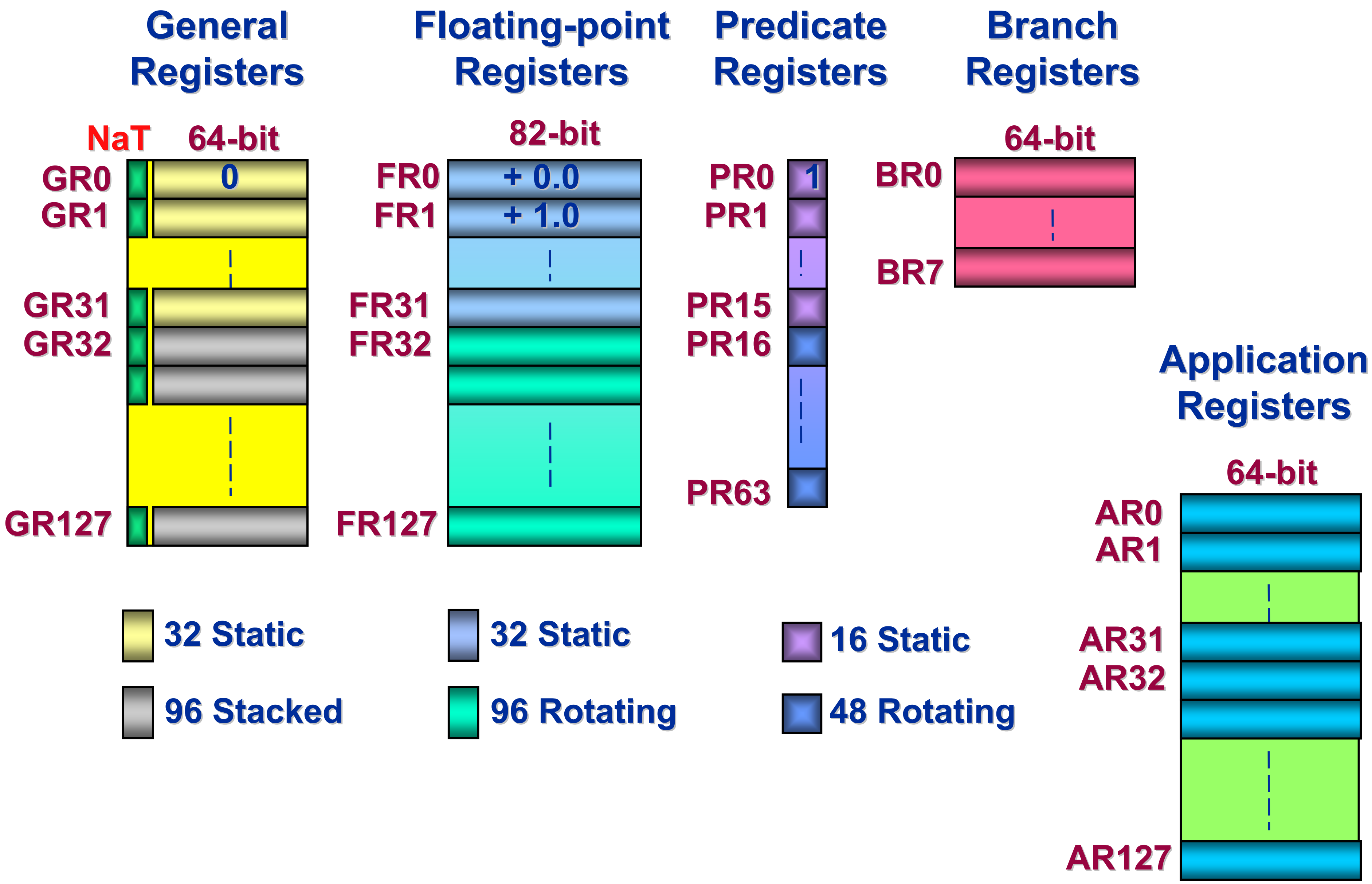
```
instr 1    // 1st. group
instr 2;;  // 1st. group
instr 3    // 2nd. group
instr 4    // 2nd. group
```

- Instruction Bundles
  - 3 instructions and 1 template in 128-bit bundle
  - Instruction dependencies by using ‘stops’
  - Instruction groups can span multiple bundles

```
{ .mii
   ld4 r28=[r8]   // load
   add r9=2,r1    // Int op.
   add r30=1,r1 // Int op.
}
```



Large Register Set



General Registers

NaT 64-bit

GR0 0
GR1
GR31
GR32
GR127

Floating-point Registers

82-bit

FR0 + 0.0
FR1 + 1.0
FR31
FR32
FR127

Predicate Registers

PR0 1
PR1
PR15
PR16
PR63

Branch Registers

64-bit

BR0
BR7

Application Registers

64-bit

AR0
AR1
AR31
AR32
AR127

32 Static

96 Stacked

32 Static

96 Rotating

16 Static

48 Rotating

Predication

- Predicate registers activate/inactivate instructions
- Predicate Registers are set by Compare Instructions
  - Example: `cmp.eq p1,p2 = r2,r3`
- (Almost) all instructions can be predicated:

```
(p1)   ldfd f32=[r32],8
(p2)   fmpy.d f36=f6,f36
```

- Predication:
  - eliminates branching in if/else logic blocks
  - creates larger code blocks for optimization
  - simplifies start up/shutdown of pipelined loops

Predication

- **Code Example: absolute difference of two numbers**

Non-Predicated Pseudo Code

```
        cmpGE r2, r3
        jump_zero P2
P1:     sub  r4 = r2, r3
        jump end
P2:     sub  r4 = r3, r2
end:    ...
```

C Code

```
if  (r2 >= r3)
        r4 = r2 - r3;
else
        r4 = r3 - r2;
```

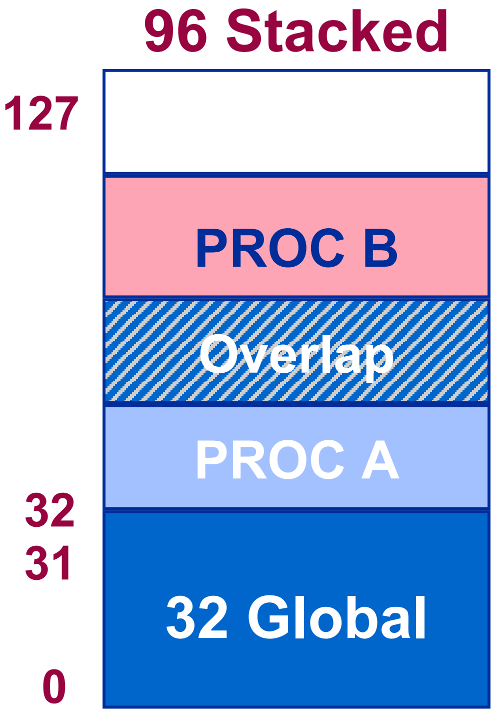
Predicated Assembly Code

```
     cmp.ge p1,p2 = r2,r3 ;;
(p1) sub r4 = r2,r3
(p2) sub r4 = r3, r2
```

Predication Removes Branches, Enables Parallel Execution

Register Stack

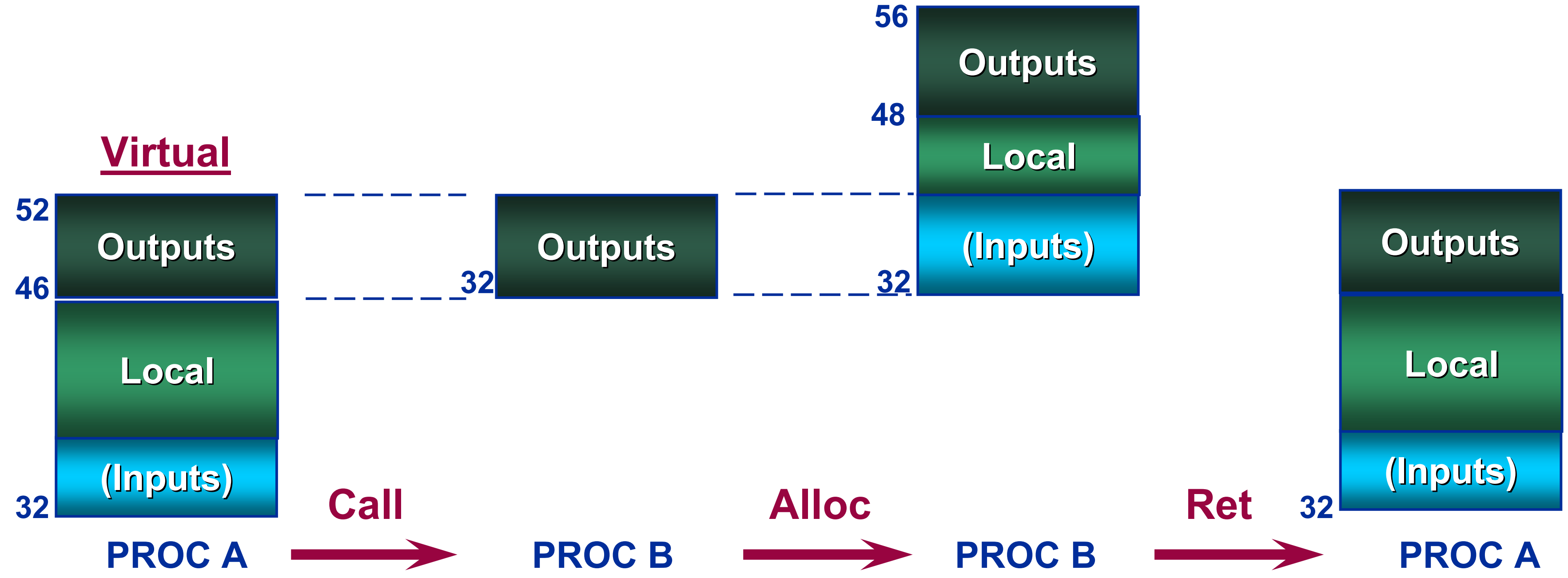
- GRs 0-31 are global to all procedures
- Stacked registers begin at GR32 and are local to each procedure
- Each procedure's register stack frame varies from 0 to 96 registers
- Only GRs implement a register stack
  - The FRs, PRs, and BRs are global to all procedures
- Register Stack Engine (RSE)
  - Upon stack overflow/underflow, registers are saved/restored to/from a backing store transparently



Optimizes the Call/Return Mechanism

Register Stack Engine at Work

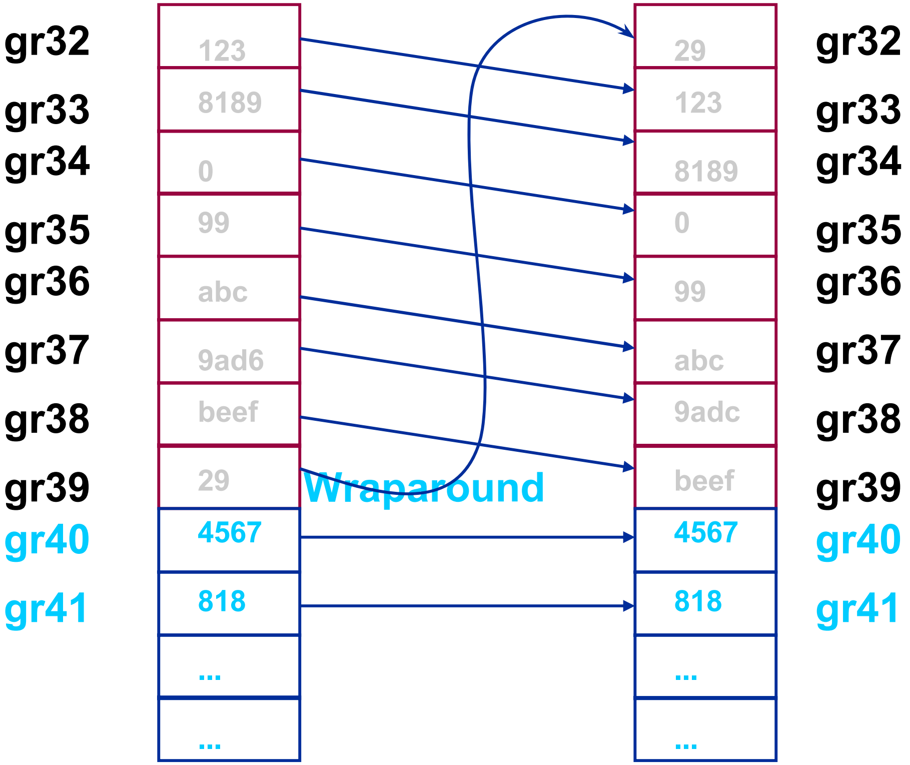
- Call changes frame to contain only the caller's output
- Alloc instr. sets the frame region to the desired size
  - Three architecture parameters: local, output, and rotating
- Return restores the stack frame of the caller



Can improve performance for OOP (C++, Java)

Register rotation

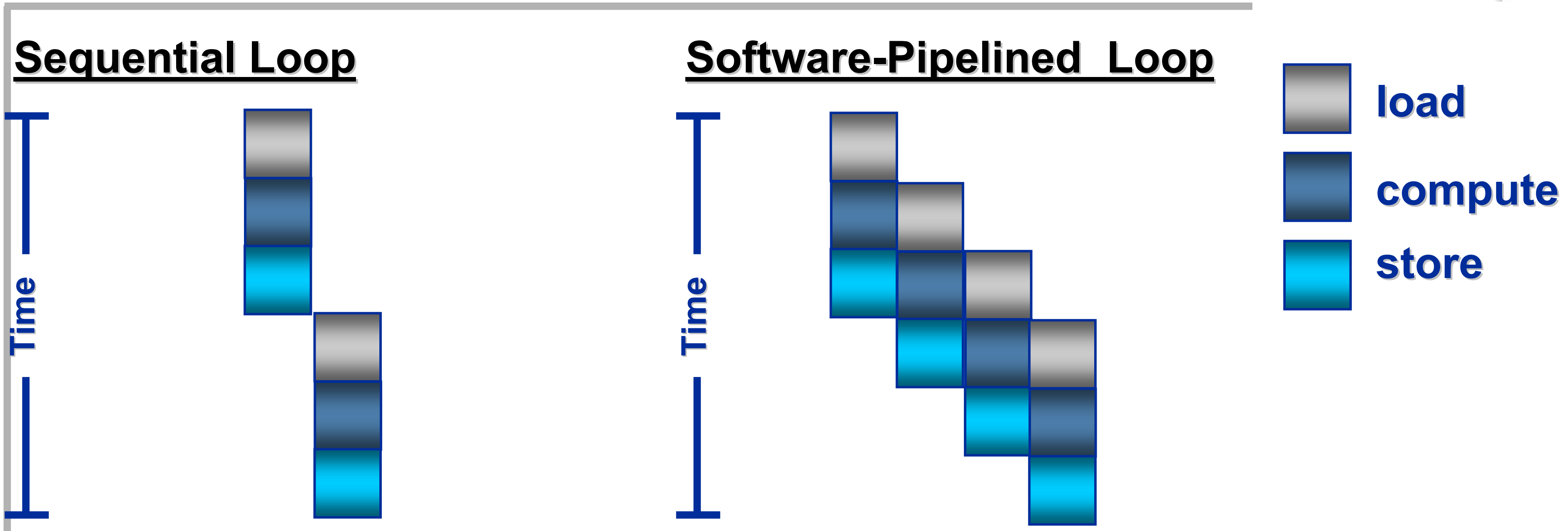
- **Example: 8 general registers rotating, counted loop (br.ctop)**



	Before:	After br.ctop taken:
gr32	123	29
gr33	8189	123
gr34	0	8189
gr35	99	0
gr36	abc	99
gr37	9ad6	abc
gr38	beef	9adc
gr39	29	beef
gr40	4567	4567
gr41	818	818
	...	...
	...	...

- **Floating point and predicate registers**
  - **Always rotate the same set of registers**
    - **FR 32-127**
  - **Rotate in the same direction as general registers**
    - **highest rotates to lowest register number**
    - **all other values rotate towards larger register numbers**
  - **Rotate at the same time as general registers (at the modulo-scheduled loop instruction)**

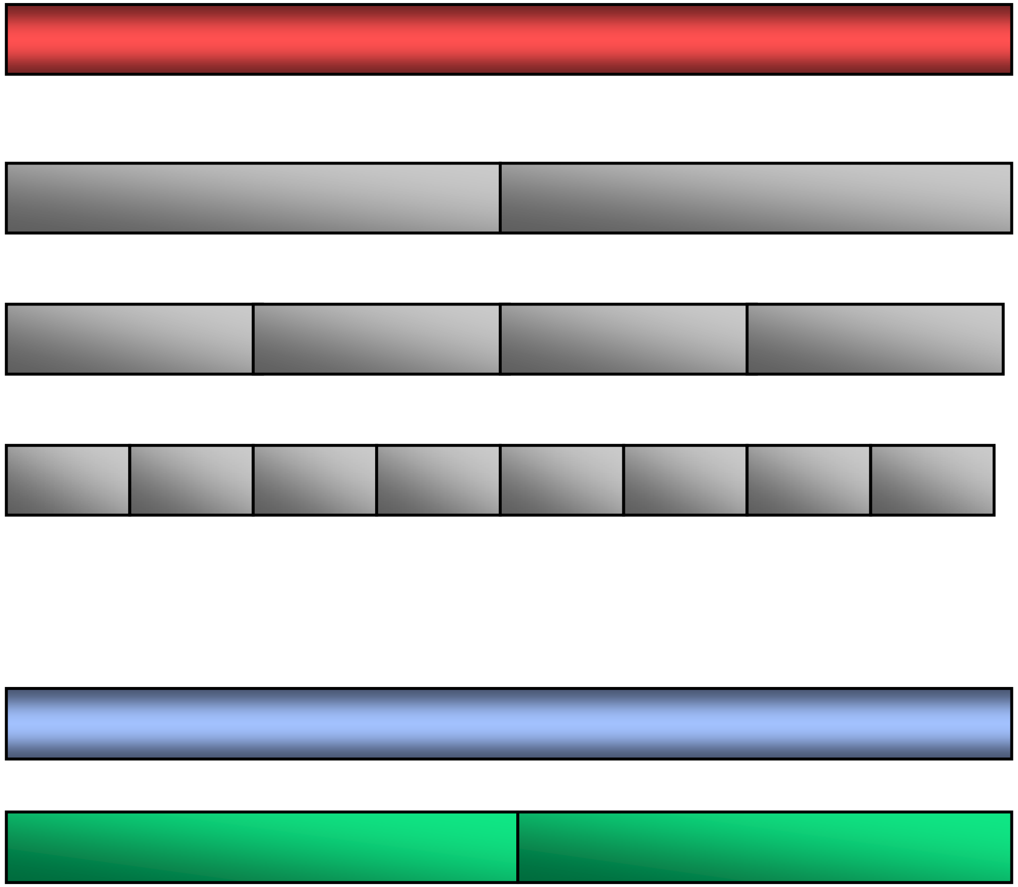
Software Pipelining



- Traditional architectures use loop unrolling
  - Results in code expansion and increased I-cache misses
- Itanium™ Software Pipelining uses rotating registers
  - Allows overlapping execution of multiple loop instances
  - compiler diagnostics help identify pipelining problems (later talk)

Itanium™ provides direct support for Software Pipelining

Itanium® Hardware Data Types



64-bit Integer

2x32-bit SIMD Integer

4x16-bit SIMD Integer

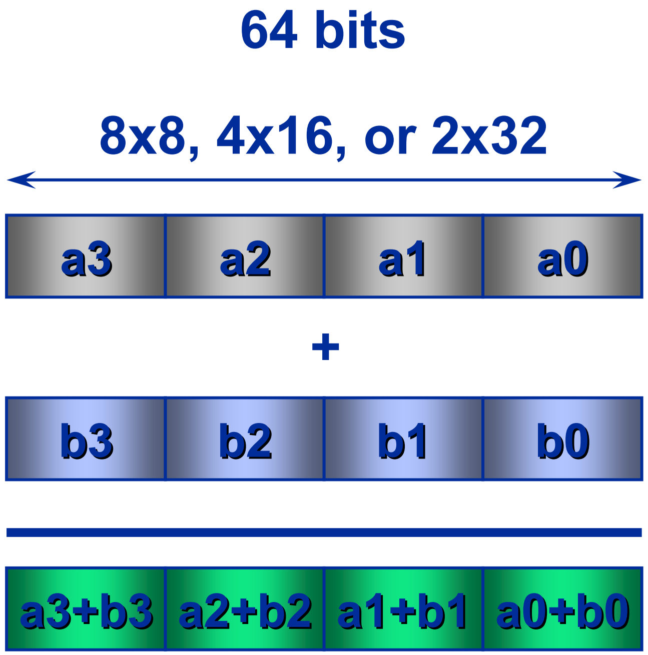
8x8-bit SIMD Integer

64-bit DP F.P.

2x32-bit SP F.P.

SIMD - Integer

- Exploits data parallelism with SIMD (*Single Instruction Multiple Data*)
- Performance boost for audio, video, imaging etc. functions
- GRs treated as 8x8, 4x16, or 2x32 bit elements
- Several instruction types
  - Addition and subtraction, multiply
  - Pack/Unpack
  - Left shift, signed/unsigned right shift
- Compatible with Intel MMX™ Technology



64 bits

8x8, 4x16, or 2x32

a3	a2	a1	a0

+

b3	b2	b1	b0

a3+b3	a2+b2	a1+b1	a0+b0

Available via compiler intrinsics!

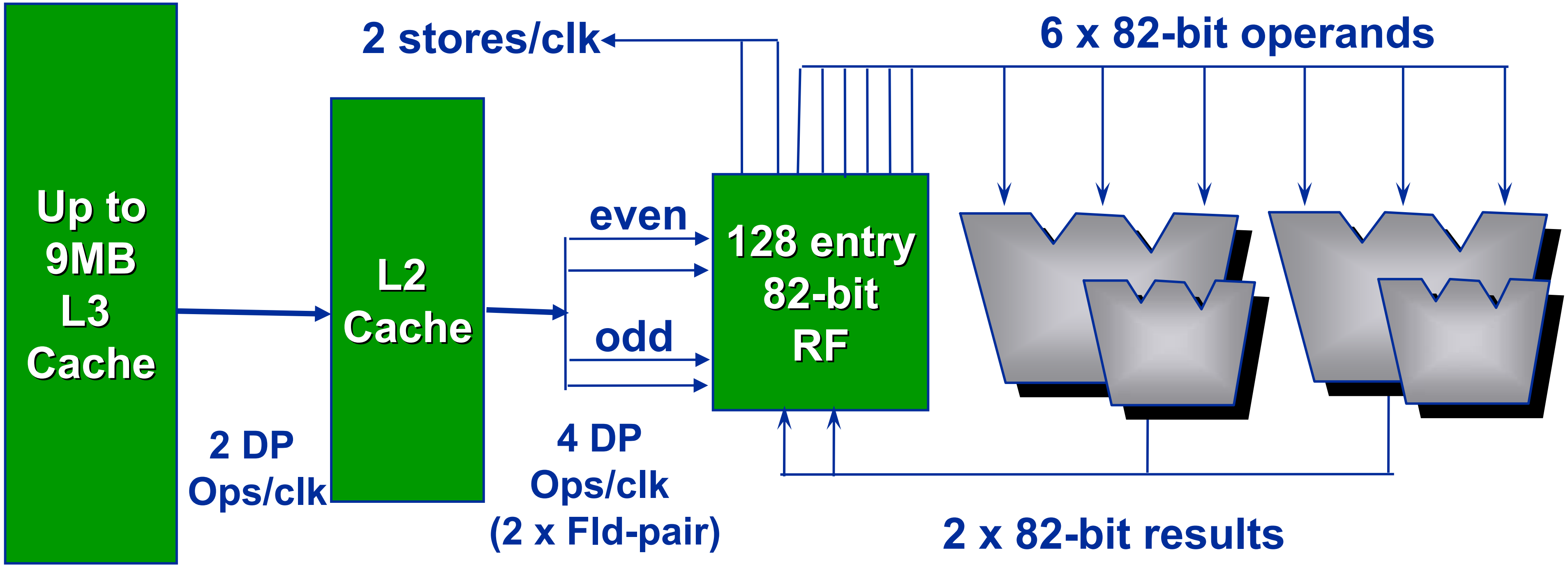
Floating-Point Architecture

- 128 Floating Point registers (82 bit)
  - Single, double, double-extended data types
- Full IEEE.754 compliance

- Arithmetic
  - FMA – Fused Multiply-Add instruction f = a * b + c
  - 2 independent FP units
  - up to 4 DP operations per cycle
  - up to 4 DP FP operands loded per clock
  - SW DIV / SQRT, provide high throughput, take advantage of wide FP machine
  - MAX, MIN instructions for floating point

- Data transfer
  - load, store, GR ⇔ FR conversion; load pair to double data

Floating Point Features



L1D cache not used for FP data!

Some Floating Point Latencies

Operation	Latency
FP Load (L2 Cache hit)	6
FMAC,FMISC	4
FP → Int (getf)	5
Int → FP (setf)	6
Fcmp to branch Fcmp to qual pred	2 2

L2 and L3 Cache

- L2: 256KB, 32GBs, 5-7 clk
  - Data array is banked - 16 banks of 16KB each
  - Montecito: Separation of Data/Instruction cache
- Non-blocking / out-of-order
  - L2 queue (32 entries) - holds all in-flight load/stores
  - out-of-order service - smoothes over load/store/bank conflicts, fills
  - Can issue/retire 4 stores/loads per clock
  - Can bypass L2 queue (5,7,9 clk bypass) if
    - no address or bank conflicts in same issue group
    - no prior ops in L2 queue want access to L2 data arrays
- Up to 9 (12) MB L3, 32GBs,12-13 clk cache on die
  - Single ported – full cache line transfers

Itanium® 2 Caches – Summary

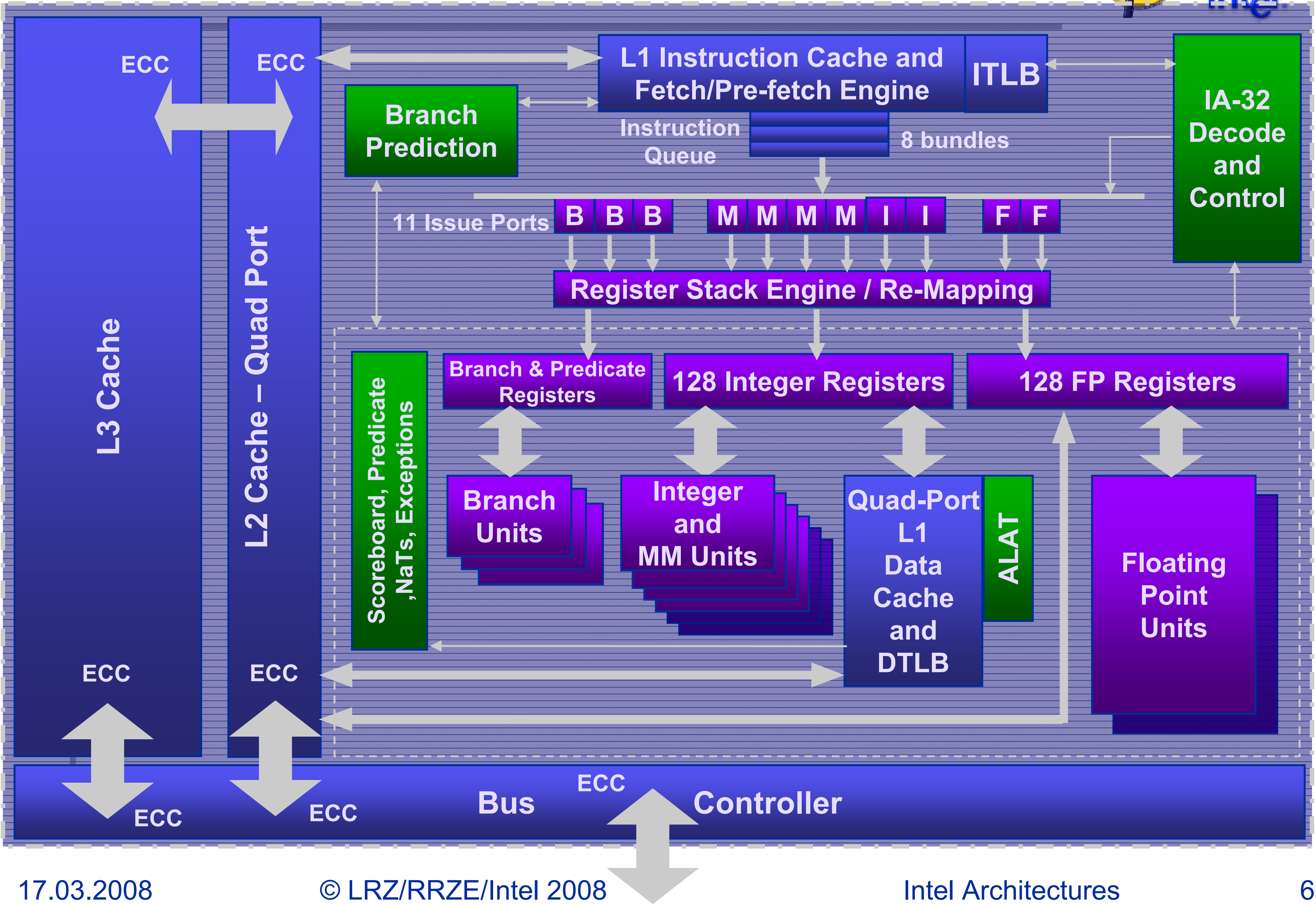
	L1I	L1D	L2	L3
Size	16K	16K	256K (1M L2I on Montecito)	1.5/3/6M/9M on die
Line Size	64B	64B	128B	128B
Ways	4	4	8	4 or 2 per MB
Replacement	LRU	NRU	NRU	NRU
Latency (load to use)	I-Fetch:1	INT:1	INT: 5 FP: 6	12/13
Write Policy	-	WT (RA)	WB (WA+RA)	WB (WA)
Bandwidth	R: 32 GBs	R: 16 GBs W: 16 GBs	R: 32 GBs W: 32 GBs	R: 32 GBs W: 32 GBs

TLBs

- 2-level TLB hierarchy
  - DTC/ITC (32/32 entry, fully associative, 0.5 clk)
    - Small fast translation caches tied to L1D/L1I
      - Key to achieving very fast 1-clk L1D, L1I cache accesses
  - DTLB/ITLB (128/128 entry, fully associative, 1 clk)
    - All architected page sizes (4K to 4GB)
    - standard page size = 16 kB → TLB can cache 2 MB
    - Supports up to 64/64 ITR/DTRs
    - TLB miss starts hardware page walker

Small fast TLBs enable low latency caches, but TLB misses might be an issue

Itanium® 2 Block Diagram

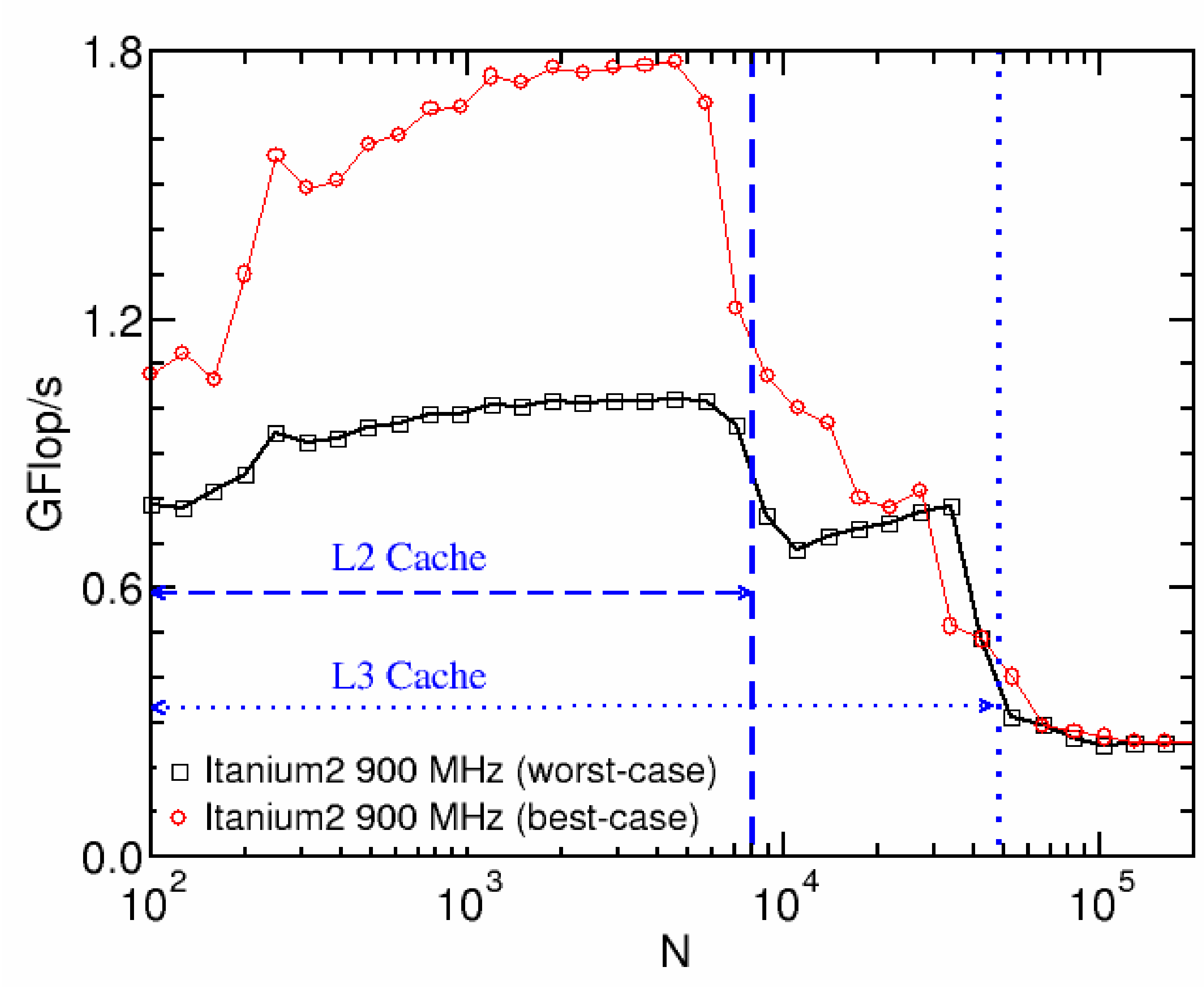


Itanium2: Intel´s current 64 Bit processor
Application Performance Peculiarities

- Some dos and don'ts for Itanium2

  - Do try to use FMA where possible; formulate your inner loops in an FMA-friendly way
  - Avoid very short, tight inner loops; if a short trip count cannot be avoided, try to unroll outer loops to make the body fatter
  - When working in L2 cache, try different array paddings; due to banked L2 layout, significant performance boosts can be achieved by not hitting the same bank in every loop iteration
  - With current Intel compilers, try to avoid too many synchronization points in OpenMP programs – locks and barriers tend to be slow
  - Use !DIR$ IVDEP when applicable (indirect array access)

Itanium2: Intel´s current 64 Bit processor
Triads Performance at Different Array Paddings



References

- R. Gerber: *The Software Optimization Cookbook.* High Performance Recipes for the Intel Architecture. Intel Press (2002)
  - good introduction, must be complemented with compiler and architecture documentation
- W. Triebel et al: *Programming Itanium-based Systems.* Developing High Performance Applications for Intel‘s New Architecture. Intel Press (2001)
  - extremely detailed, suitable for assembler programmers
  - slightly outdated
- `http://developer.intel.com/`
  - tutorials, manuals, white papers, discussion forums etc.
- c‘t Magazine 13/2003, several articles (25th birthday of Intel‘s x86 architecture)