Intel® Architecture and Tools JURECA - Tuning for the platform II

Presenter: Dr. Heinrich Bockhorst

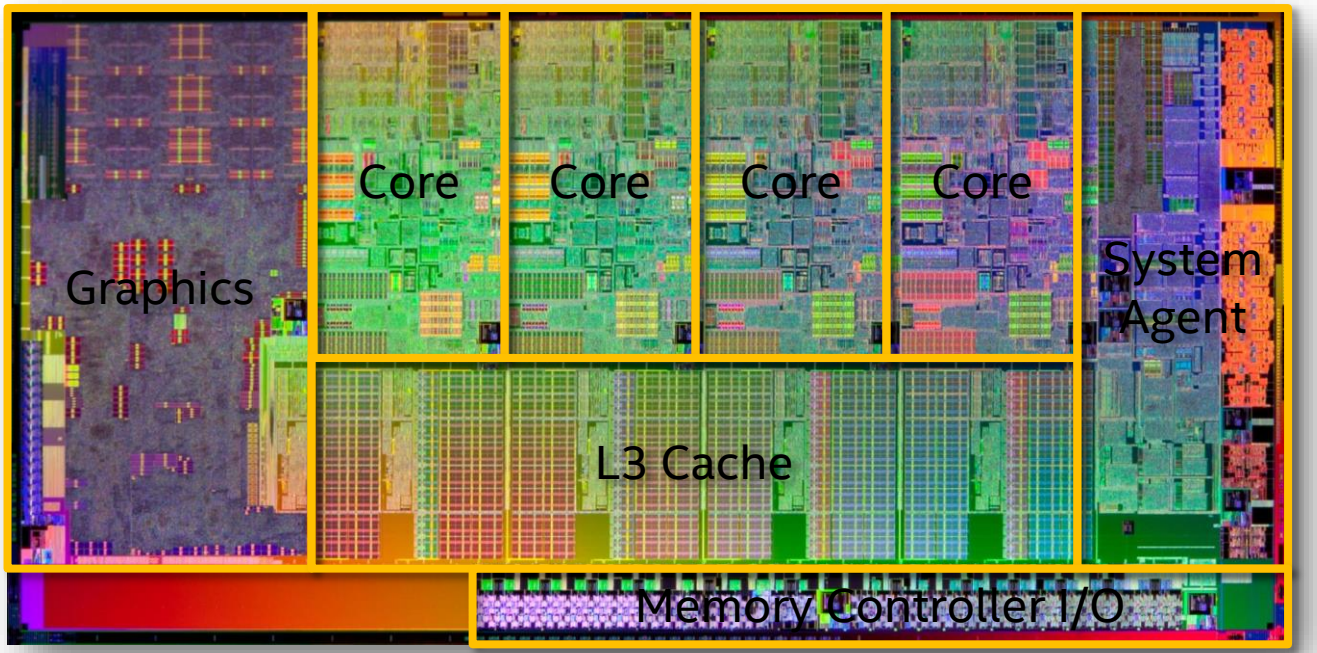
Date: 18-05-2015

Agenda

- **Introduction**
- Processor Architecture Basics
- Composer XE
- Selected Intel® Tools

Chips and Dies

- A chip is the package containing one or more dies (silicon)
- Major components of the die can be easily identified
- Example of a typical die:



"Die shot" of Intel® Core™ Processor

Moore's "Law"

"The number of transistors on a chip will **double** approximately **every two years.**"

[Gordon Moore]

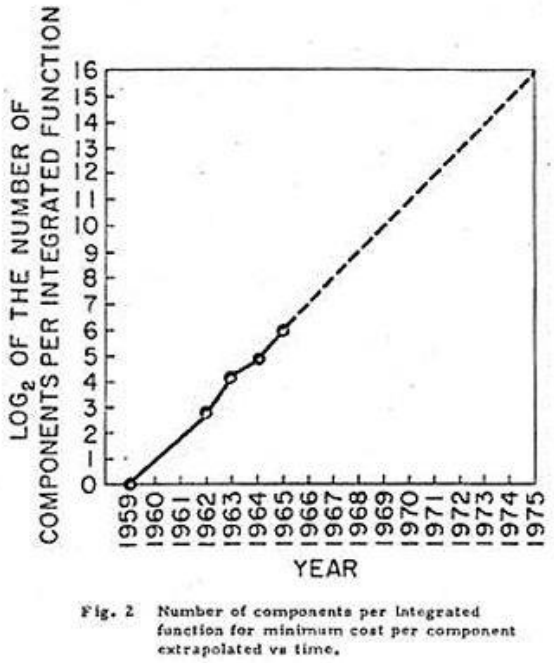
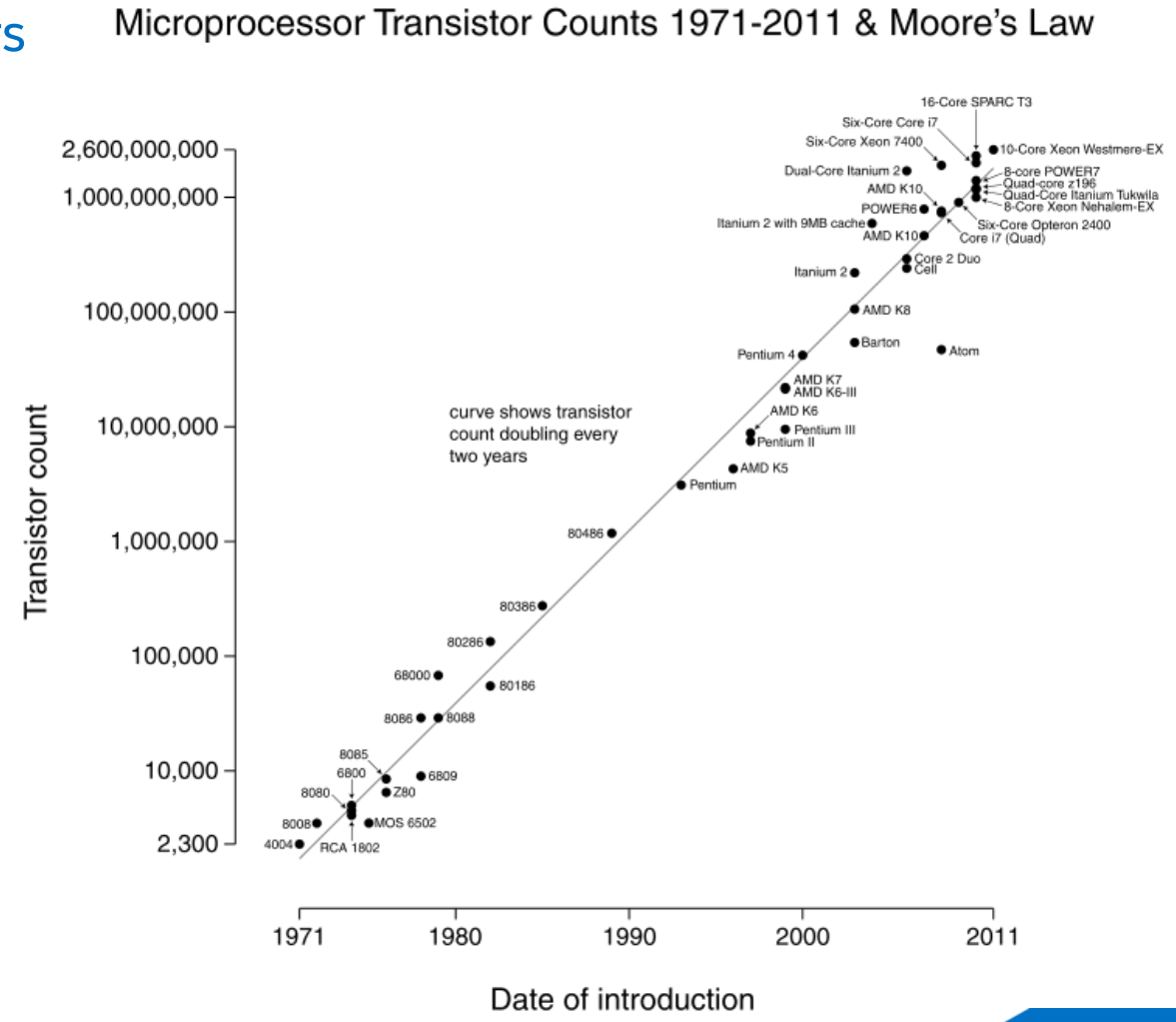


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Moore's Law graph, 1965



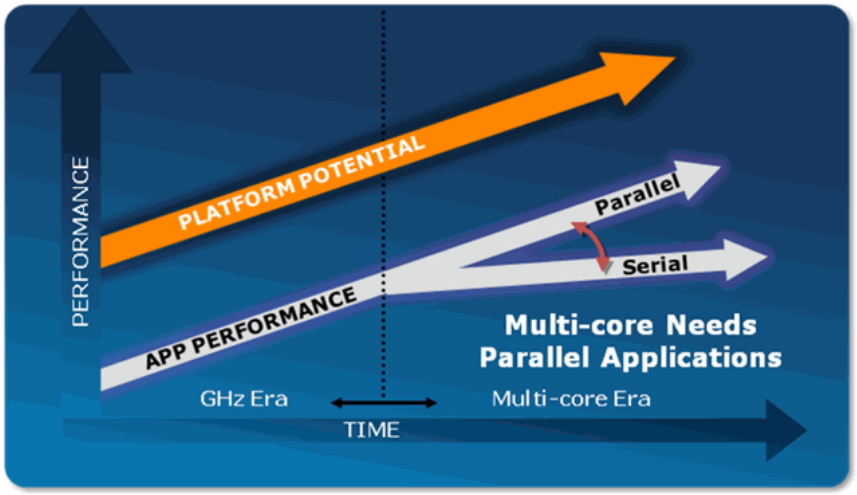
Source: http://en.wikipedia.org/wiki/Moore's_law

Parallelism

Problem:
Economical operation **frequency of (CMOS) transistors is limited**.
⇨ **No free lunch anymore!**

Solution:
More transistors allow more gates/logic on the same die space and power envelop, **improving parallelism**:

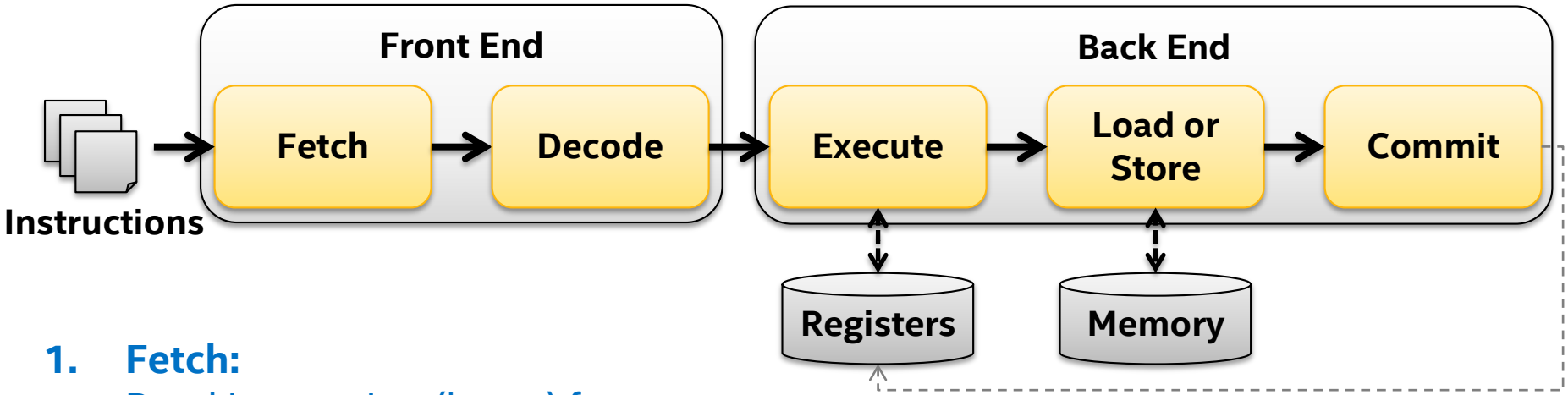
- **Thread level** parallelism (TLP):
  Multi- and many-core
- **Data level** parallelism (DLP):
  Wider vectors (SIMD)
- **Instruction level** parallelism (ILP):
  Microarchitecture improvements, e.g. threading, superscalarity, …



Processor Architecture Basics

Pipeline

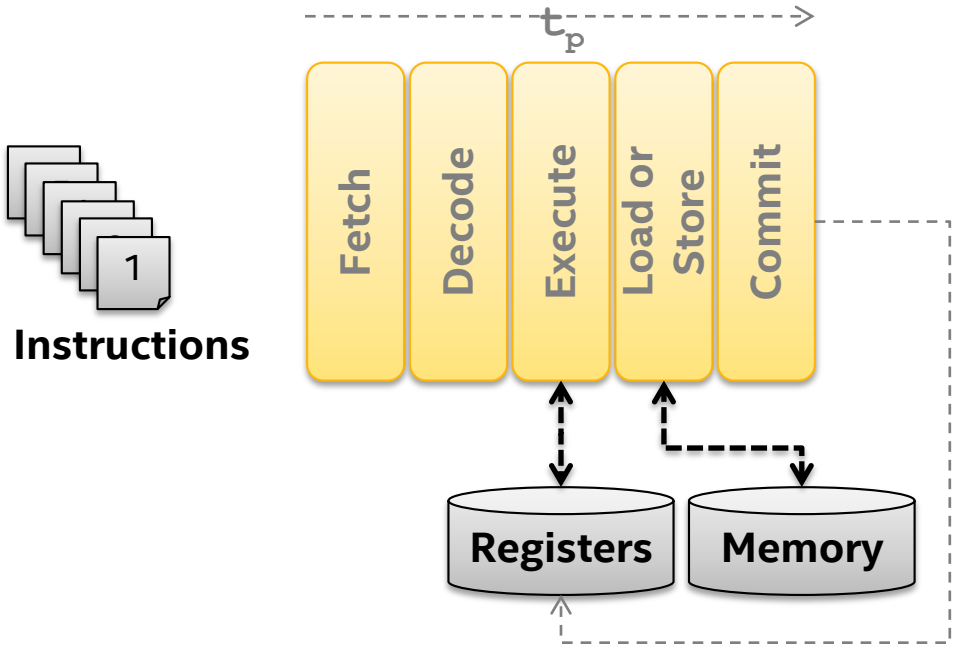
Computation of instructions requires several stages:



1. **Fetch:**
   Read instruction (bytes) from memory
2. **Decode:**
   Translate instruction (bytes) to microarchitecture
3. **Execute:**
   Perform the operation with a functional unit
4. **Memory:**
   Load (read) or store (write) data, if required
5. **Commit:**
   Retire instruction and update micro-architectural state

Processor Architecture Basics

Naïve Pipeline: Serial Execution



Characteristics of strict serial execution:

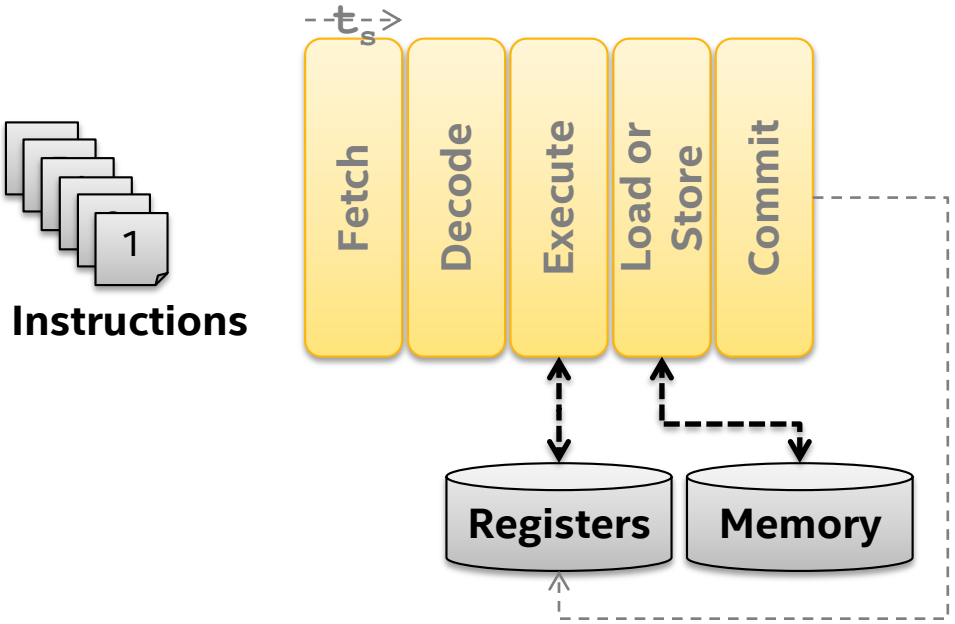
- Only one instruction for the entire pipeline (all stages)
- Low complexity
- Execution time: $n_{instructions} * t_p$

Problem:

- Inefficient because only one stage is active at a time

Processor Architecture Basics

Pipeline Execution



Characteristics of pipeline execution:

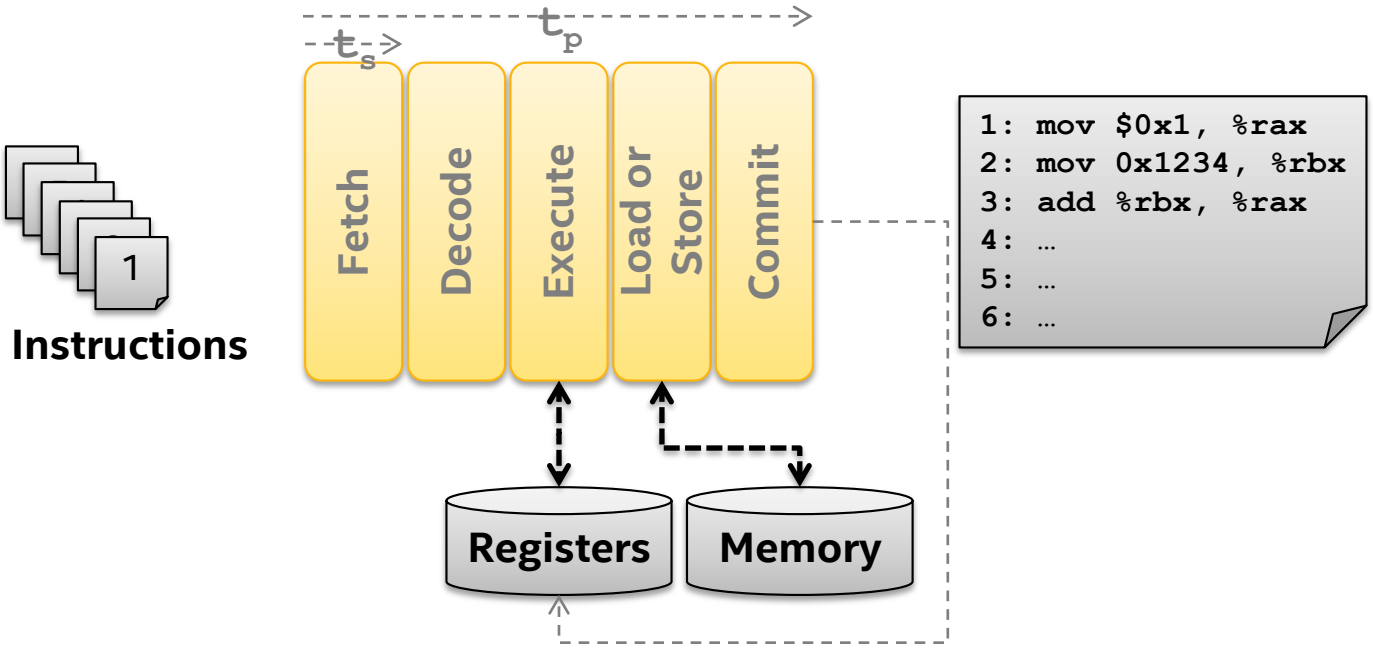
- Multiple instructions for the entire pipeline (one per stage)
- Efficient because all stages kept active at every point in time
- Execution time: $n_{instructions} * t_s$

Problem:

- Reality check: What happens if t_s is not constant?

Processor Architecture Basics

Pipeline Stalls



Pipeline stalls:

- Caused by pipeline stages to take longer than a cycle
- Caused by dependencies: order has to be maintained
- Execution time: $n_{instructions} * t_{avg}$ with $t_s \leq t_{avg} \leq t_p$

Problem:

- Stalls slow down pipeline throughput and put stages idle.

Processor Architecture Basics

Reducing Impact of Pipeline Stalls

Impact of pipeline stalls can be reduced by:

- Branch prediction
- Superscalarity + multiple issue fetch & decode
- Out of Order execution
- Cache
- Non-temporal stores
- Prefetching
- Line fill buffers
- Load/Store buffers
- Alignment
- Simultaneous Multithreading (SMT)

⇨ **Characteristics of the architecture that might require user action!**

Processor Architecture Basics

Superscalarity

Characteristics of a superscalar architecture:

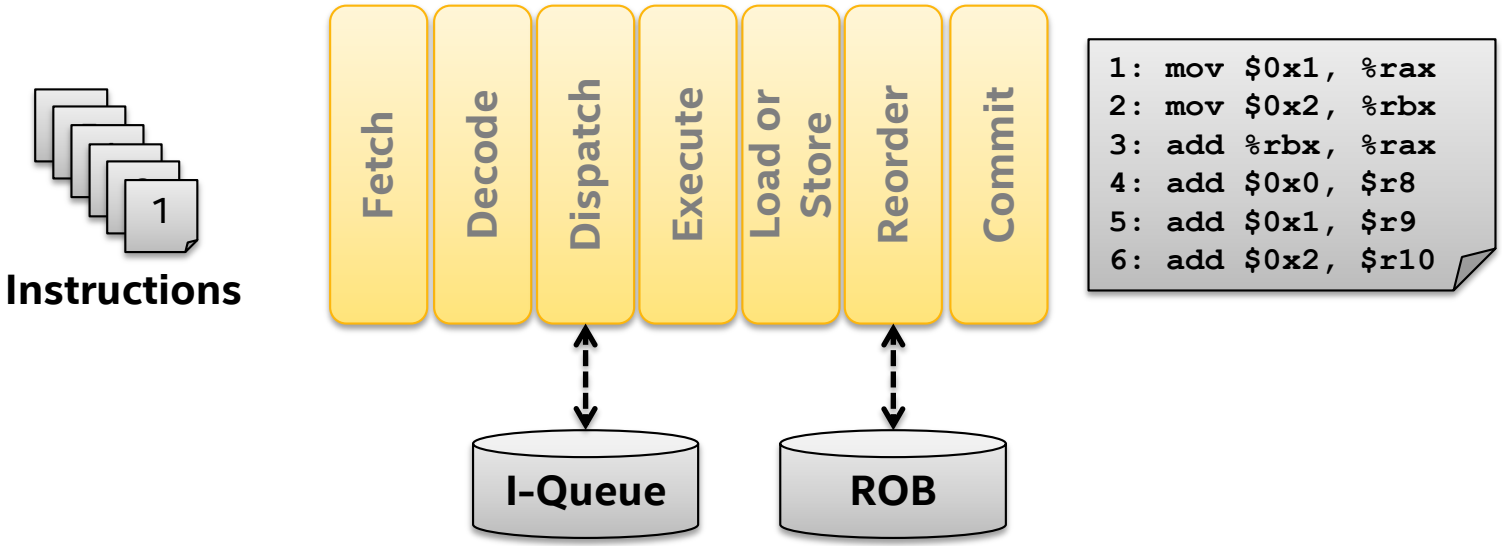
- Improves throughput by covering latency (ports are independent)
- Ports can have different functionalities (floating point, integer, addressing, …)
- Requires multiple issue fetch & decode (here: 2 issue)
- Execution time: $n_{instructions} * t_{avg} / n_{ports}$

Problem:

- More complex and prone in case of dependencies
  ⇨ Solution: Out of Order Execution

Processor Architecture Basics

Out of Order Execution



Characteristics of out of order (OOO) execution:

- Instruction queue (I-Queue) moves stalling instructions out of pipeline
- Reorder buffer (ROB) maintains correct order of committing instructions
- Reduces pipeline stalls, but not entirely!
- Speculative execution possible
- Opposite of OOO execution is in order execution

Processor Architecture Basics

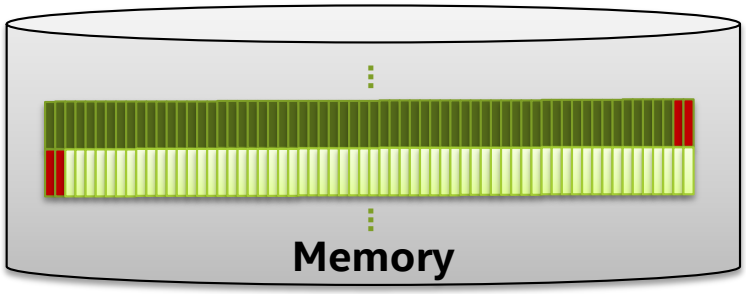
Alignment

What happens if data spawns across cache-lines?

- Loads/stores require all cache-lines of a datum to be loaded into cache.
- In worst case this doubles load/store times:

```
struct {
… // 62 byte
int x; // 4 byte
} data;
```

```
STRUCTURE /data/
  … ! 62 byte
  INTEGER id ! 4 byte
END STRUCTURE
```



Cache-line `n`

Cache-line `n+1`

Both cache-lines **`n`** and **`n+1`** need to be loaded up to L1 cache to access **`x`**!

Align data by padding or explicitly by compiler attributes to fit into exactly one cache-line, e.g.:

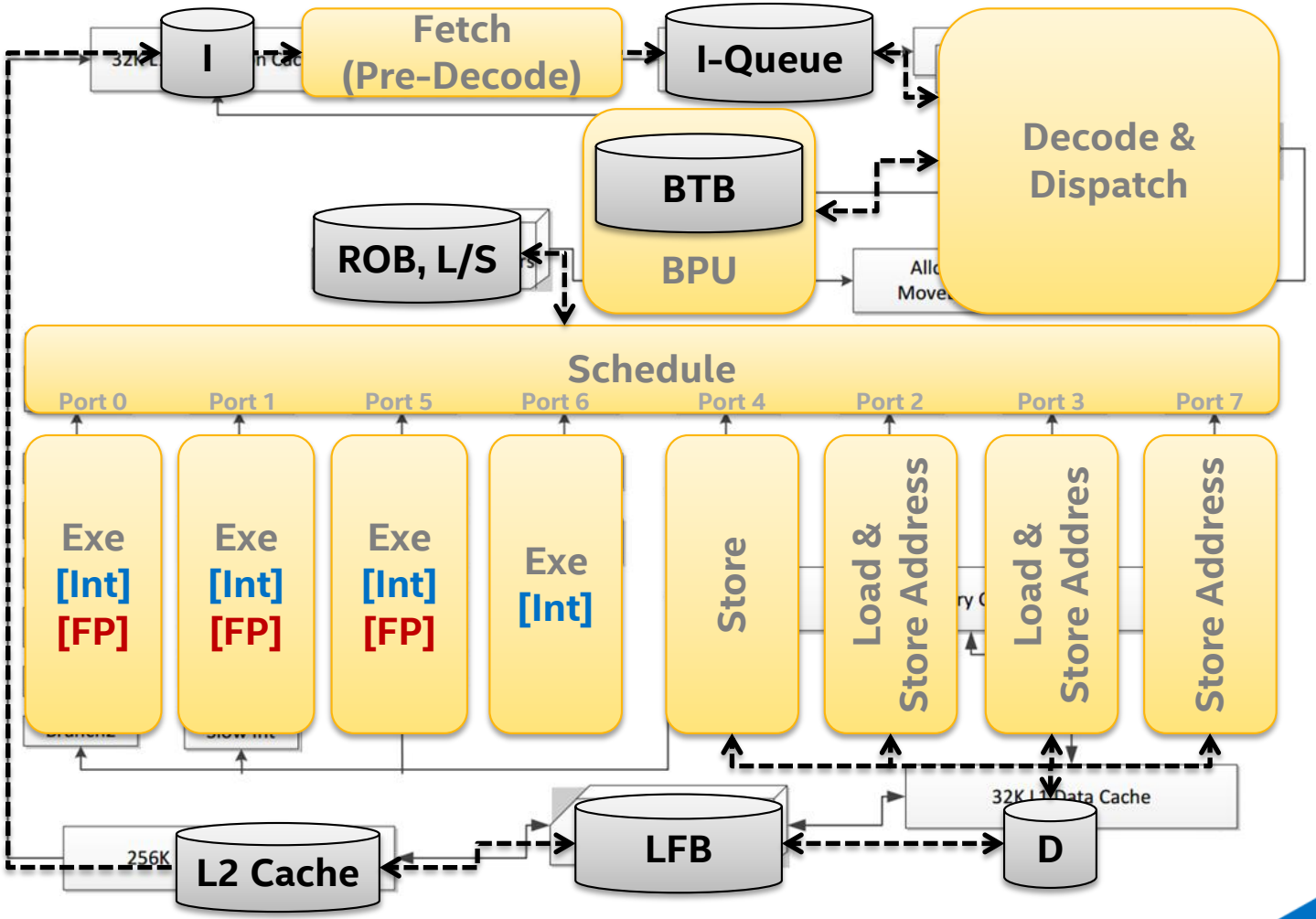
```
struct {
… // 62 byte
… // 2 byte
int x; // 4 byte
} data;
```

```
STRUCTURE /data/
  … ! 62 byte
  … ! 2 byte
  INTEGER id ! 4 byte
END STRUCTURE
```

Processor Architecture Basics

Example: 4th Generation Intel® Core™

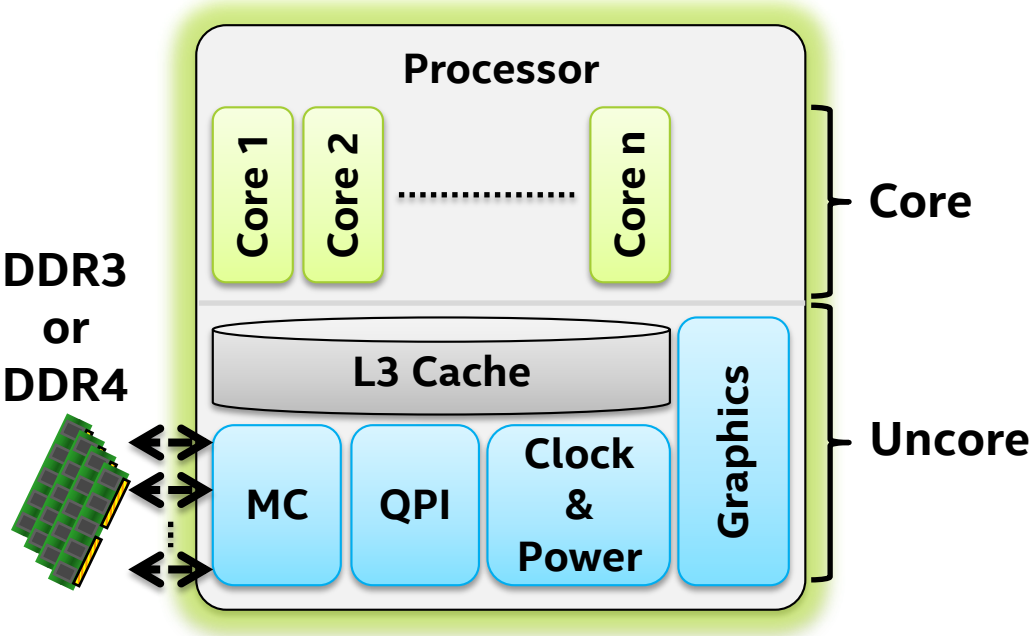
From Intel® 64 and IA-32 Architectures Optimization Reference Manual:



Processor Architecture Basics

Core vs. Uncore

- **Core:**
  Processor core's logic:
  - Execution units
  - Core caches (L1/L2)
  - Buffers & registers
  - …

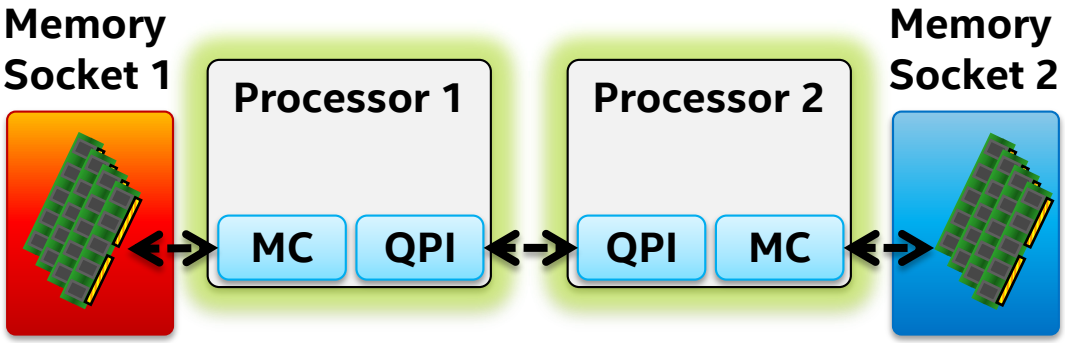


- **Uncore:**
  All outside a processor core:
  - Memory controller/channels (MC) and Intel® QuickPath Interconnect (QPI)
  - L3 cache shared by all cores
  - Type of memory
  - Power management and clocking
  - Optionally: Integrated graphics

⇨ **Only uncore is differentiation within same processor family!**

Processor Architecture Basics

UMA and NUMA



- **UMA (aka. non-NUMA):**
  - Uniform Memory Access (UMA)
  - Addresses interleaved across memory nodes **by cache line**
  - Accesses may or may not have to cross QPI link
    - ⇨ **Provides good portable performance without tuning**

- **NUMA:**
  - Non-Uniform Memory Access (NUMA)
  - Addresses not interleaved across memory nodes by cache line
  - Each processor has direct access to contiguous block of memory
    - ⇨ **Provides peek performance but requires special handling**

Processor Architecture Basics

NUMA - Thread Affinity & Enumeration

Non-NUMA:

Thread affinity **might** be beneficial (e.g. cache locality) but not required

NUMA:

Thread affinity is **required**:

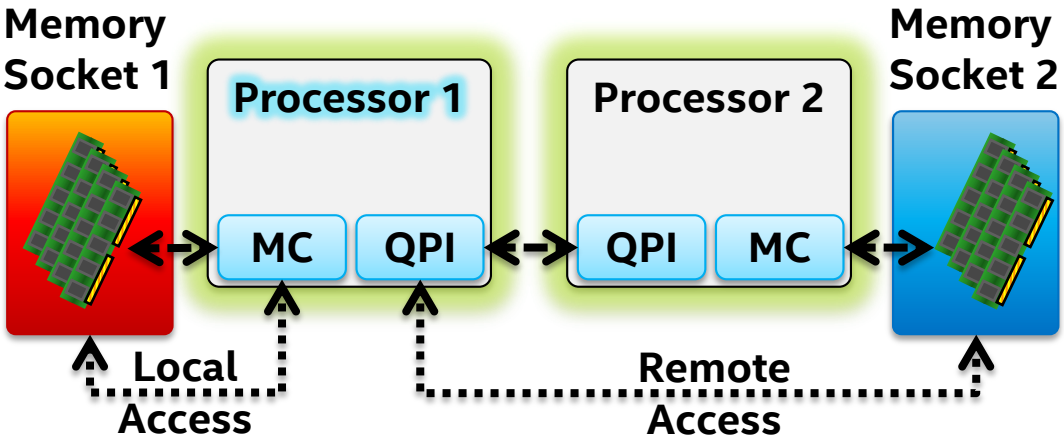
- Improve accesses to local memory vs. remote memory
- Ensure 3rd party components support affinity mapping, e.g.:
  - Intel® OpenMP* via **`$KMP_AFFINITY`** or **`$OMP_PLACES`**
  - Intel® MPI via **`$I_MPI_PIN_DOMAIN (default may be OK)`**
    `tool cpuinfo provides additional information`
  - …
- Right way to get enumeration of cores:
  Intel® 64 Architecture Processor Topology Enumeration
  https://software.intel.com/en-us/articles/intel-64-architecture-processor-topology-enumeration

Processor Architecture Basics

NUMA - Memory, Bandwidth & Latency

Memory allocation:

- Differentiate: implicit vs. explicit memory allocation
- Explicit allocation with NUMA aware libraries, e.g. **`libnuma`** (Linux*)
- Bind **memory ⇔ (SW) thread**, and **(SW) thread ⇔ processor**
- More information on optimizing for performance: https://software.intel.com/de-de/articles/optimizing-applications-for-numa



Performance:

- Remote memory access **latency ~1.7x** greater than local memory
- Local memory **bandwidth** can be up to **~2x** greater than remote

Clock Rate and Power Gating

Control power utilization and performance:

- Clock rate:
  - Idle components can be clocked lower:
    Save energy
    ⇨ Intel SpeedStep®
  - If thermal specification allows, components can be over-clocked:
    Higher performance
    ⇨ Intel® Turbo Boost Technology
  - Base frequency at P1; P0 might be turbo, depending on processor to decide (how many cores/GPU are active)
  - Intel® Turbo Boost Technology 2.0: processor can decide whether turbo mode can **exceed** the TDP with higher frequencies

- Power gating:
  Turn off components to save power

- P-state: Processor state; low latency; combined with speed step
- C-state: chip state; longer latency, more aggressive static power reduction

Documentation

Intel® 64 and IA-32 Architectures Software Developer Manuals:

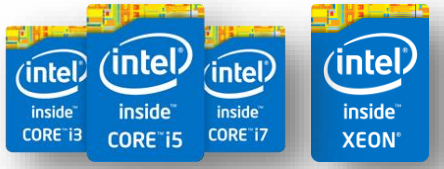
- Intel® 64 and IA-32 Architectures Software Developer's Manuals
  - Volume 1: Basic Architecture
  - Volume 2: Instruction Set Reference
  - Volume 3: System Programming Guide
- Software Optimization Reference Manual
- Related Specifications, Application Notes, and White Papers

https://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals

Intel® Processor Numbers (who type names are encoded):
http://www.intel.com/products/processor_number/eng/

Desktop, Mobile & Server

Tick/Tock Model



Intel® Core™	2nd Generation Intel® Core™	3rd Generation Intel® Core™	4th Generation Intel® Core™
Nehalem (2008) New Microarchitecture 45nm **Tock**	**Westmere (2010)** New Process Technology 32nm **Tick**	**Sandy Bridge (2011)** New Microarchitecture 32nm **Tock**	**Ivy Bridge (2012)** New Process Technology 22nm **Tick**	**Haswell (2013)** New Microarchitecture 22nm **Tock**

Future			
Broadwell (2014) New Process Technology 14nm **Tick**	**Skylake** New Microarchitecture 14nm **Tock**	New Process Technology 11nm **Tick**	New Microarchitecture 11nm **Tock**

Tock: Innovate

Tick: Shrink

5/22/2015

Desktop, Mobile & Server

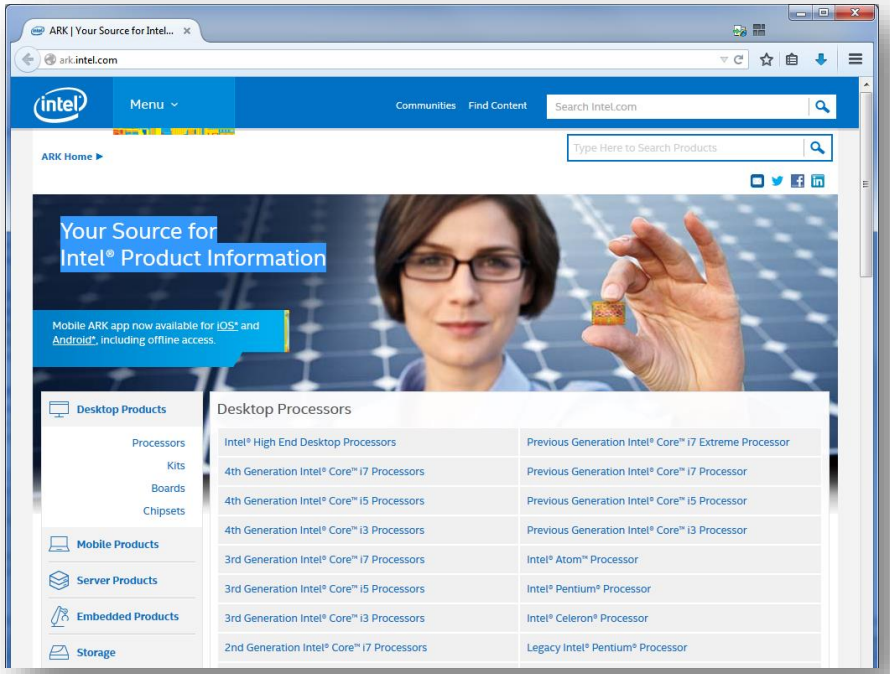
Your Source for Intel® Product Information

Naming schemes:

- Desktop & Mobile:
  - **Intel® Core™ i3/i5/i7** processor family
  - 4 generations, e.g.:
    4th Generation Intel® Core™ i7-XXXX

- Server:
  - **Intel® Xeon® E3/E5/E7** processor family
  - 3 generations, e.g.:
    Intel® Xeon® Processor E3-XXXX v3

Information about available Intel products can be found here: http://ark.intel.com/



Desktop, Mobile & Server
Characteristics

- Processor core:
  - **4 issue**
  - Superscalar **out-of-order** execution
  - Simultaneous multithreading:
    Intel® Hyper-Threading Technology with **2 HW threads per core**

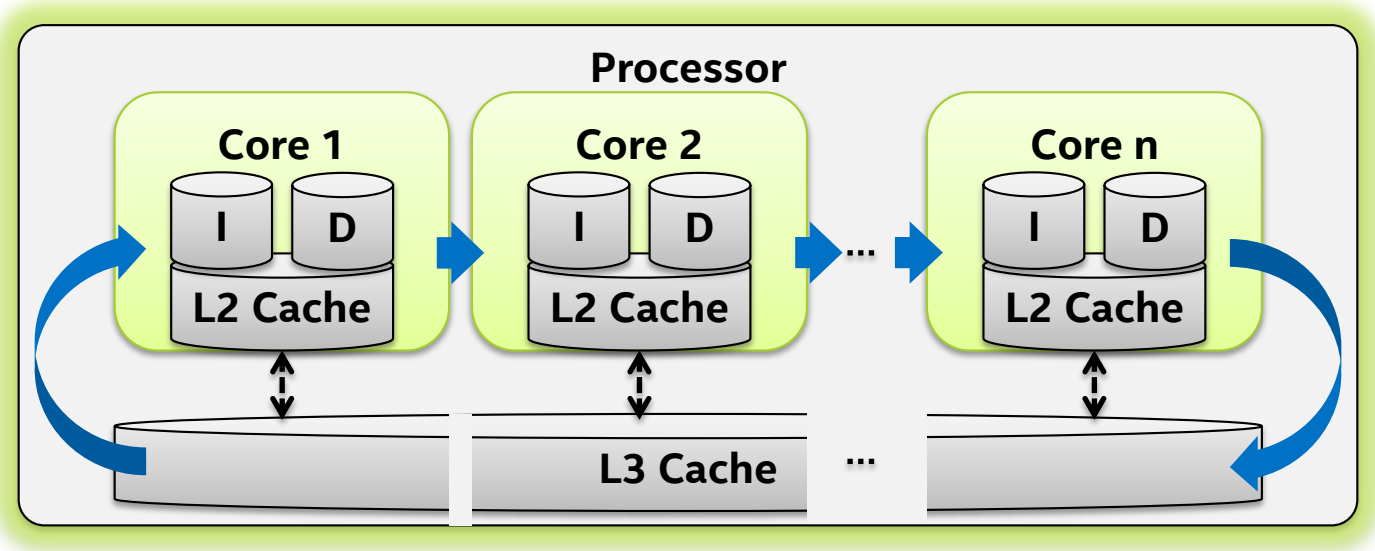
- Multi-core:
  - Intel® Core™ processor family: up to 8 cores (i7-5960) -- mobile desktop
  - Intel® Xeon® processor family: up to 18 cores (E5-2693 v3) – server
    all even numbers from 4-18 available

- Caches:
  - **Three level** cache hierarchy – L1/L2/L3 (Nehalem and later)
  - 64 byte cache line

Desktop, Mobile & Server

Caches

Cache hierarchy:



Level	Latency (cycles)	Bandwidth (per core per cycle)	Size
L1-D	4	2x 16 bytes	32KiB
L2 (unified)	12	1x 32 bytes	256KiB
L3 (LLC)	26-31	1x 32 bytes	varies (≥ 2MiB per core)
L2 and L1 D-Cache in other cores	43 (clean hit), 60 (dirty hit)		

Desktop, Mobile & Server

Intel® Hyper-Threading Technology II

- Not shared are:
  - Registers
  - Architectural state
- Smaller extensions needed:
  - More uops in backend need to be handled
  - ROB needs to be increased
- Easy and efficient to implement:
  - Low die cost: Logic duplication is minimal
  - Easy to handle for a programmer (multiple SW threads)
  - Can be selectively used, depending on workload
  - Many workloads can benefit from SMT – just enable it
- More insights to Intel® Hyper-Threading Technology: https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology

Desktop, Mobile & Server
Performance

- Following Moore's Law:

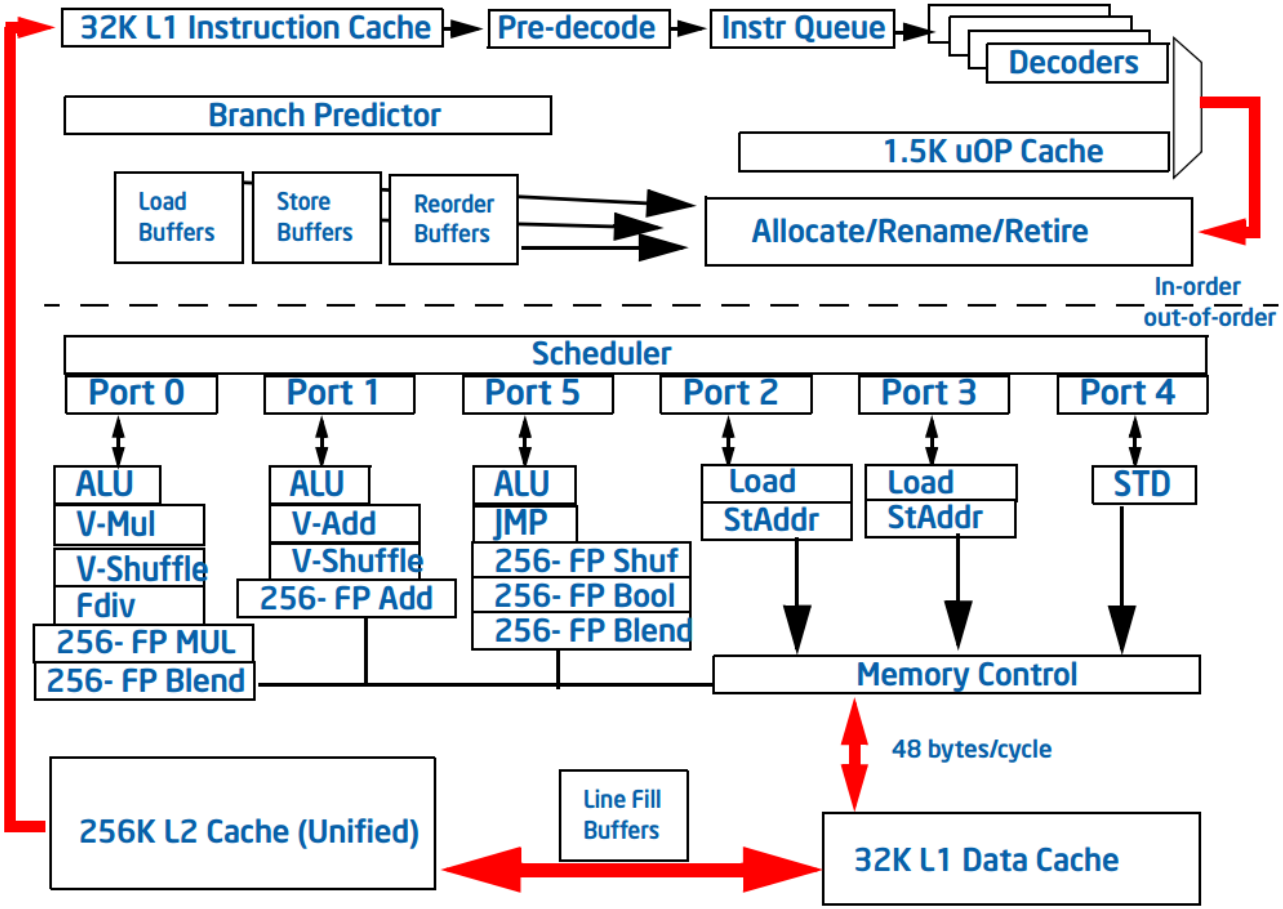
Microarchitecture	Instruction Set	SP FLOPs per Cycle per Core	DP FLOPs per Cycle per Core	L1 Cache Bandwidth (bytes/cycle)	L2 Cache Bandwidth (bytes/cycle)
Nehalem	SSE (128-bits)	8	4	32 (16B read + 16B write)	32
Sandy Bridge	Intel® AVX (256-bits)	16	8	48 (32B read + 16B write)	32
Haswell	Intel® AVX2 (256-bits)	32	16	96 (64B read + 32B write)	64

- Example of **theoretic peak** FLOP rates:
  - Intel® Core™ i7-2710QE (Sandy Bridge):
    2.1 GHz * 16 SP FLOPs * 4 cores = **134.4 SP GFLOPs**
  - Intel® Core™ i7-4765T (Haswell):
    2.0 GHz * 32 SP FLOPs * 4 cores = **256 SP GFLOPs**

Desktop, Mobile & Server

2nd Generation Intel® Core™

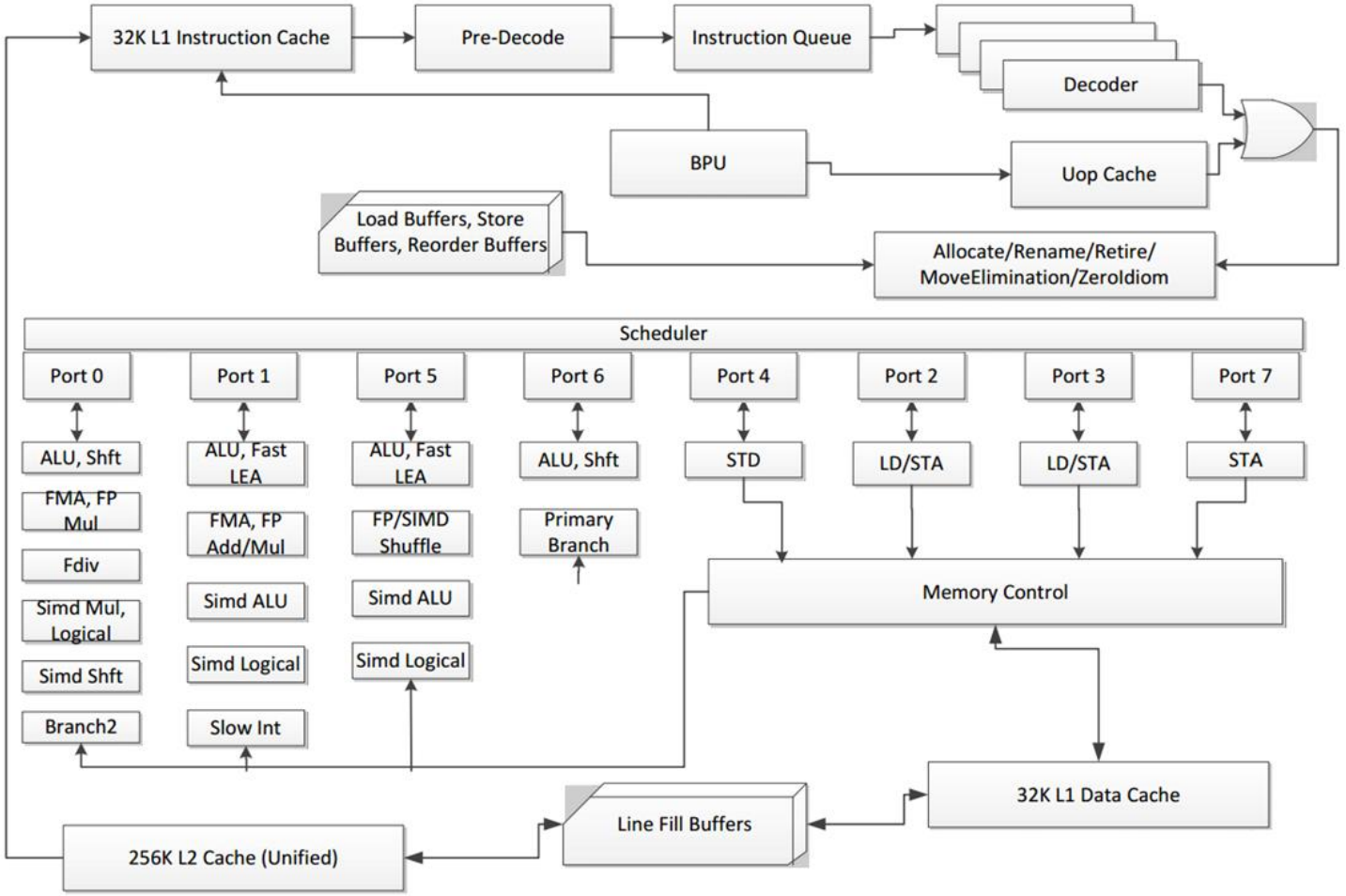
From Intel® 64 and IA-32 Architectures Optimization Reference Manual:



Desktop, Mobile & Server

4th Generation Intel® Core™

From Intel® 64 and IA-32 Architectures Optimization Reference Manual:



Intel® Composer XE

32

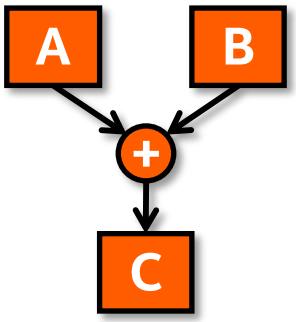
Common Optimization Options

	Windows*	Linux*, OS* X
Disable optimization	/Od	-O0
Optimize for speed (no code size increase)	/O1	-O1
Optimize for speed (default)	/O2	-O2
High-level loop optimization	/O3	-O3
Create symbols for debugging	/Zi	-g
Multi-file inter-procedural optimization	/Qipo	-ipo
Profile guided optimization (multi-step build)	/Qprof-gen /Qprof-use	-prof-gen -prof-use
Optimize for speed across the entire program ("prototype switch") ***fast* options definitions changes over time!**	/fast same as: /O3 /Qipo /Qprec-div-, /fp:fast=2 /QxHost)	-fast same as: Linux: -ipo –O3 -no-prec-div –static –fp-model fast=2 -xHost) OS X: -ipo -mdynamic-no-pic -O3 -no-prec-div -fp-model fast=2 -xHost
OpenMP support	/Qopenmp	-qopenmp
Automatic parallelization	/Qparallel	-parallel

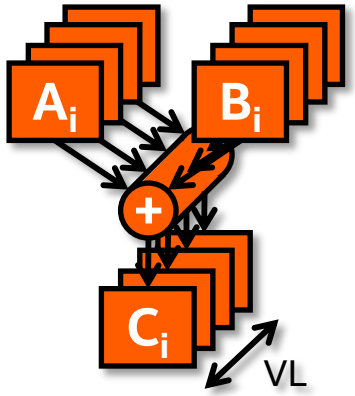
Vectorization

- **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD):
  - Processing vector with a single operation
  - Provides data level parallelism (DLP)
  - Because of DLP more efficient than scalar processing
- **Vector:**
  - Consists of more than one element
  - Elements are of same scalar data types (e.g. floats, integers, …)
- **Vector length (VL):** Elements of the vector

Scalar Processing

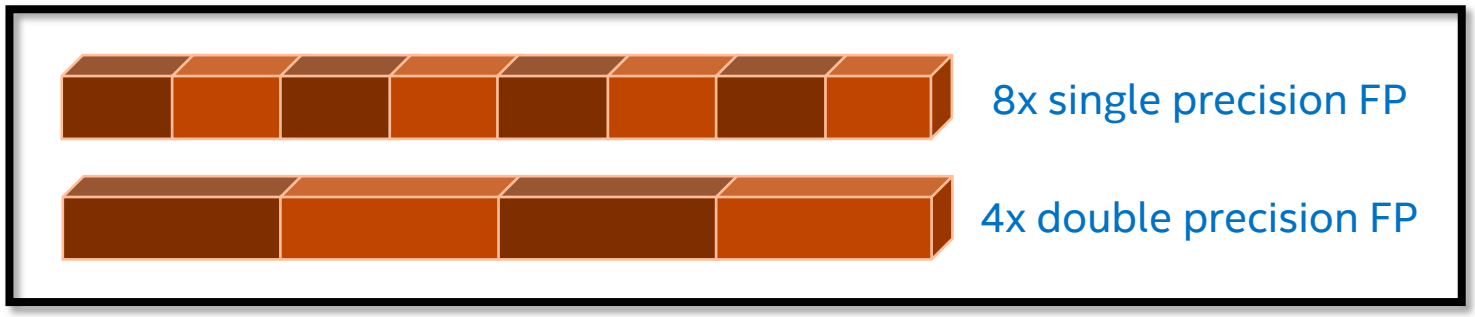


Vector Processing

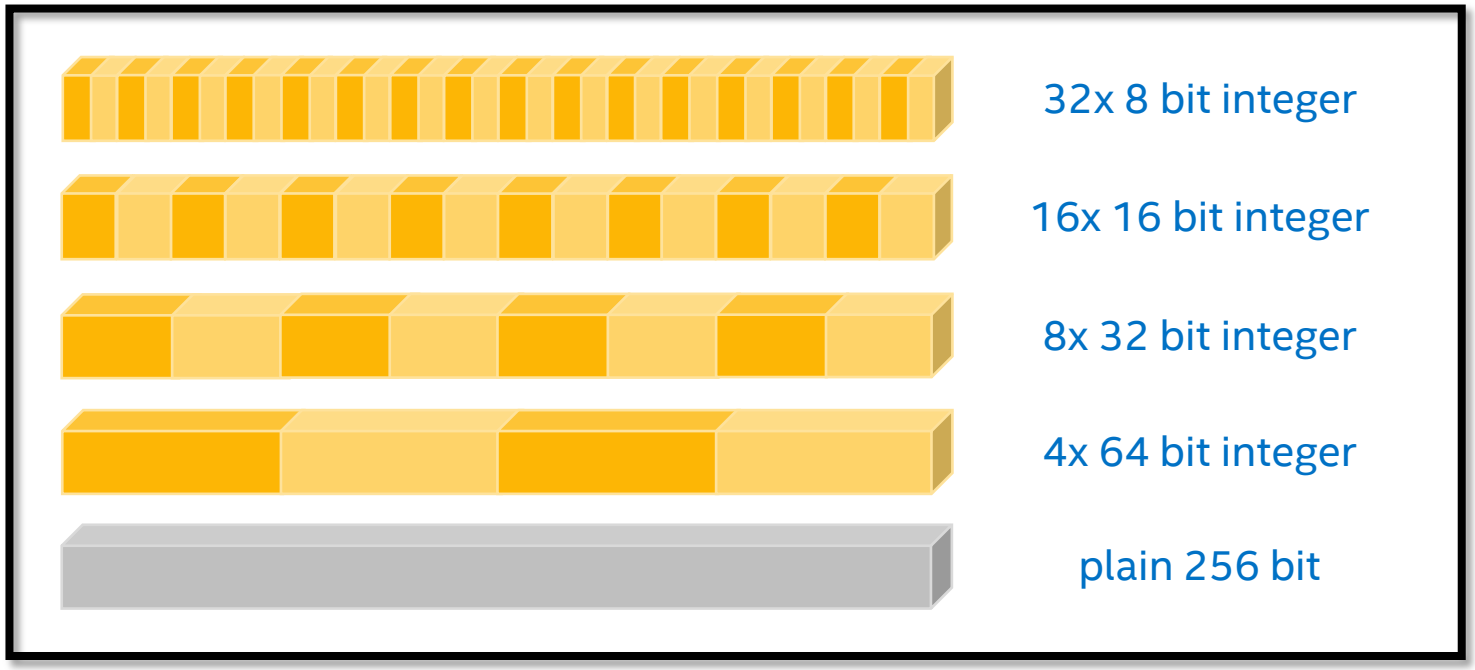


AVX Vector Types

Intel® AVX



Intel® AVX2



Intel® AVX2 I

- Basically same as Intel® AVX with following additions:
  - **Doubles** width of **integer vector** instructions to 256 bits
  - Floating point fused multiply add (**FMA**)

Processor Family	Instruction Set	Single Precision FLOPs Per Clock	Double Precision FLOPs Per Clock	
Pre 2nd generation Intel® Core™ Processors	SSE 4.2	8	4	
2nd and 3rd generation Intel® Core™ Processors	AVX	16	8	2x
4th generation Intel® Core™ Processors	**AVX2**	**32**	**16**	**4x**

  - Any-to-any permutes
  - Vector-vector shifts

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

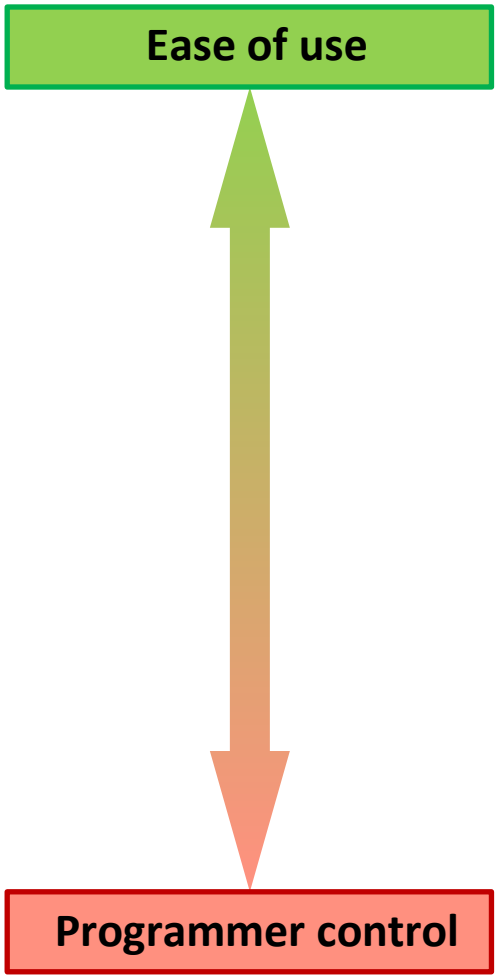
Compiler:
Auto-vectorization hints (`#pragma vector`, …)

Compiler:
Intel® Cilk™ Plus Array Notation Extensions

SIMD intrinsic class
(e.g.: `F32vec`, `F64vec`, …)

Vector intrinsic
(e.g.: `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)

Assembler code
(e.g.: `[v]addps`, `[v]addss`, …)



Basic Vectorization Switches I

- Linux*, OS X*: **-x<feature>**, Windows*: **/Qx<feature>**
  - Might enable Intel processor specific optimizations
  - Processor-check added to “main” routine:
    Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message

- Linux*, OS X*: **-ax<features>**, Windows*: **/Qax<features>**
  - Multiple code paths: baseline and optimized/processor-specific
  - Optimized code paths for Intel processors defined by **<features>**
  - Multiple SIMD features/paths possible, e.g.: **-axSSE2,AVX**
  - Baseline code path defaults to **-msse2** (**/arch:sse2**)
  - The baseline code path can be modified by **-m<feature>** or **-x<feature>** (**/arch:<feature>** or **/Qx<feature>**)

Basic Vectorization Switches II

- Linux*, OS X*: **-m<feature>**, Windows*: **/arch:<feature>**
  - Neither check nor specific optimizations for Intel processors: Application optimized for both Intel and non-Intel processors for selected SIMD feature
  - Missing check can cause application to fail in case extension not available
- Default for Linux*: **-msse2**, Windows*: **/arch:sse2**:
  - Activated implicitly
  - Implies the need for a target processor with at least Intel® SSE2
- Default for OS X*: **-xsse3** (IA-32), **-xssse3** (Intel® 64)

- For 32 bit compilation, **-mia32** (**/arch:ia32**) can be used in case target processor does not support Intel® SSE2 (e.g. Intel® Pentium® 3 or older)

Basic Vectorization Switches III

- Special switch for Linux*, OS X*: **-xHost**, Windows*: **/QxHost**
  - Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
  - Code only executes on processors with same SIMD feature or later as on build host
  - As for **-x<feature>** or **/Qx<feature>**, if "main" routine is built with **-xHost** or **/QxHost** the final executable only runs on Intel processors

Vectorization Pragma/Directive

- SIMD features can also be set on a function/subroutine level via pragmas/directives:
  - C/C++:
    `#pragma intel optimization_parameter target_arch=<CPU>`
  - Fortran:
    `!DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:TARGET_ARCH= <CPU>`
- Examples:
  - C/C++:

```
#pragma intel optimization_parameter target_arch=AVX
void optimized_for_AVX()
{
  …
}
```

  - Fortran:

```
function optimized_for_AVX()
!DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:TARGET_ARCH=AVX
  …
end function
```

Control Vectorization I

- Disable vectorization:
  - Globally via switch:
    Linux*, OS X*: **`-no-vec`**, Windows*: **`/Qvec-`**
  - For a single loop:
    C/C++: **`#pragma novector`**, Fortran: **`!DIR$ NOVECTOR`**
  - Compiler still can use some SIMD features
- Using vectorization:
  - Globally via switch (default for optimization level 2 and higher):
    Linux*, OS X*: **`-vec`**, Windows*: **`/Qvec`**
  - Enforce for a single loop (override compiler efficiency heuristic) if semantically correct:
    C/C++: **`#pragma vector always`**, Fortran: **`!DIR$ VECTOR ALWAYS`**
  - Influence efficiency heuristics threshold:
    Linux*, OS X*: **`-vec-threshold[n]`**
    Windows*: **`/Qvec-threshold[[:]n]`**
    `n`: **`100`** (default; only if profitable) … **`0`** (always)

Control Vectorization II

- Verify vectorization:
  - Globally:
    Linux*, OS X*: **`-opt-repot`**, Windows*: **`/Qopt-report`**
  - Abort compilation if loop cannot be vectorized:
    C/C++: **`#pragma vector always assert`**
    Fortran: **`!DIR$ VECTOR ALWAYS ASSERT`**
- Advanced:
  - Ignore vector dependencies (IVDEP):
    C/C++: **`#pragma ivdep`**
    Fortran: **`!DIR$ IVDEP`**
  - "Enforce" vectorization:
    C/C++: **`#pragma simd`**
    Fortran: **`!DIR$ SIMD`**

How to detect Vectorization?

- We need feedback! Compiling with a flag below reveals a short report showing which loops are vectorized and the reason for refusing vectorisation for certain loops:
  - **`-opt-report=<n>`**
    n = 0-5 increases the level of detail, 2 is the default
    `-opt-report-phase=vec`
    shows just vectorization
- Profile the code, compile without vectorization (-no-vec) and profile again. Look for code sections with different timing
- Look into assembly. Can be done by using Intel® VTune™ Amplifier XE

How to start?

- Compile with minimal options and check timing
- Compile with -xHost and –opt-report=5 and check timing
- Compile with –xHost and –no-vec
  disables vectorization. Compare with previous timing

- Use loopprofiler for single threaded profile on loop basis:

  icc –profile-functions – profile-loops=all –profile-loops-report=2 …

  view results with loopprofilerviewer.sh <datafile>
- New Book: **Optimizing HPC Applications with Intel Cluster Tools**

Other Intel® HPC tools

Intel® MPI

Intel® VTune™ Amplifier XE

Intel® Trace Analyzer and Collector (ITAC)

Intel® Math Kernel Library (MKL)

Intel® MPI Benchmarks (IMB)

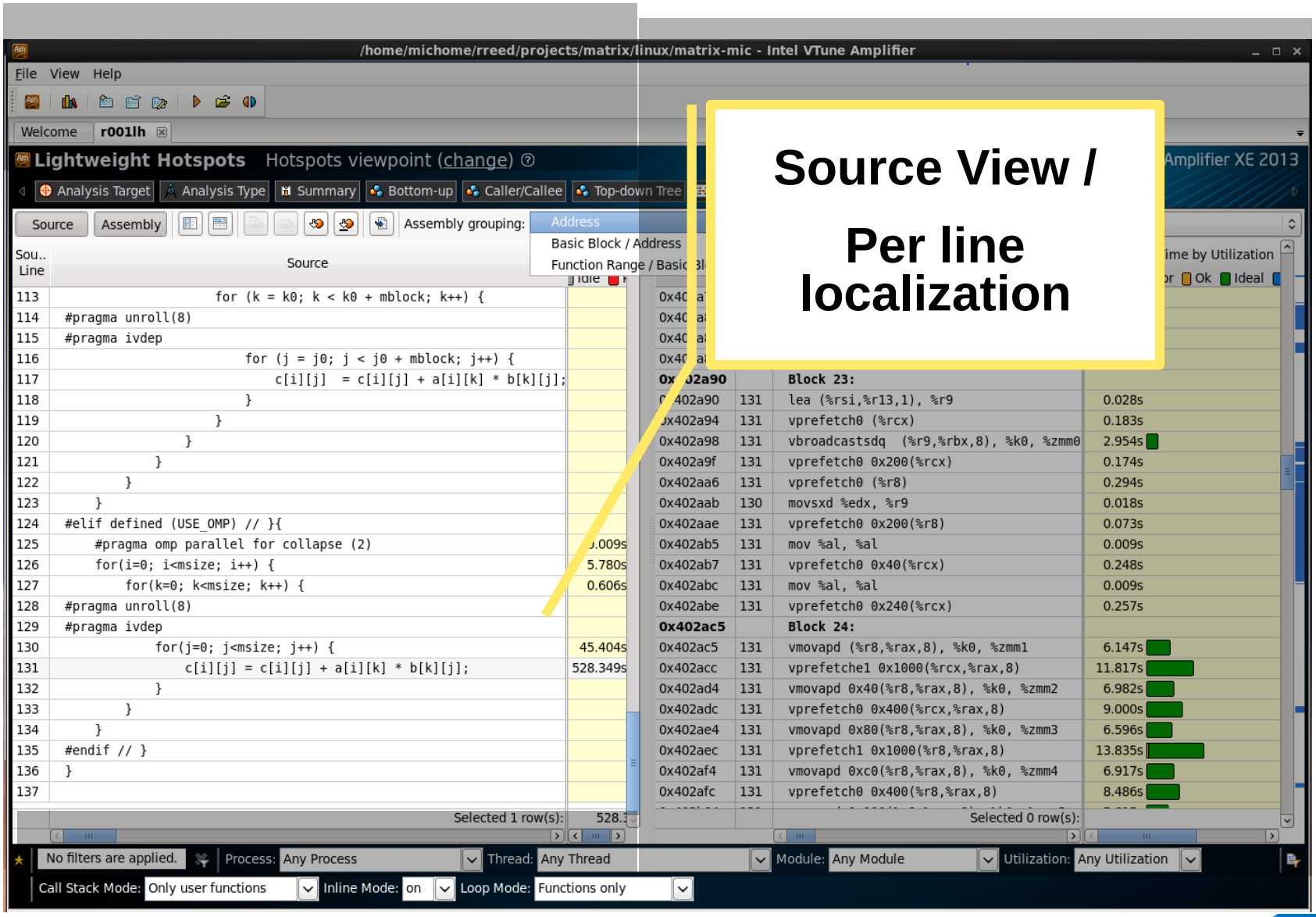
Intel® Inspector XE

Intel® Advisor

All Tools available in Intel® Cluster Studio XE (ICS)

http://software.intel.com/en-us/articles/intel-cluster-studio-xe/

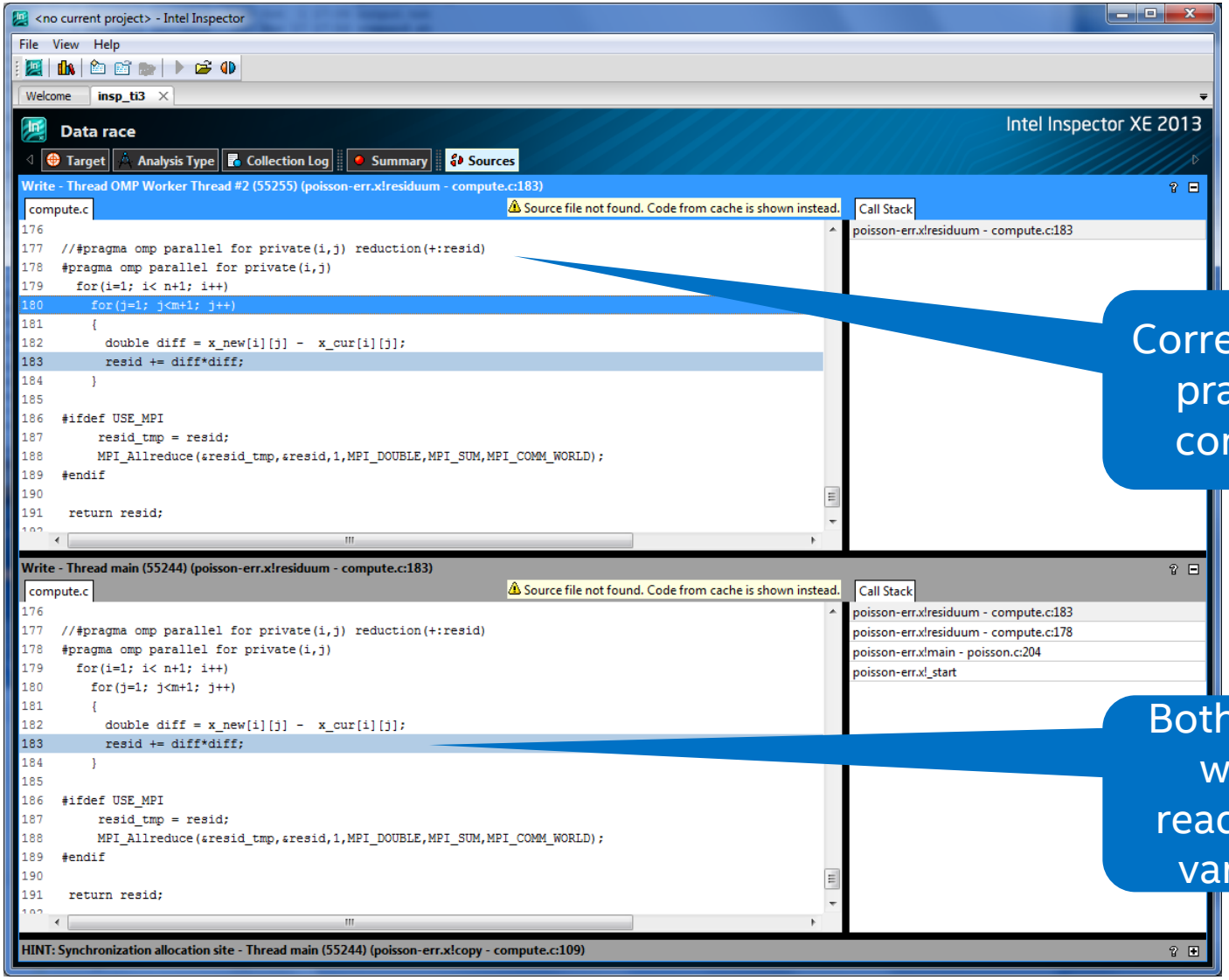
VTune™ Amplifier XE hotspots



Intel® Inspector XE Threading Correctness

- Inspector XE simulates all possible orders of data access
- Not only errors in this run. Inspector XE finds all possible errors even if they don't happen in this run!
- Typical race conditions in OpenMP will be due to forgotten variables in the private or reduction clause
- Following example shows a forgotten reduction clause
- Inspector can be used to find all necessary variables for the private/reduction clause!

Inspector XE – Data Races



Correct OpenMP pragma was commented

Both threads are writing and reading to/from variable resid

Legal Disclaimer & Optimization Notice

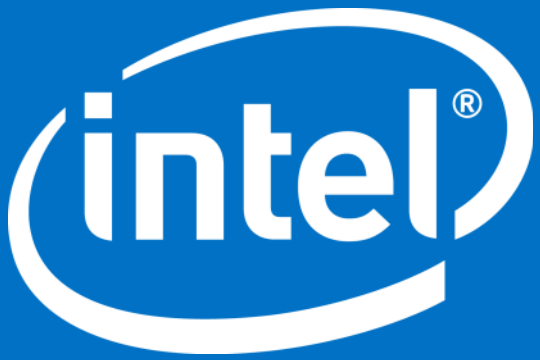
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804


intel®